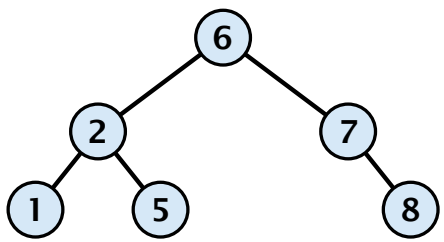
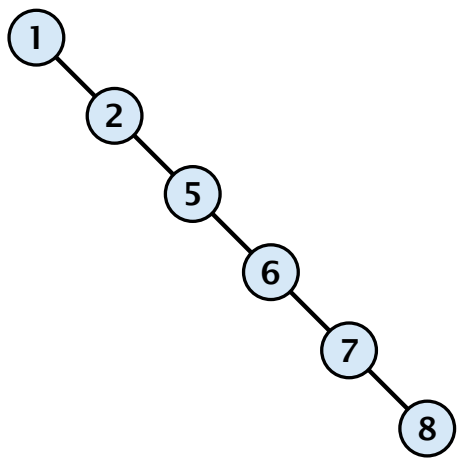
7.4 Binäre Suchbäume

Ein binärer Suchbaum speichert Elemente in einem binären Baum. Jeder Baumknoten enthält ein Element. Alle Elemente im linken Teilbaum eines Knotens v haben einen kleineren Schlüssel als $\text{key}[v]$; Elemente im rechten Teilbaum haben einen größeren Schlüssel. (Annahme: alle Schlüssel sind unterschiedlich).

Beispiele:

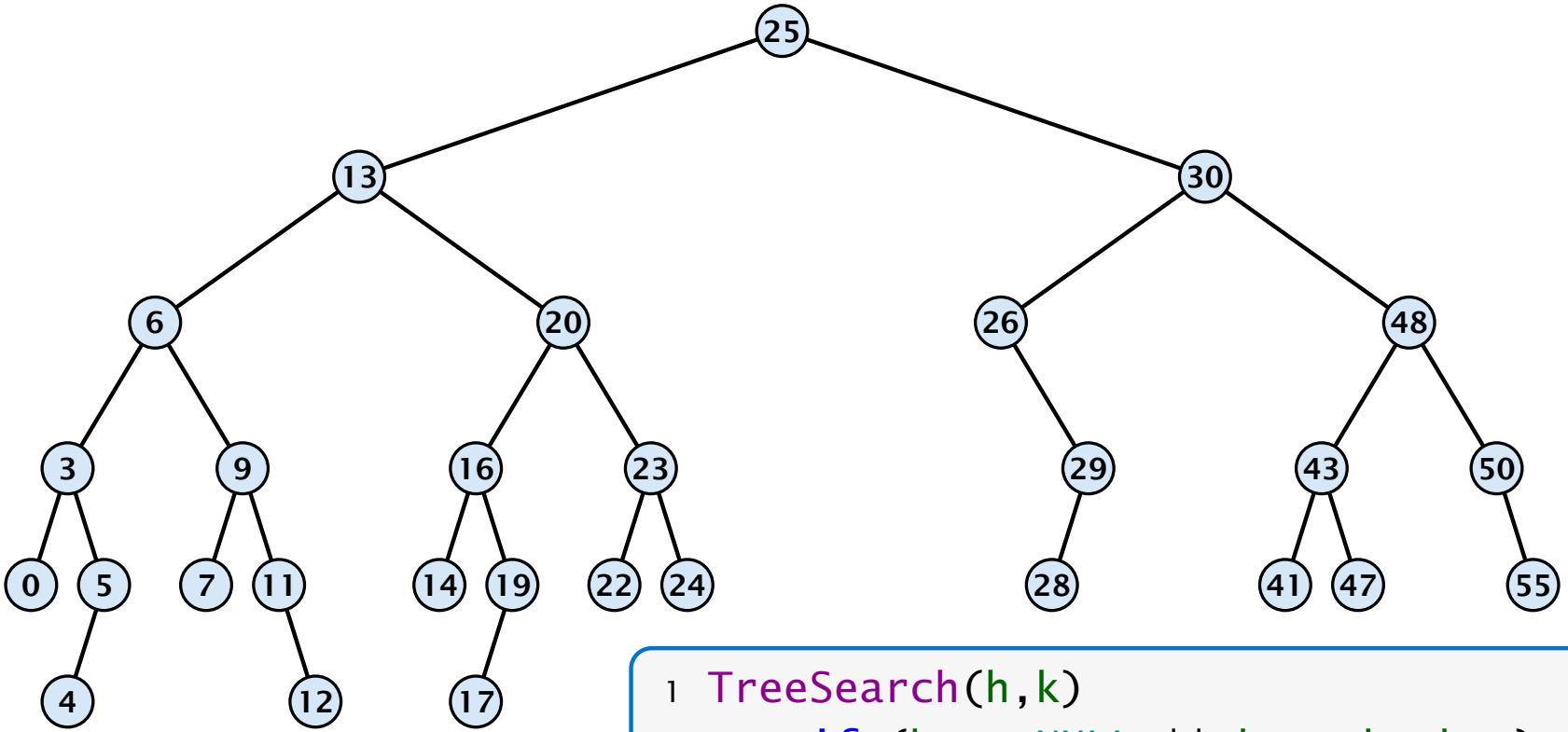




7.4 Binäre Suchbäume

- **`T.insert(x)`** Fügt Objekt `x` ein; `T` darf kein Objekt mit Schlüssel `key[x]` enthalten.
- **`T.delete(h)`** Entfernt durch handle `h` referenziertes Objekt aus `T`.
- **`T.search(k)`** Gibt `handle` auf in `T` gespeichertes Objekt mit Schlüssel `k` zurück falls existent. Sonst `NULL`.
- **`T.successor(h)`** Gibt `handle` auf Nachfolger von durch `h` referenziertes Objekt zurück; falls existent. Sonst `NULL`.
- **`T.predecessor(h)`** Gibt `handle` auf Vorgänger von durch `h` referenziertes Objekt zurück; falls existent. Sonst `NULL`.
- **`T.minimum()`** Gibt `handle` auf in `T` gespeichertes Objekt mit kleinstem Schlüssel zurück falls existent. Sonst `NULL`.
- **`T.maximum()`** Gibt `handle` auf in `T` gespeichertes Objekt mit größtem Schlüssel zurück falls existent. Sonst `NULL`.

