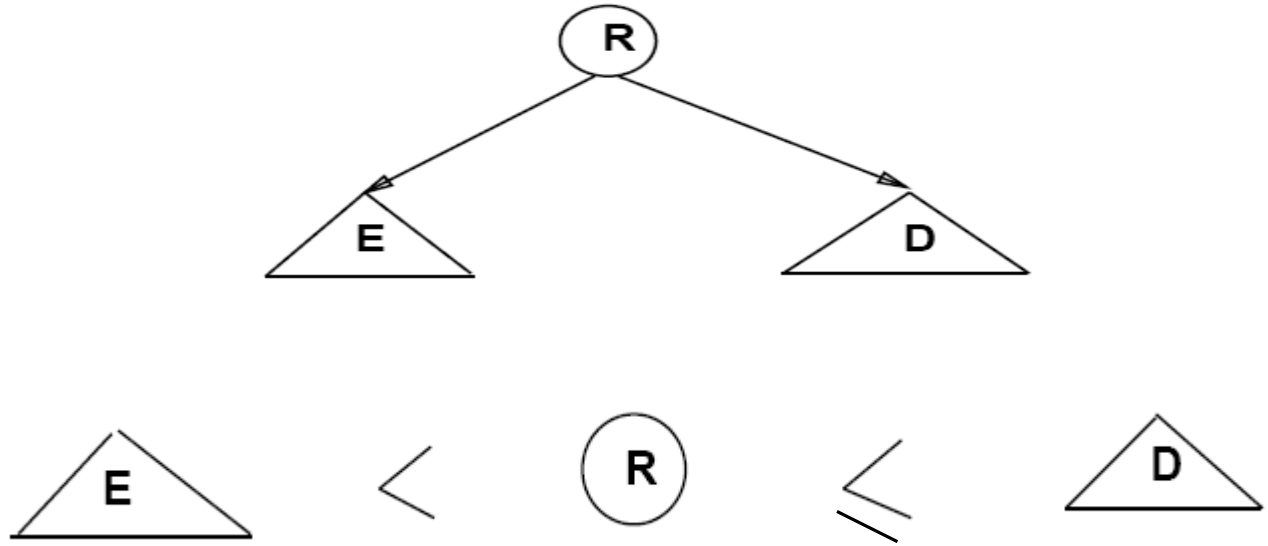
Árvores de Pesquisa sem Balanceamento

Algoritmos e Estruturas de Dados II

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

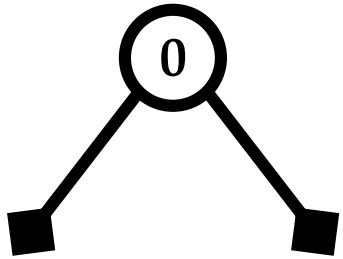


Árvore Binária de Pesquisa

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

struct arvore *cria_arvore(Registro reg) {
struct arvore *novo;