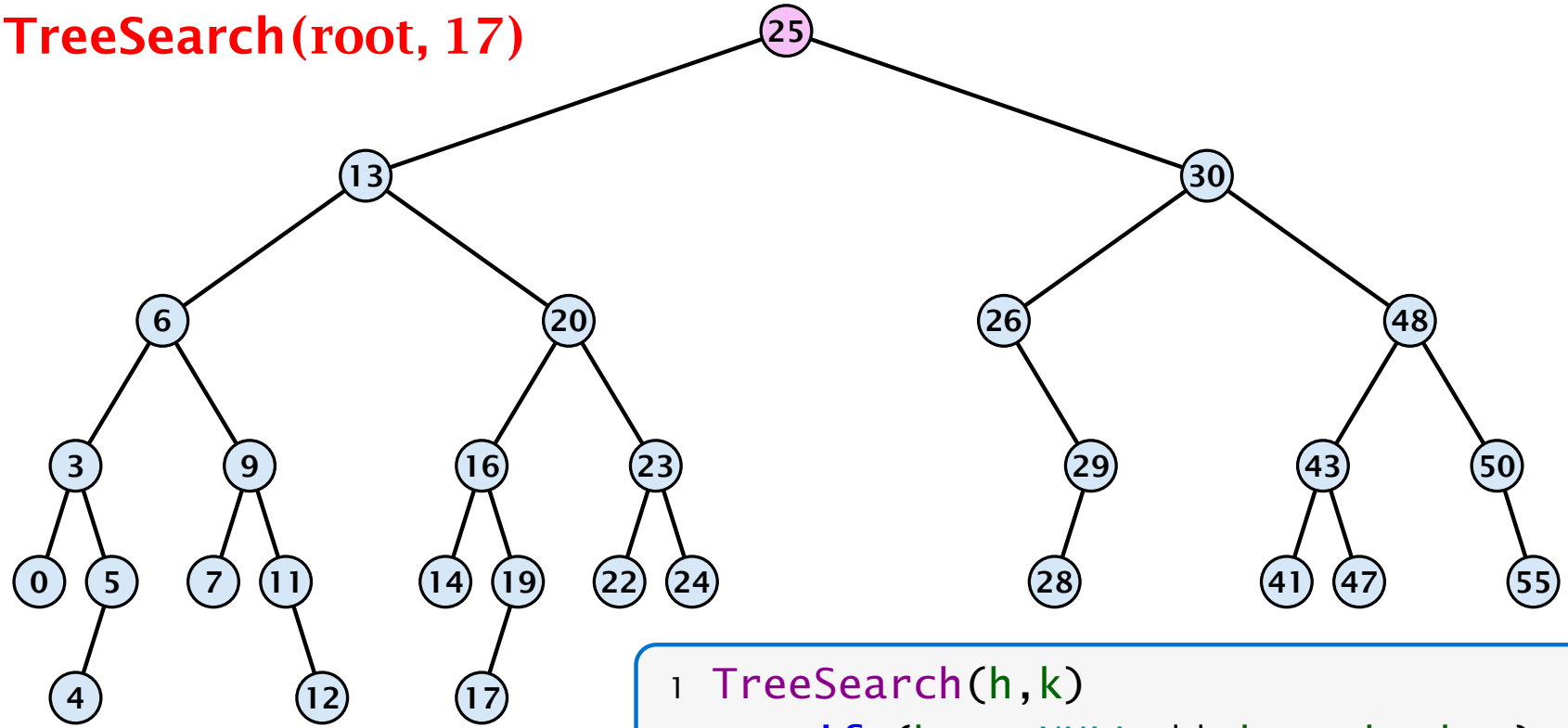
Binäre Suchbäume: Suchen




```
1 TreeSearch(h,k)
2     if (h == NULL || k == h->key)
3         return h;
4     if (k < h->key)
5         return TreeSearch(h->left,k);
6     else
7         return TreeSearch(h->right,k);
```

Binäre Suchbäume: Suchen

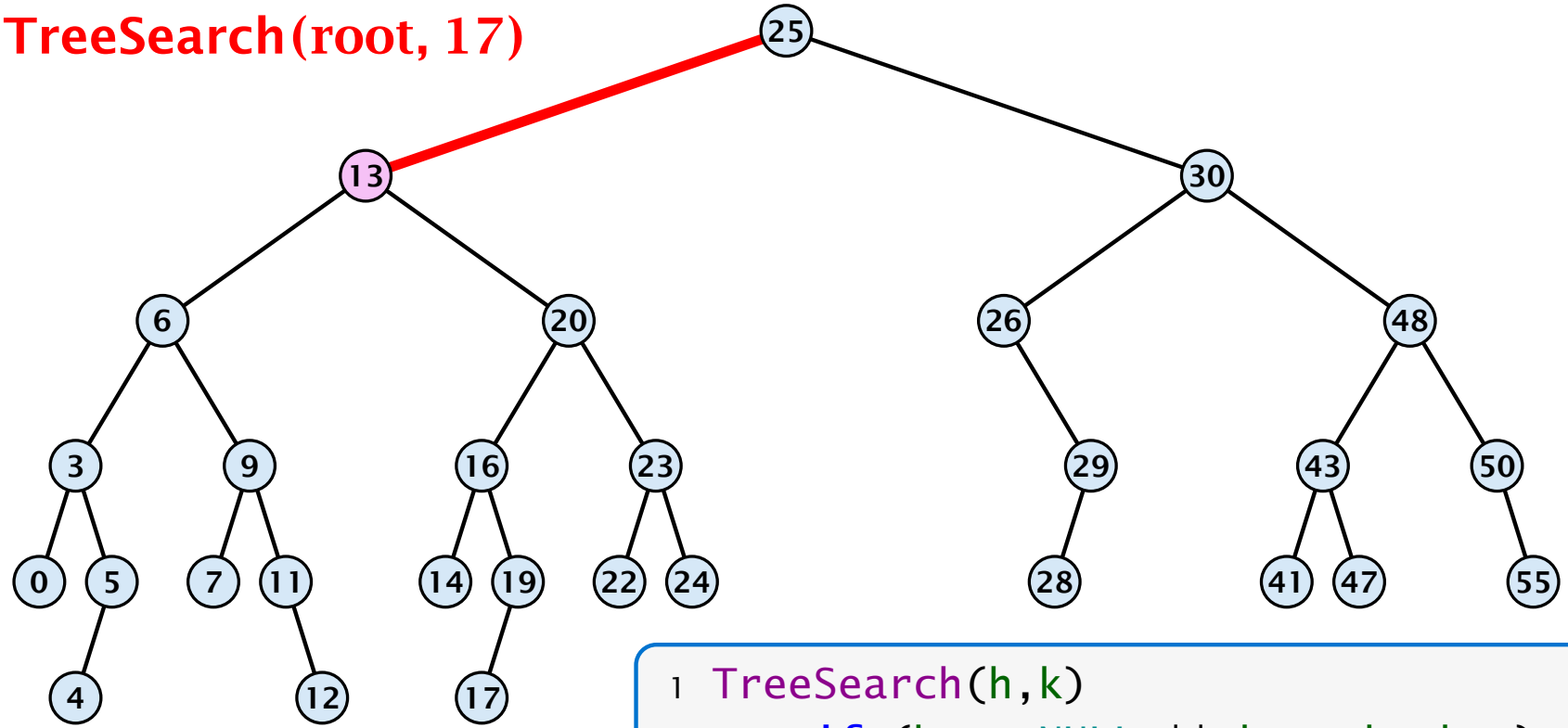
TreeSearch(root, 17)




```
1 TreeSearch(h,k)
2     if (h == NULL || k == h->key)
3         return h;
4     if (k < h->key)
5         return TreeSearch(h->left,k);
6     else
7         return TreeSearch(h->right,k);
```