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

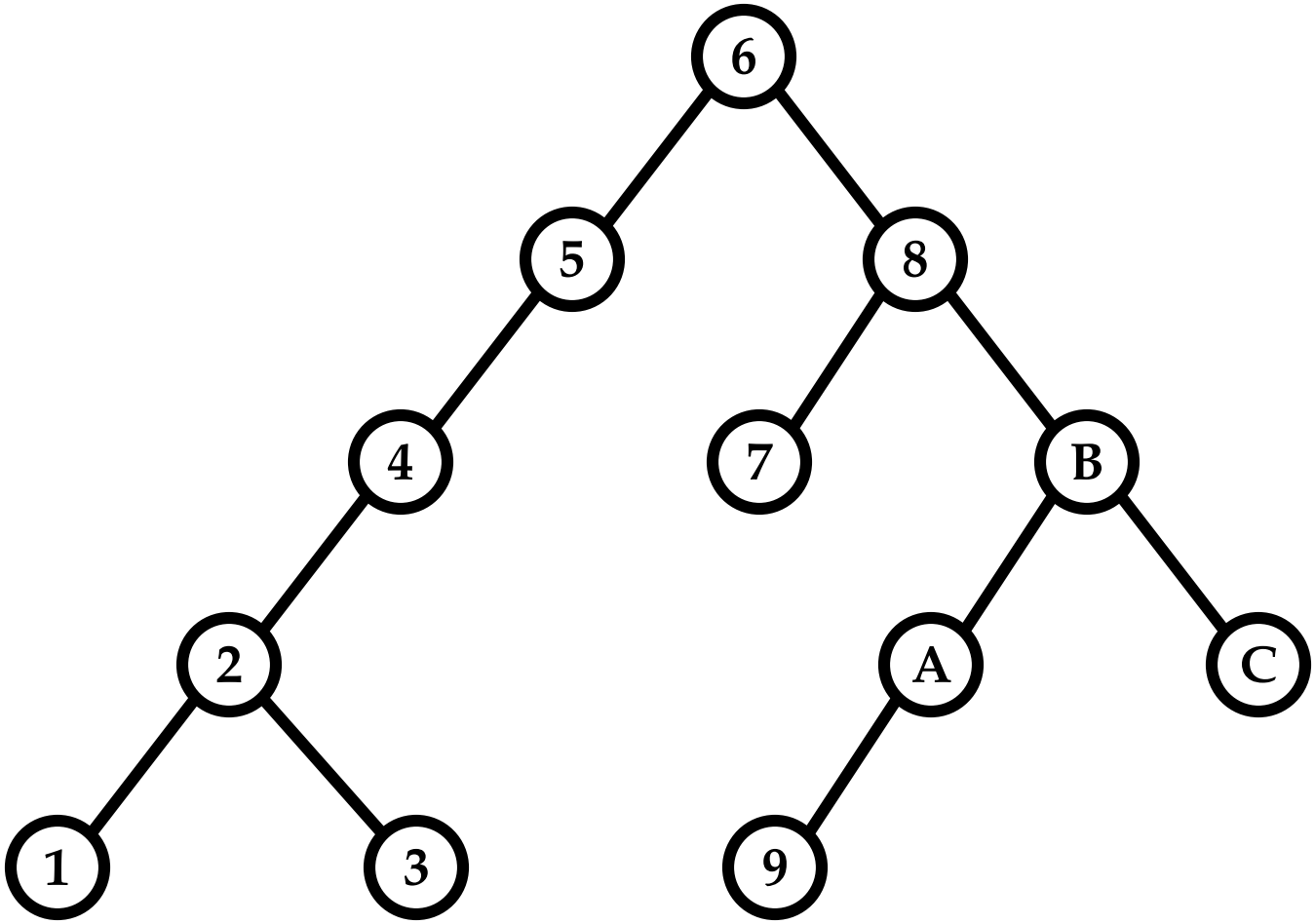


Árvore Binária de Pesquisa: Busca

```
void Pesquisa(Registro *x, struct arvore *t) {


    if (t == NULL) {
       printf("Registro não esta presente na árvore\n");
    }
    else  if (x->Chave < t->reg.Chave)
       Pesquisa(x, t->Esq); /* busca no filho esquerdo */
    else if (x->Chave > t->reg.Chave)
       Pesquisa(x, t->Dir); /* busca no filho direito */
    else
       *x = t->reg;
}
```