Binäre Suchbäume: Suchen

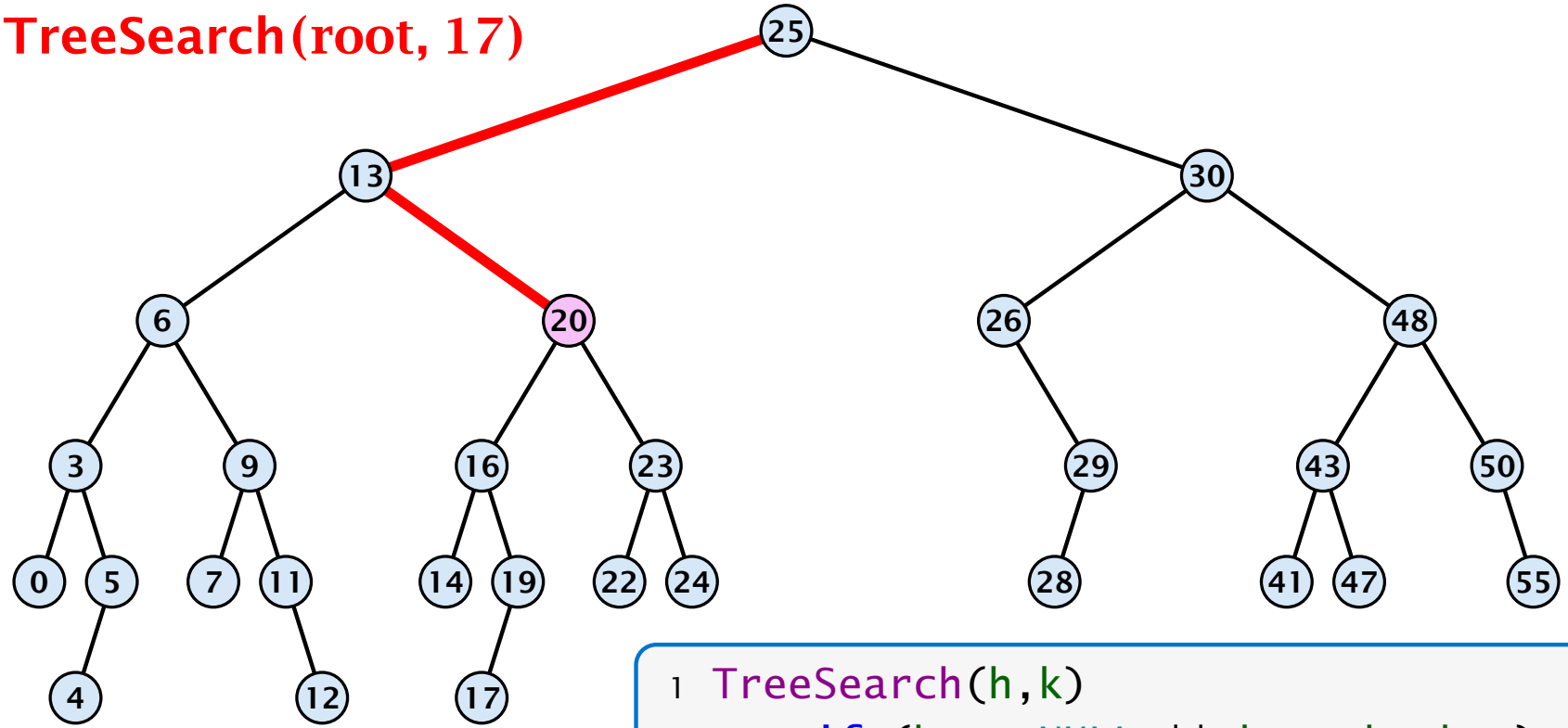
TreeSearch(root, 17)




```
1 TreeSearch(h,k)
2     if (h == NULL || k == h->key)
3         return h;
4     if (k < h->key)
5         return TreeSearch(h->left,k);
6     else
7         return TreeSearch(h->right,k);
```

Binäre Suchbäume: Suchen

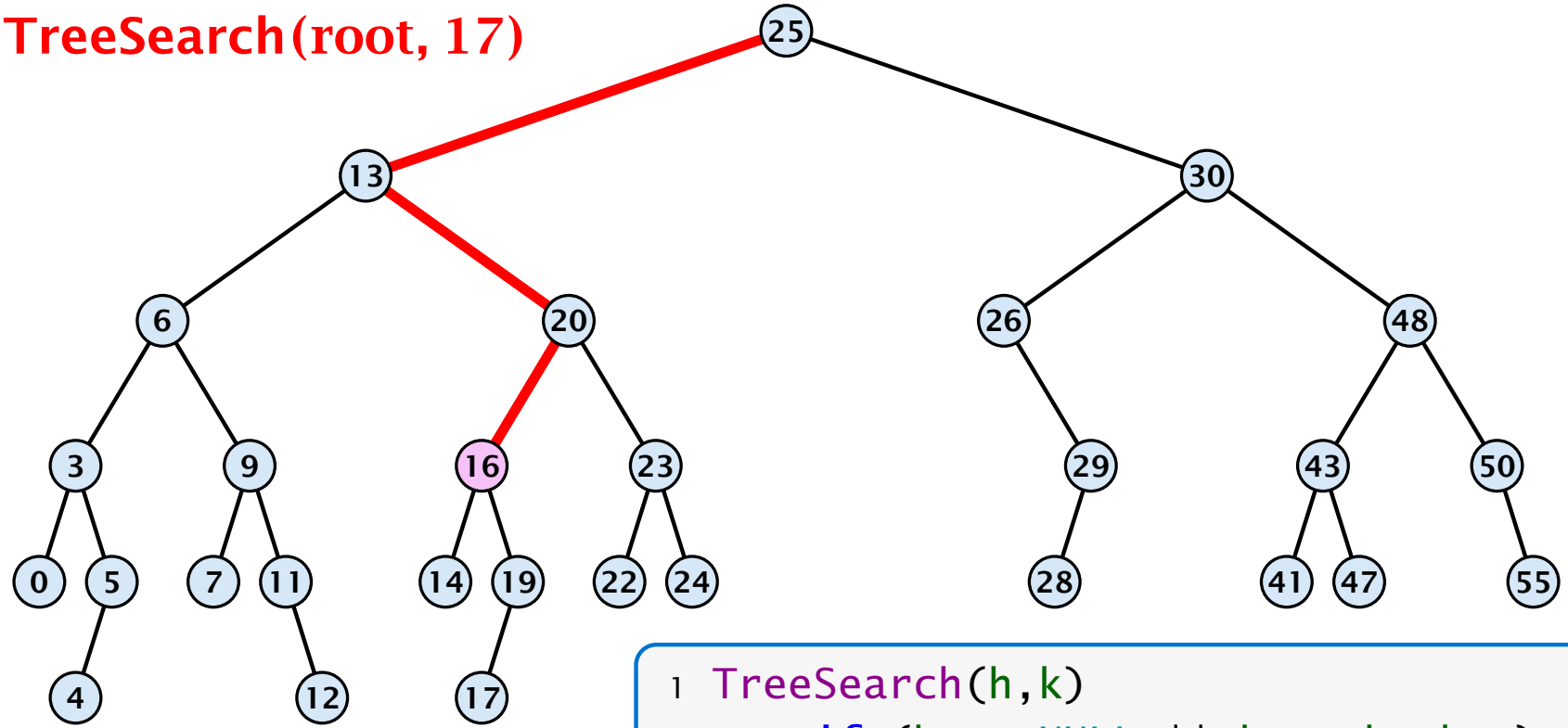
TreeSearch(root, 17)




```
1 TreeSearch(h,k)
2     if (h == NULL || k == h->key)
3         return h;
4     if (k < h->key)
5         return TreeSearch(h->left,k);
6     else
7         return TreeSearch(h->right,k);
```

Binäre Suchbäume: Suchen

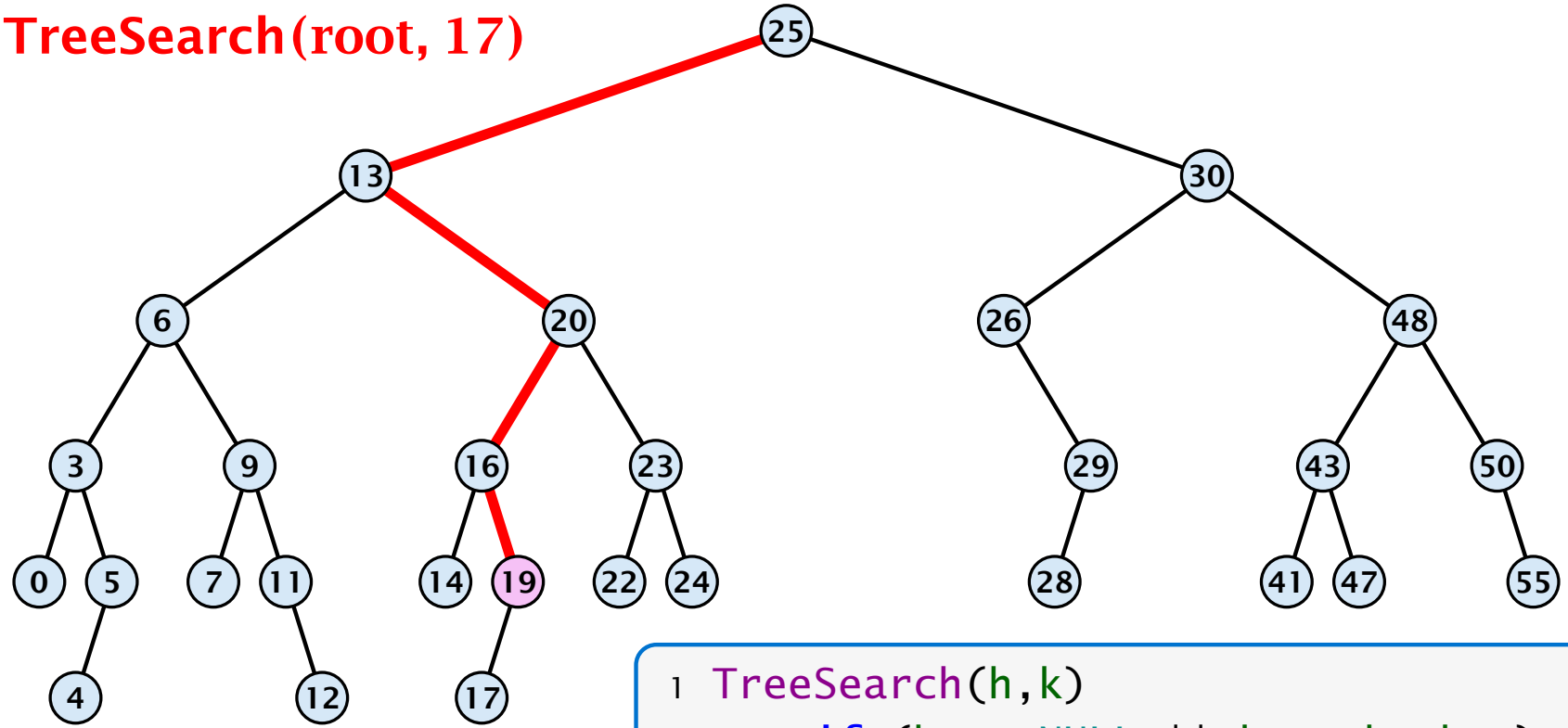
TreeSearch(root, 17)




```
1 TreeSearch(h,k)
2     if (h == NULL || k == h->key)
3         return h;
4     if (k < h->key)
5         return TreeSearch(h->left,k);
6     else
7         return TreeSearch(h->right,k);
```

Binäre Suchbäume: Suchen

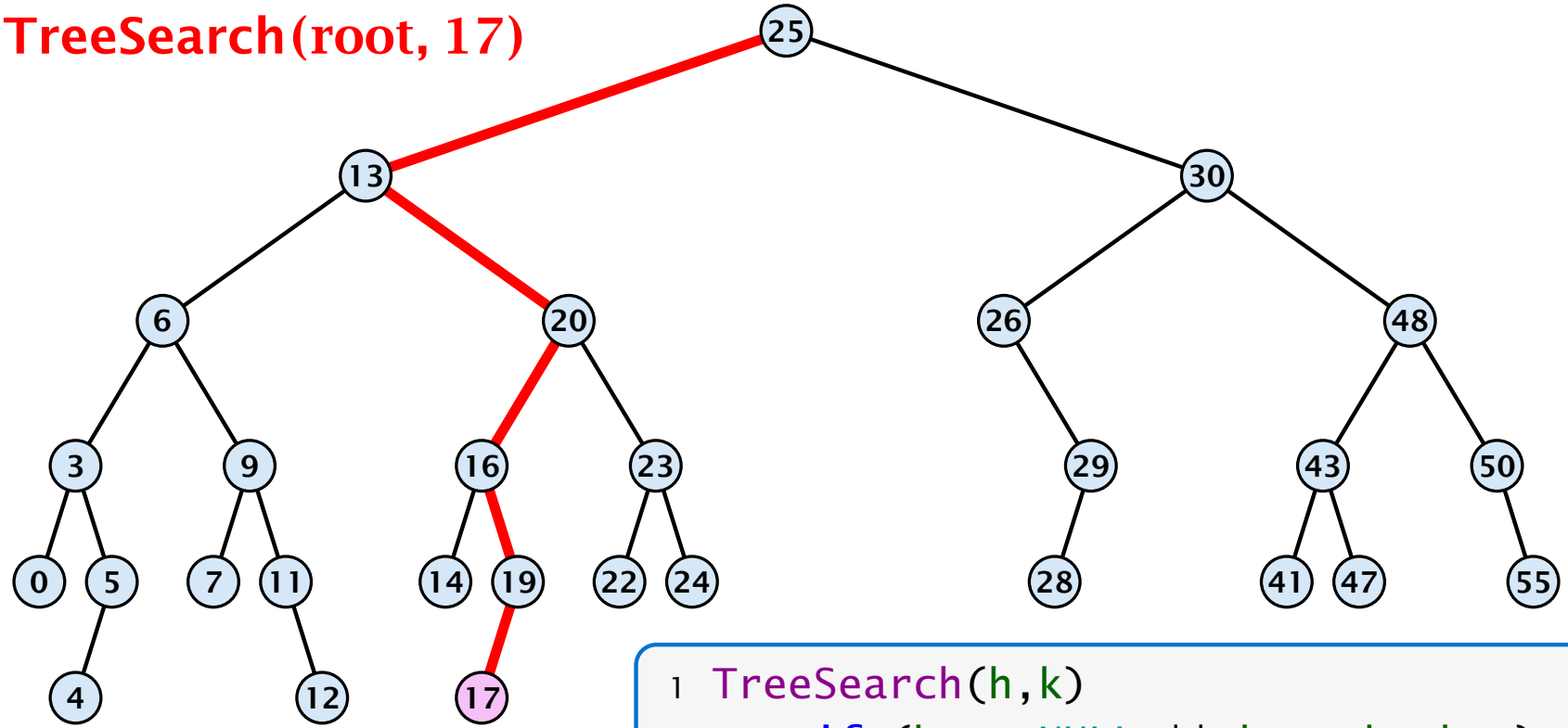
TreeSearch(root, 17)




```
1 TreeSearch(h,k)
2     if (h == NULL || k == h->key)
3         return h;
4     if (k < h->key)
5         return TreeSearch(h->left,k);
6     else
7         return TreeSearch(h->right,k);
```