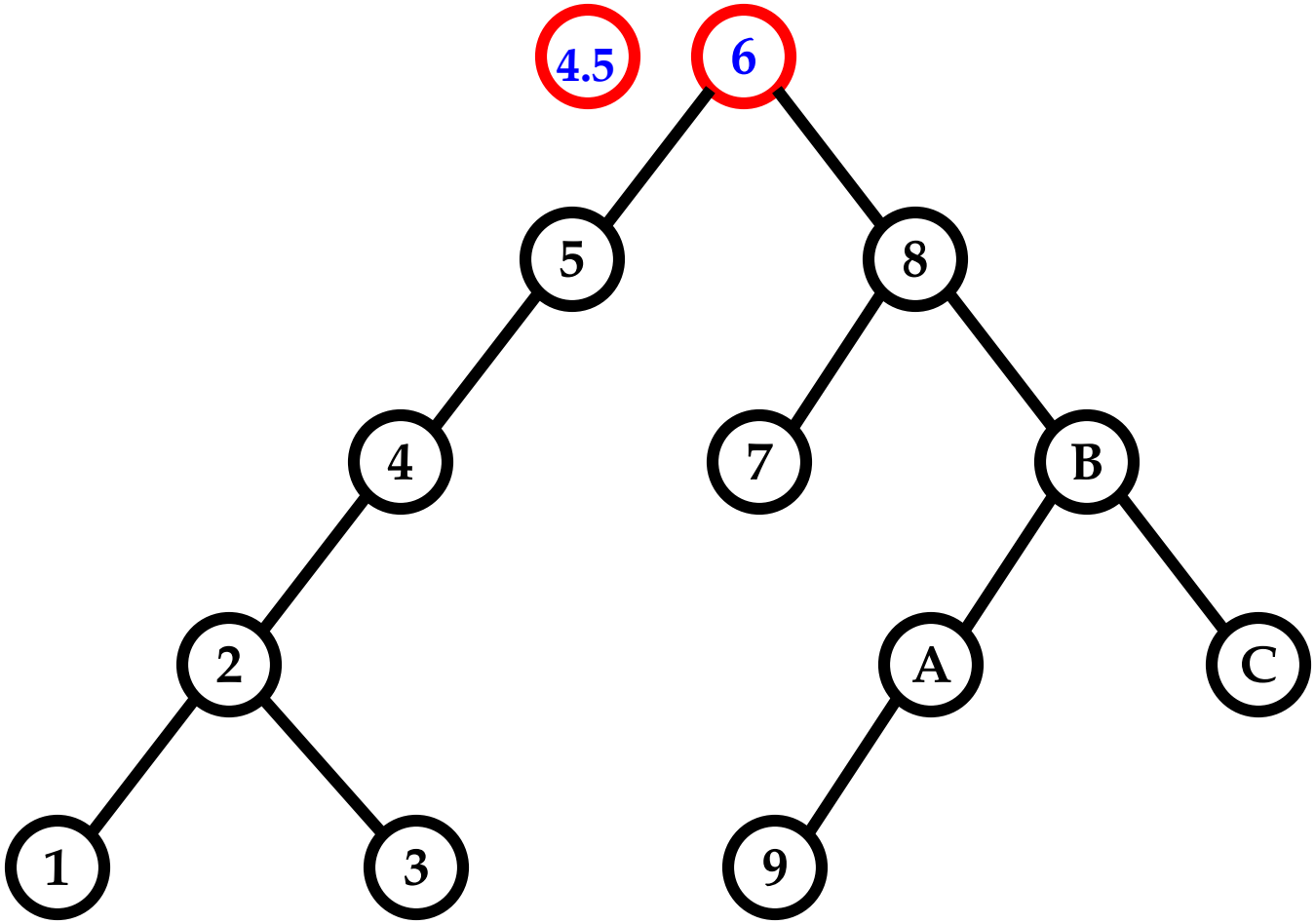
Árvore Binária de Pesquisa: Busca



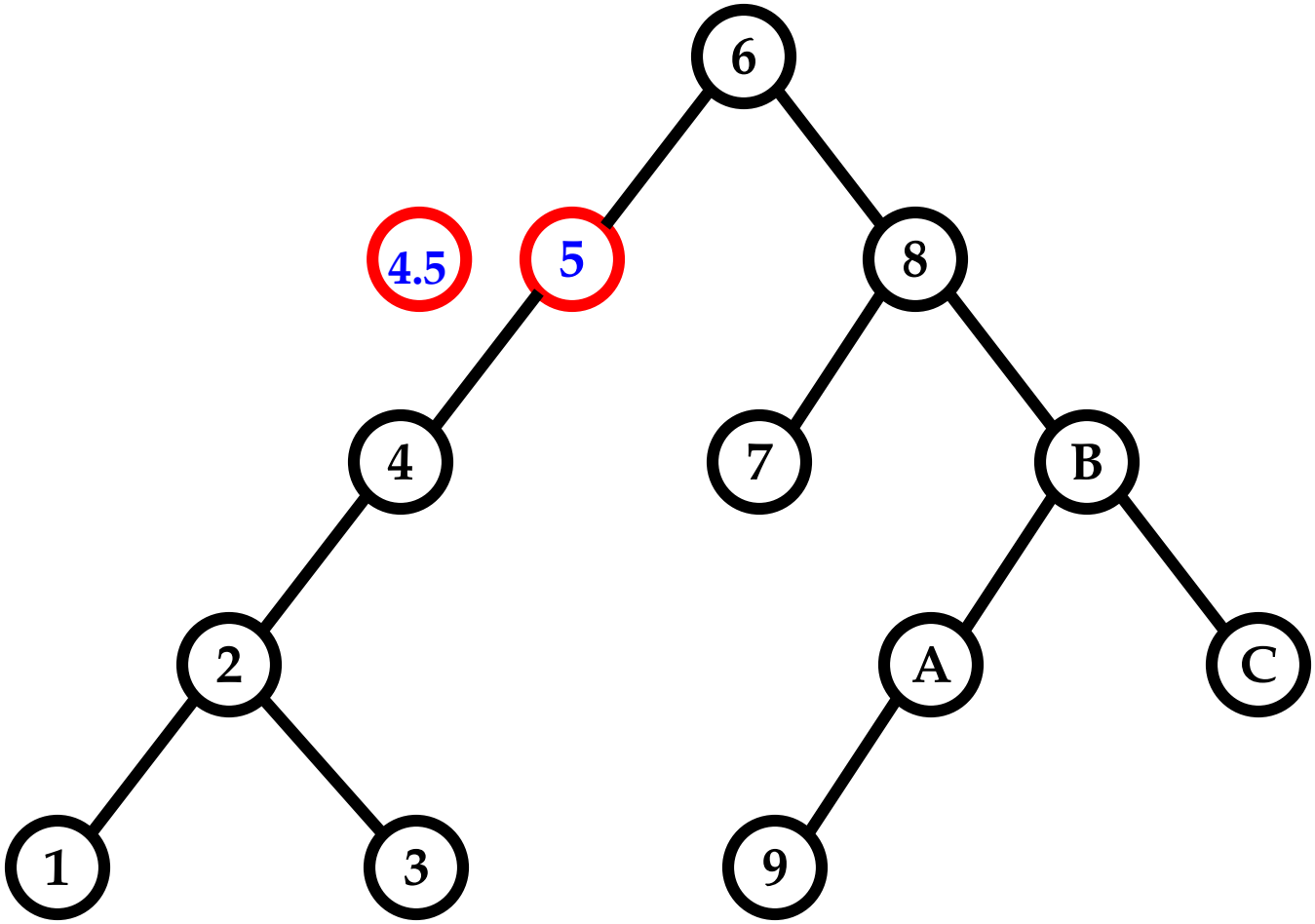
Árvore Binária de Pesquisa: Inserção

- 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

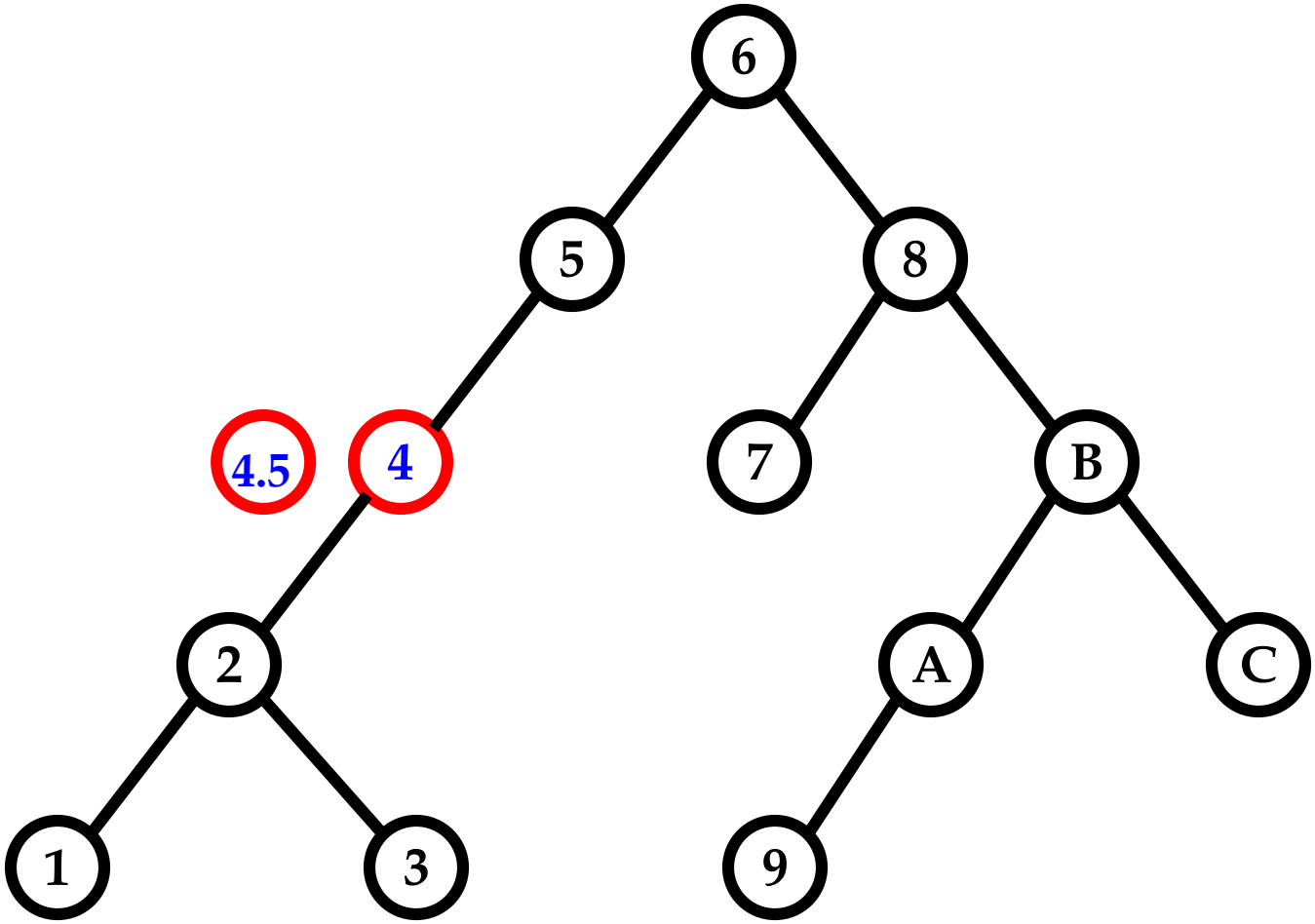
Árvore Binária de Pesquisa: Inserção



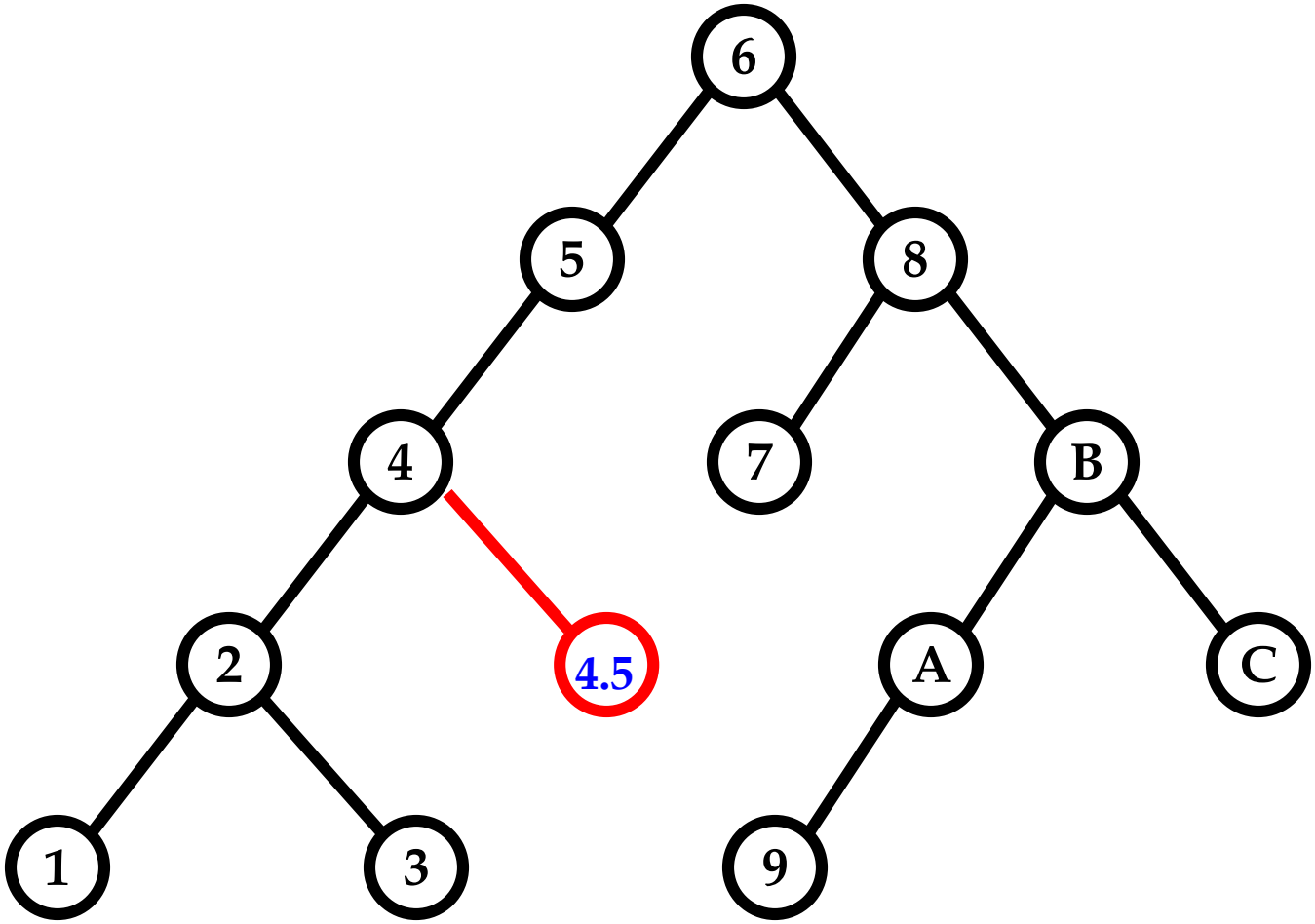
Árvore Binária de Pesquisa: Inserção



Árvore Binária de Pesquisa: Inserção



Árvore Binária de Pesquisa: Inserção