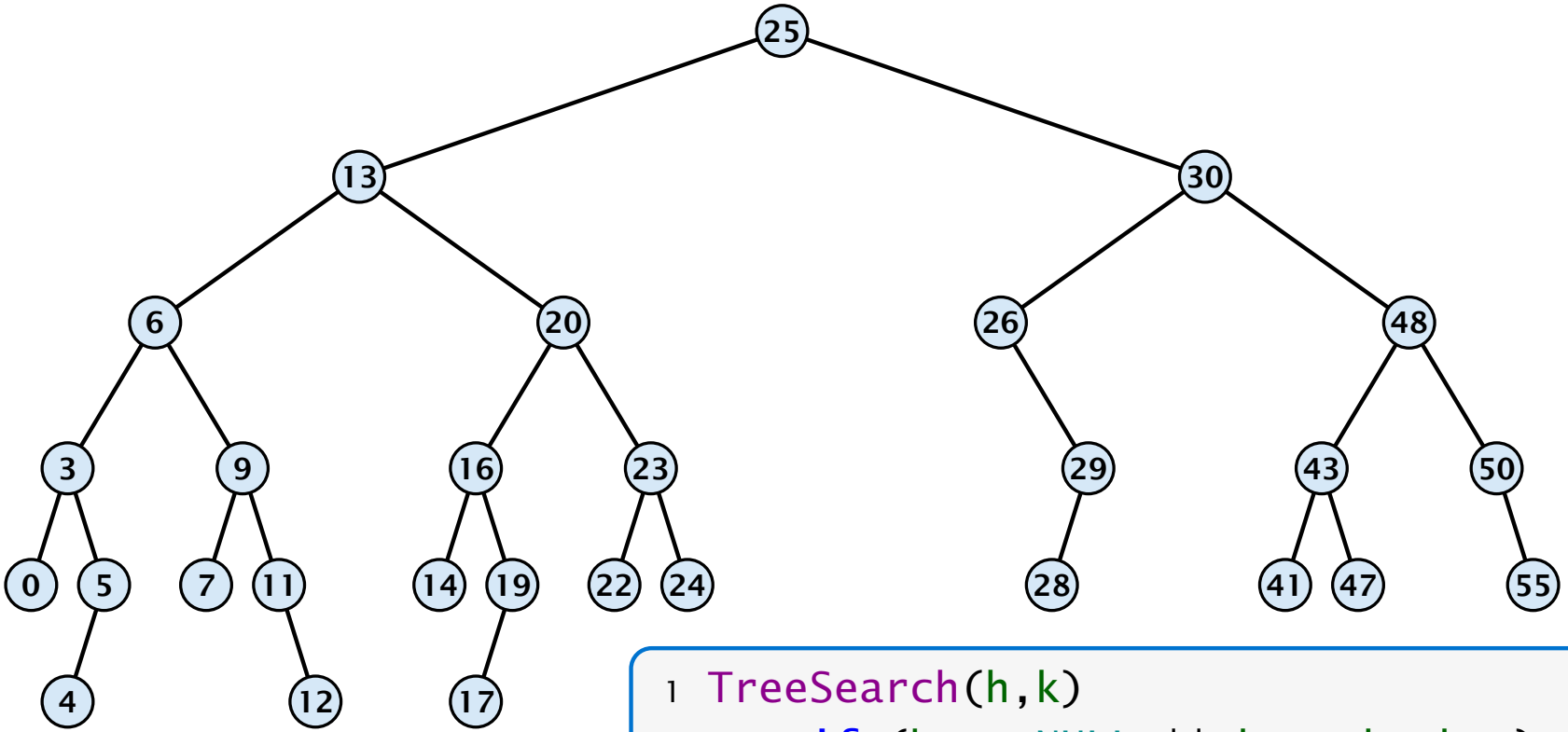
Binäre Suchbäume: Suchen

TreeSearch(root, 17)




```
1 TreeSearch(h,k)
2     if (h == NULL || k == h->key)
3         return h;
4     if (k < h->key)
5         return TreeSearch(h->left,k);
6     else
7         return TreeSearch(h->right,k);
```

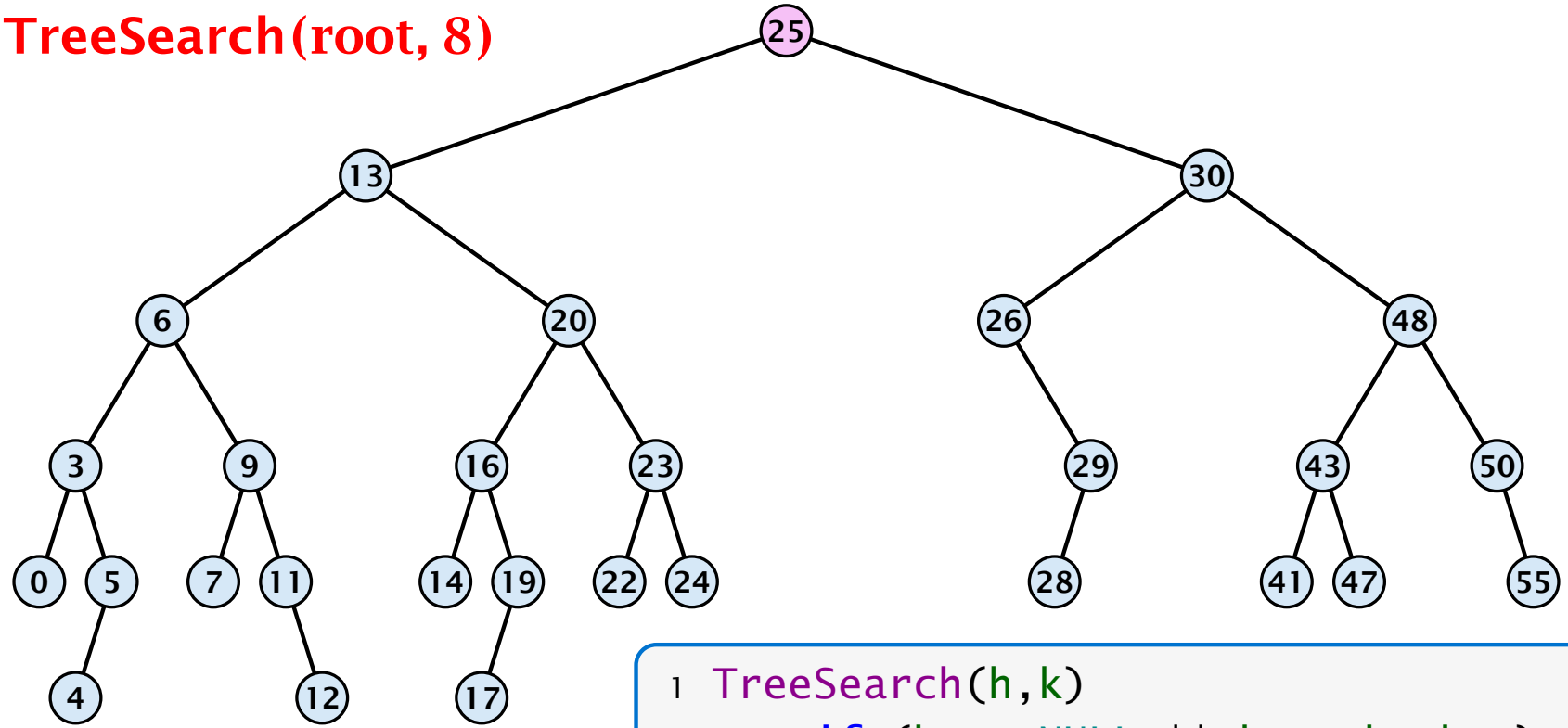
Binäre Suchbäume: Suchen




```
1 TreeSearch(h,k)
2     if (h == NULL || k == h->key)
3         return h;
4     if (k < h->key)
5         return TreeSearch(h->left,k);
6     else
7         return TreeSearch(h->right,k);
```

Binäre Suchbäume: Suchen

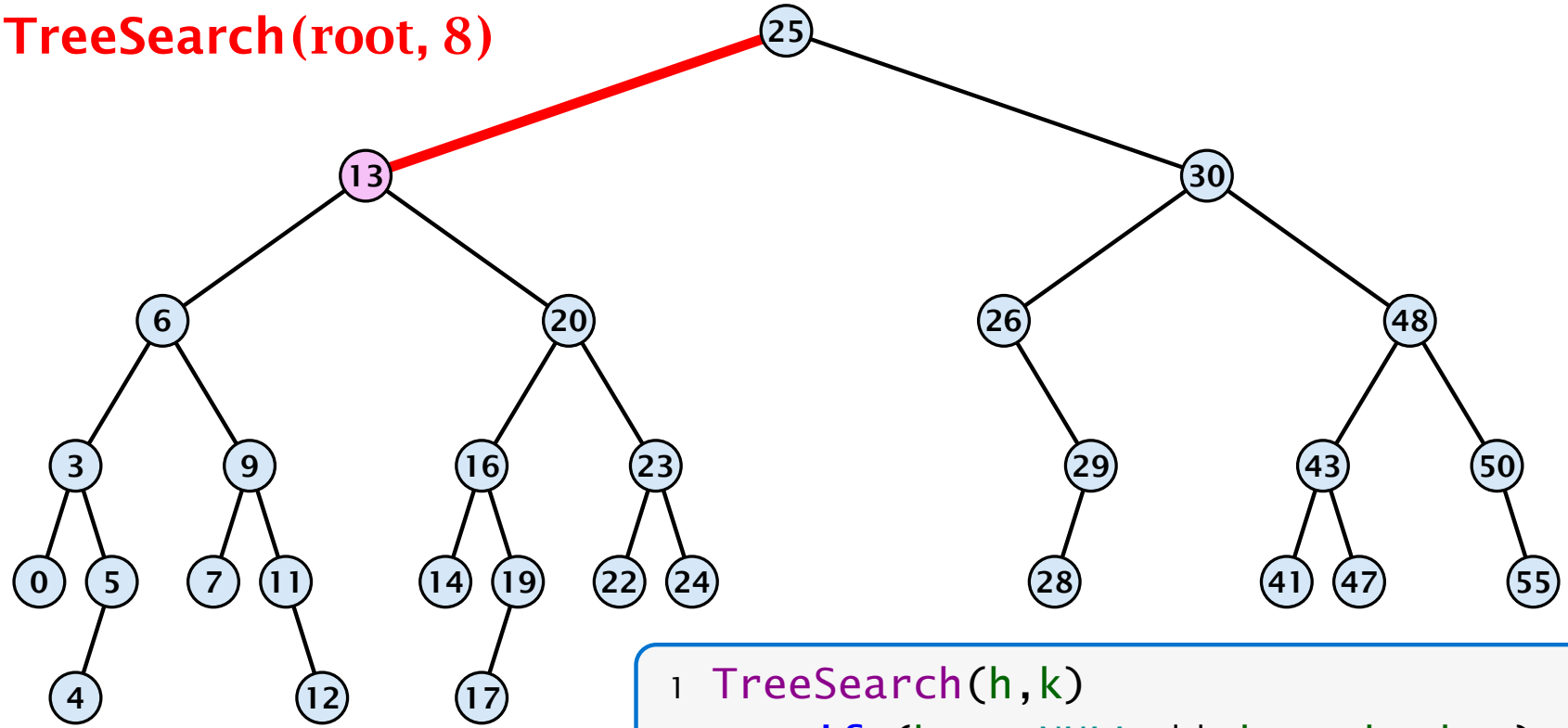
TreeSearch(root, 8)




```
1 TreeSearch(h,k)
2     if (h == NULL || k == h->key)
3         return h;
4     if (k < h->key)
5         return TreeSearch(h->left,k);
6     else
7         return TreeSearch(h->right,k);
```