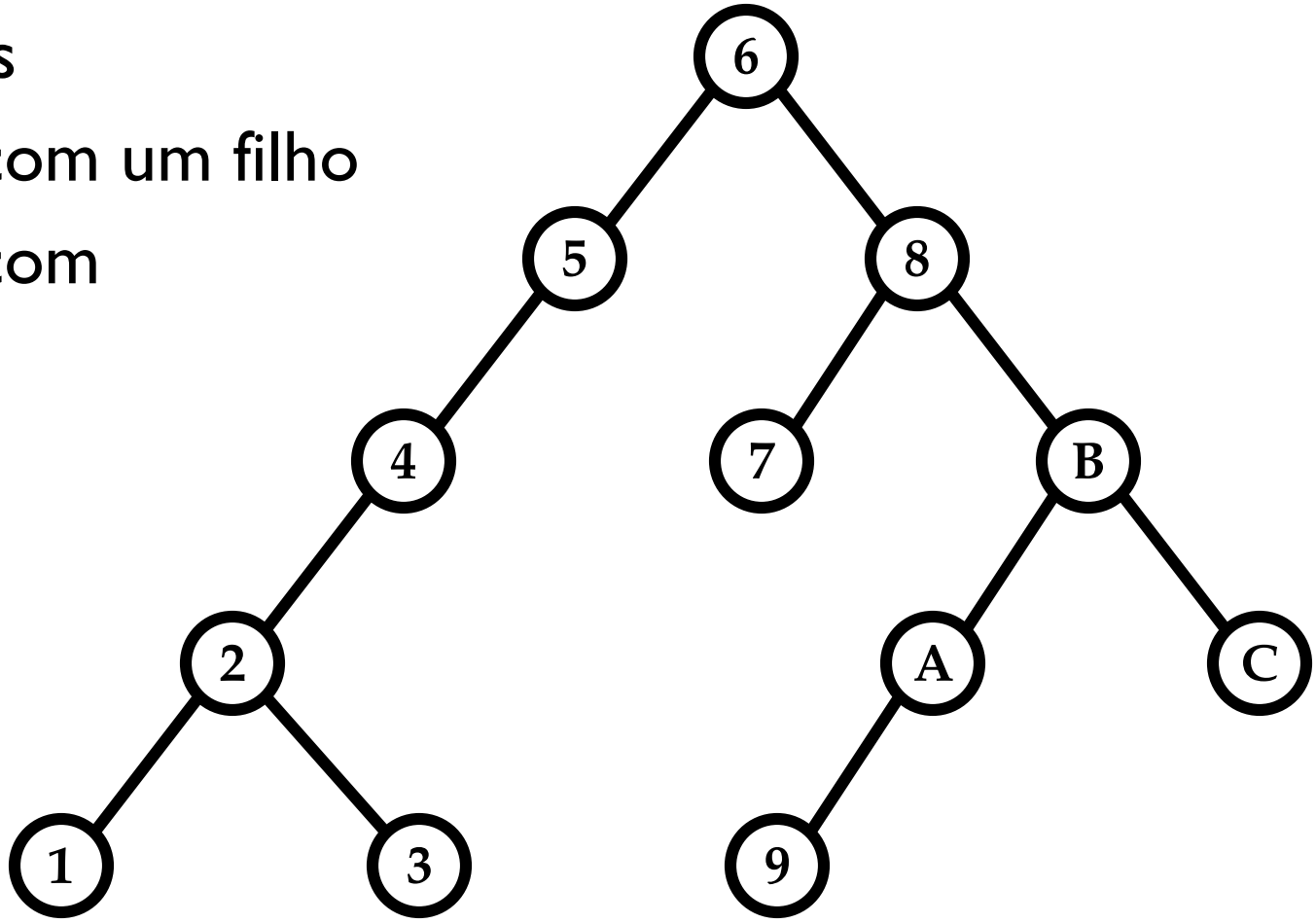


Árvore Binária de Pesquisa: Inserção

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

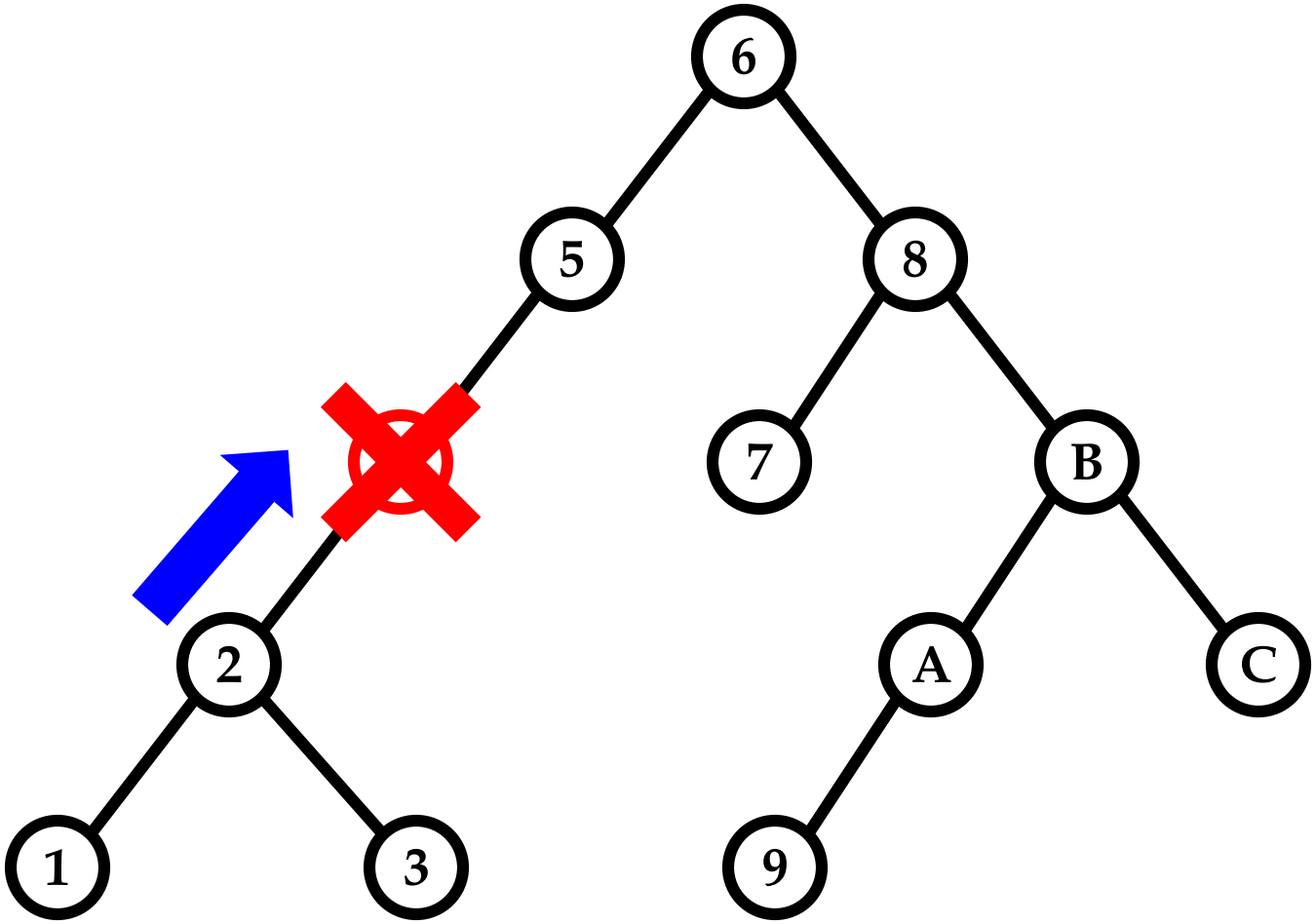
Árvore Binária de Pesquisa: Remoção

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