Binäre Suchbäume: Suchen

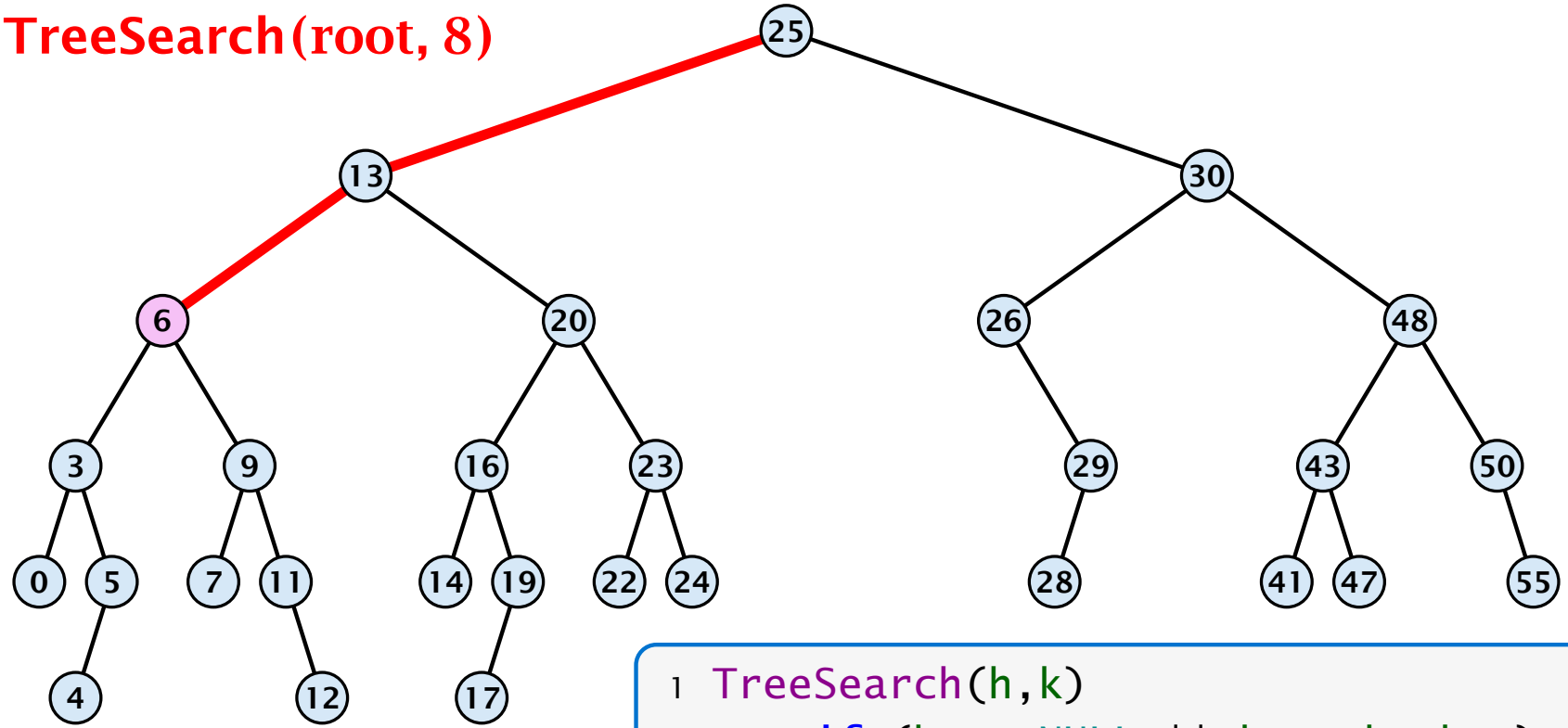
TreeSearch(root, 8)




```
1 TreeSearch(h,k)
2     if (h == NULL || k == h->key)
3         return h;
4     if (k < h->key)
5         return TreeSearch(h->left,k);
6     else
7         return TreeSearch(h->right,k);
```

Binäre Suchbäume: Suchen

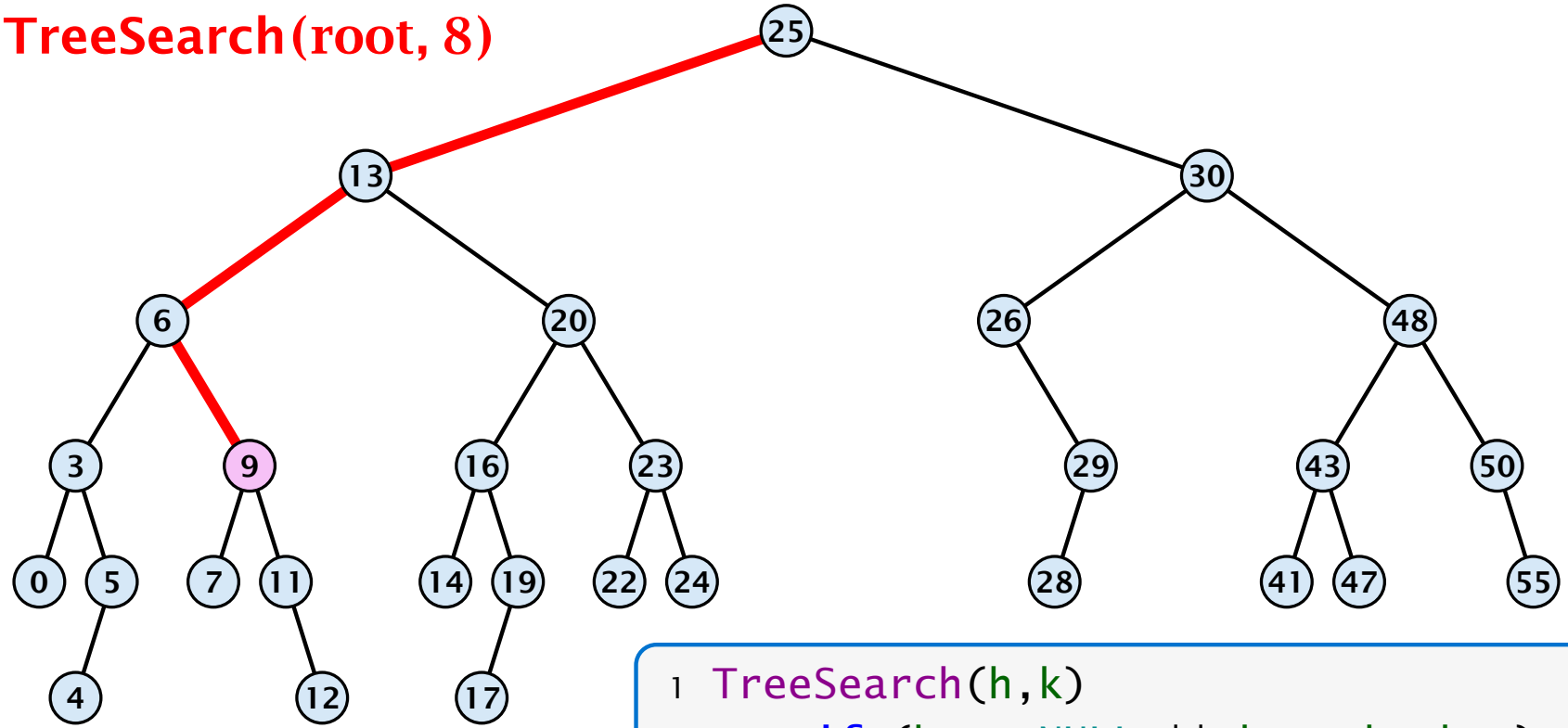
TreeSearch(root, 8)




```
1 TreeSearch(h,k)
2     if (h == NULL || k == h->key)
3         return h;
4     if (k < h->key)
5         return TreeSearch(h->left,k);
6     else
7         return TreeSearch(h->right,k);
```