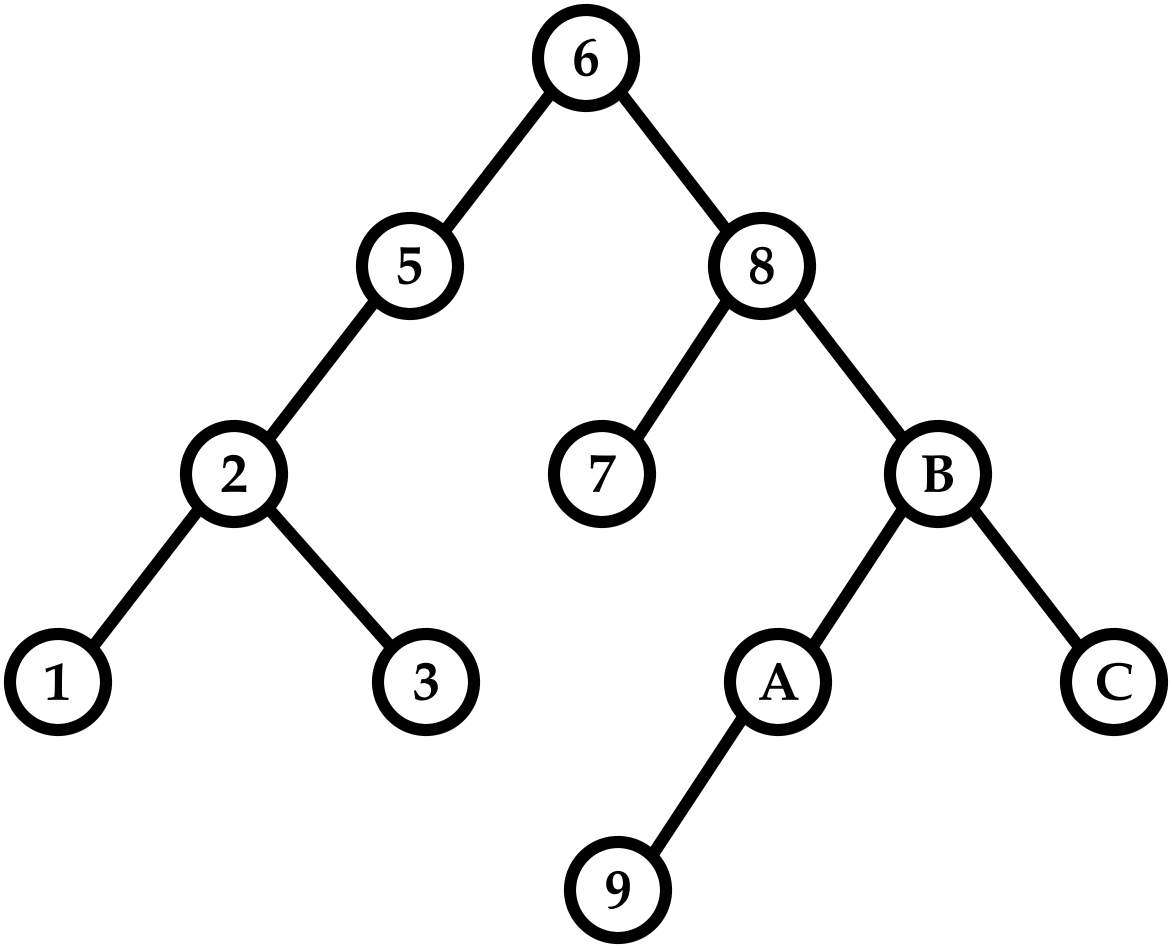
Árvore Binária de Pesquisa: Remoção

- Nó com 1 filho



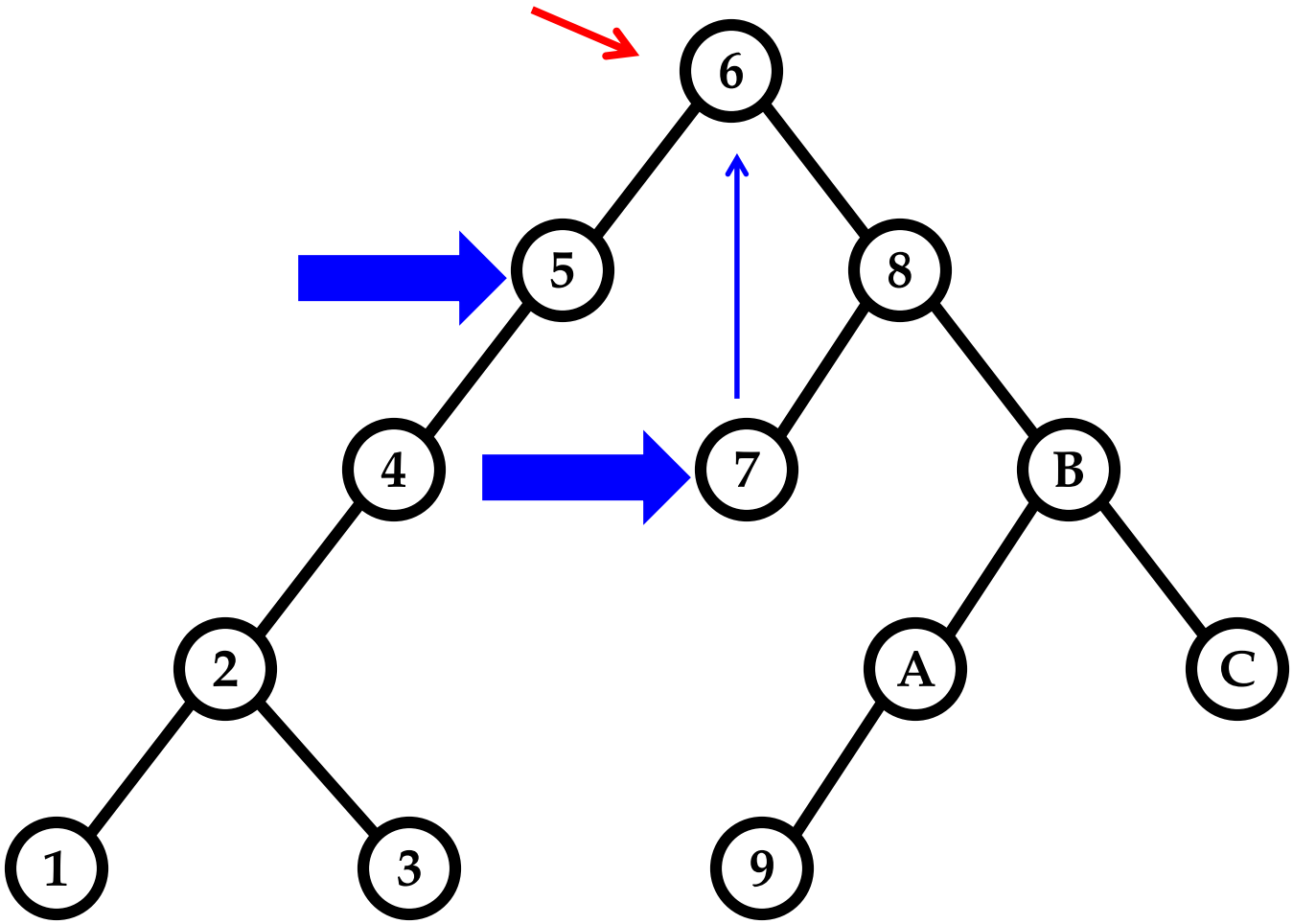
Árvore Binária de Pesquisa: Remoção

- Nó com 1 filho



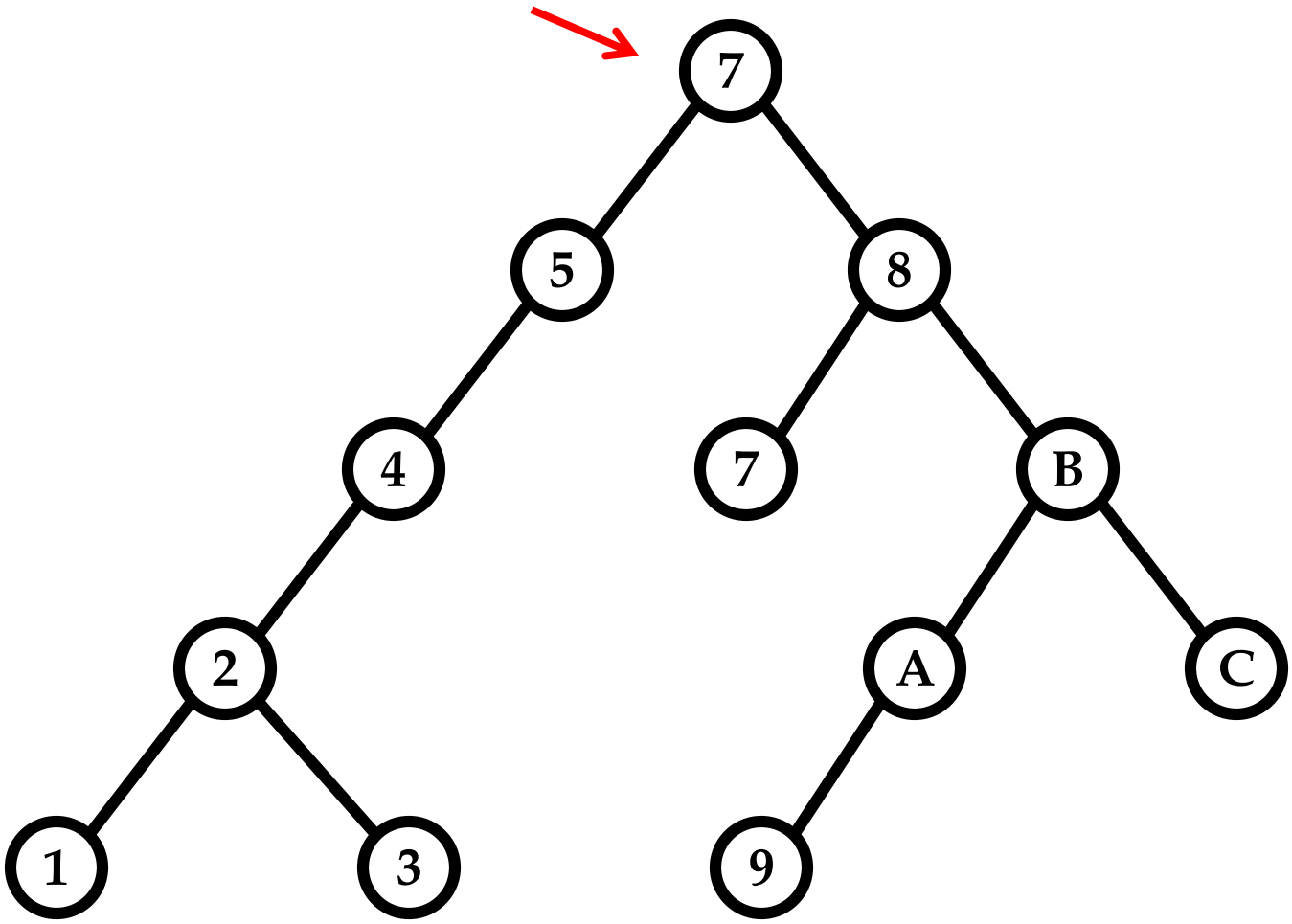
Árvore Binária de Pesquisa: Remoção

- Nó com 2 filhos



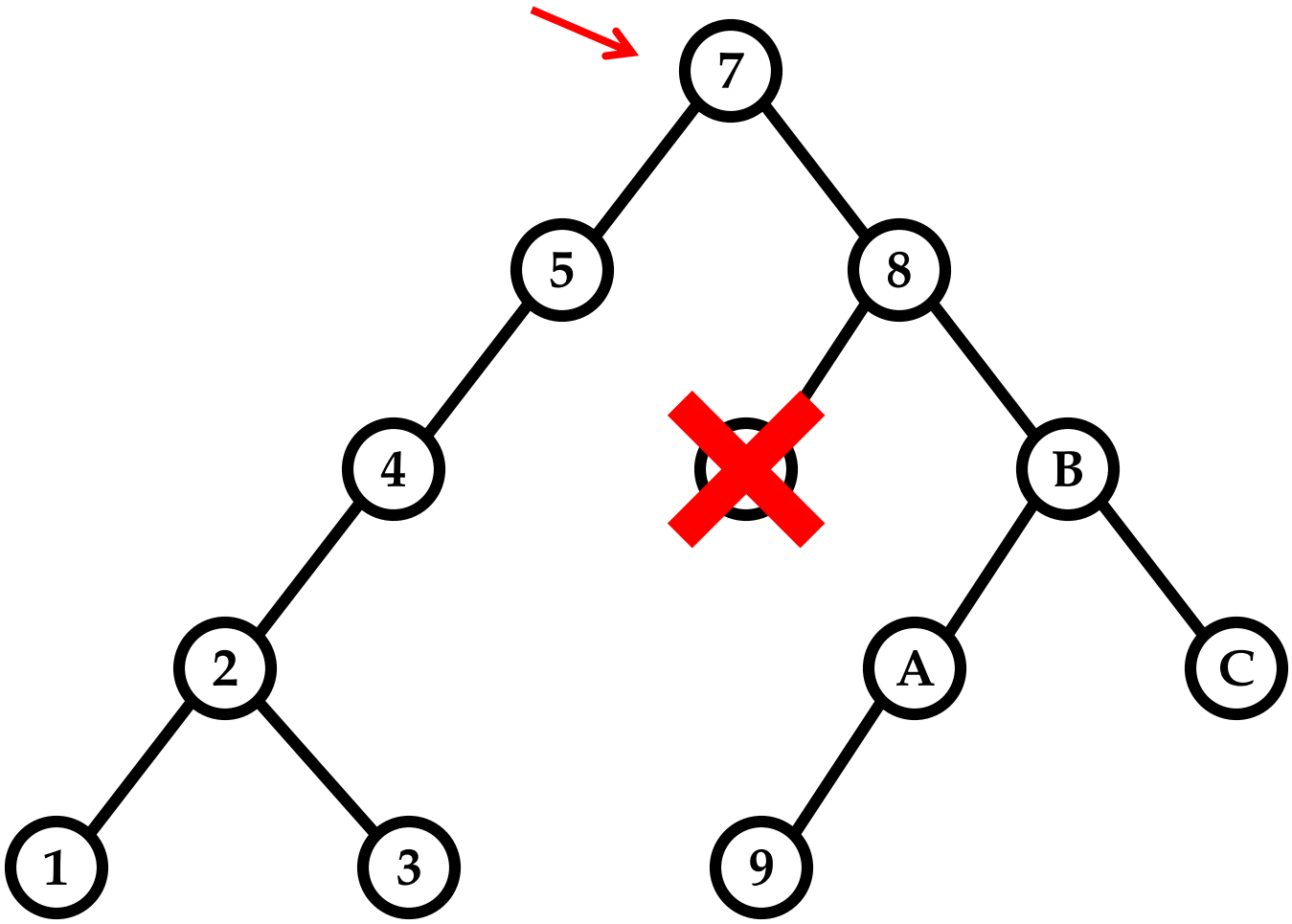
Árvore Binária de Pesquisa: Remoção

- Nó com 2 filhos



Árvore Binária de Pesquisa: Remoção

- Nó com 2 filhos



Árvore Binária de Pesquisa: Remoção

```
struct arvore *remove(struct arvore *t, TipoChave 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;
}
```

Árvore Binária de Pesquisa: Remoção

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