Binäre Suchbäume: Suchen

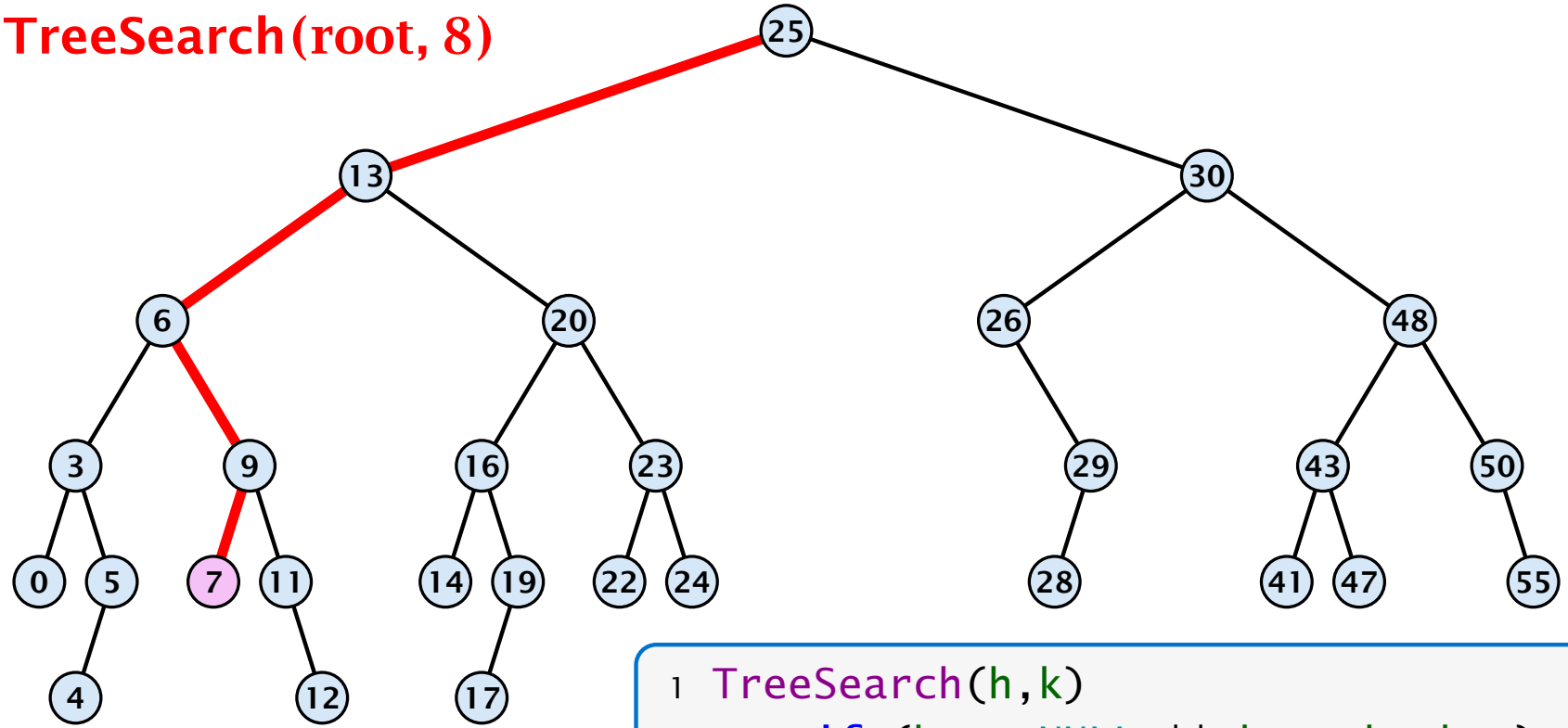
TreeSearch(root, 8)




```
1 TreeSearch(h,k)
2     if (h == NULL || k == h->key)
3         return h;
4     if (k < h->key)
5         return TreeSearch(h->left,k);
6     else
7         return TreeSearch(h->right,k);
```

Binäre Suchbäume: Suchen

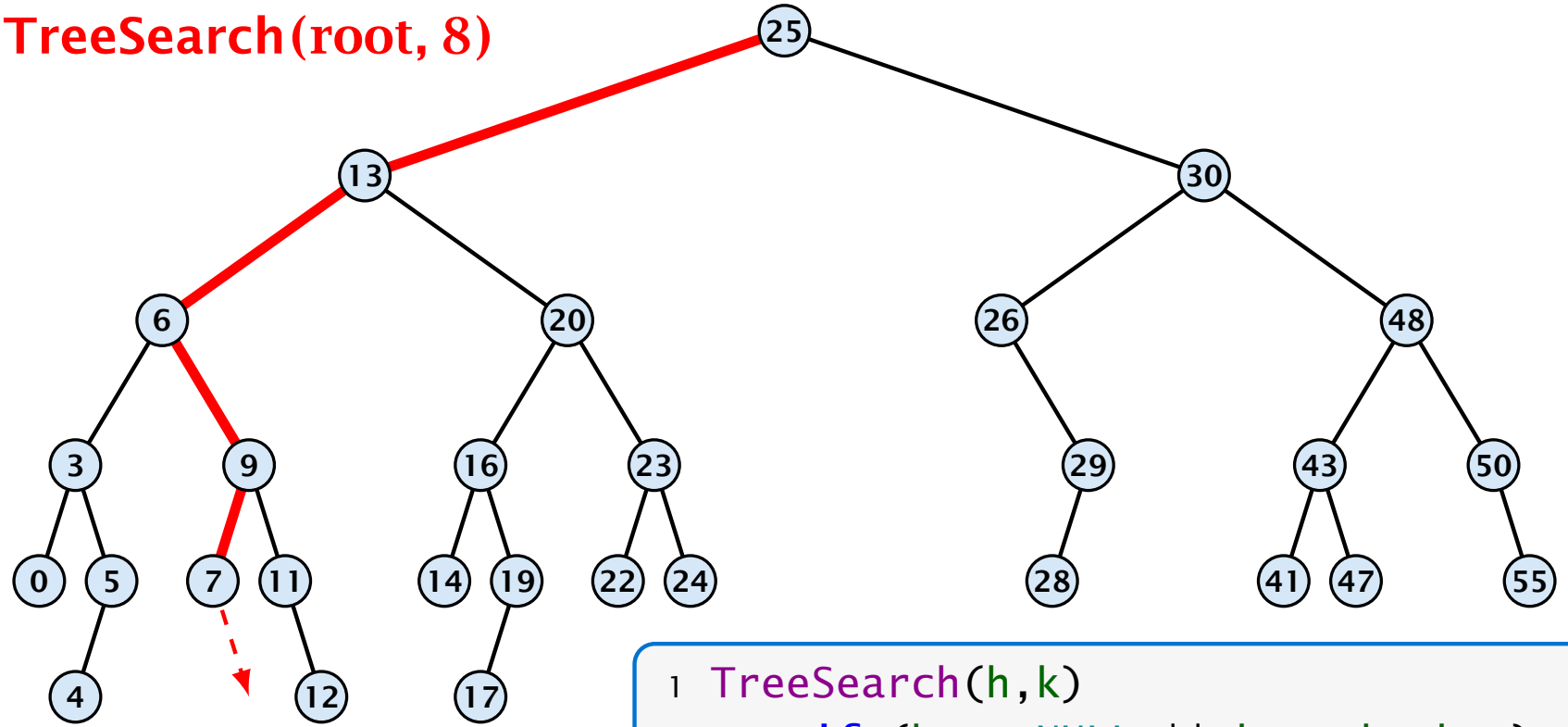
TreeSearch(root, 8)




```
1 TreeSearch(h,k)
2     if (h == NULL || k == h->key)
3         return h;
4     if (k < h->key)
5         return TreeSearch(h->left,k);
6     else
7         return TreeSearch(h->right,k);
```

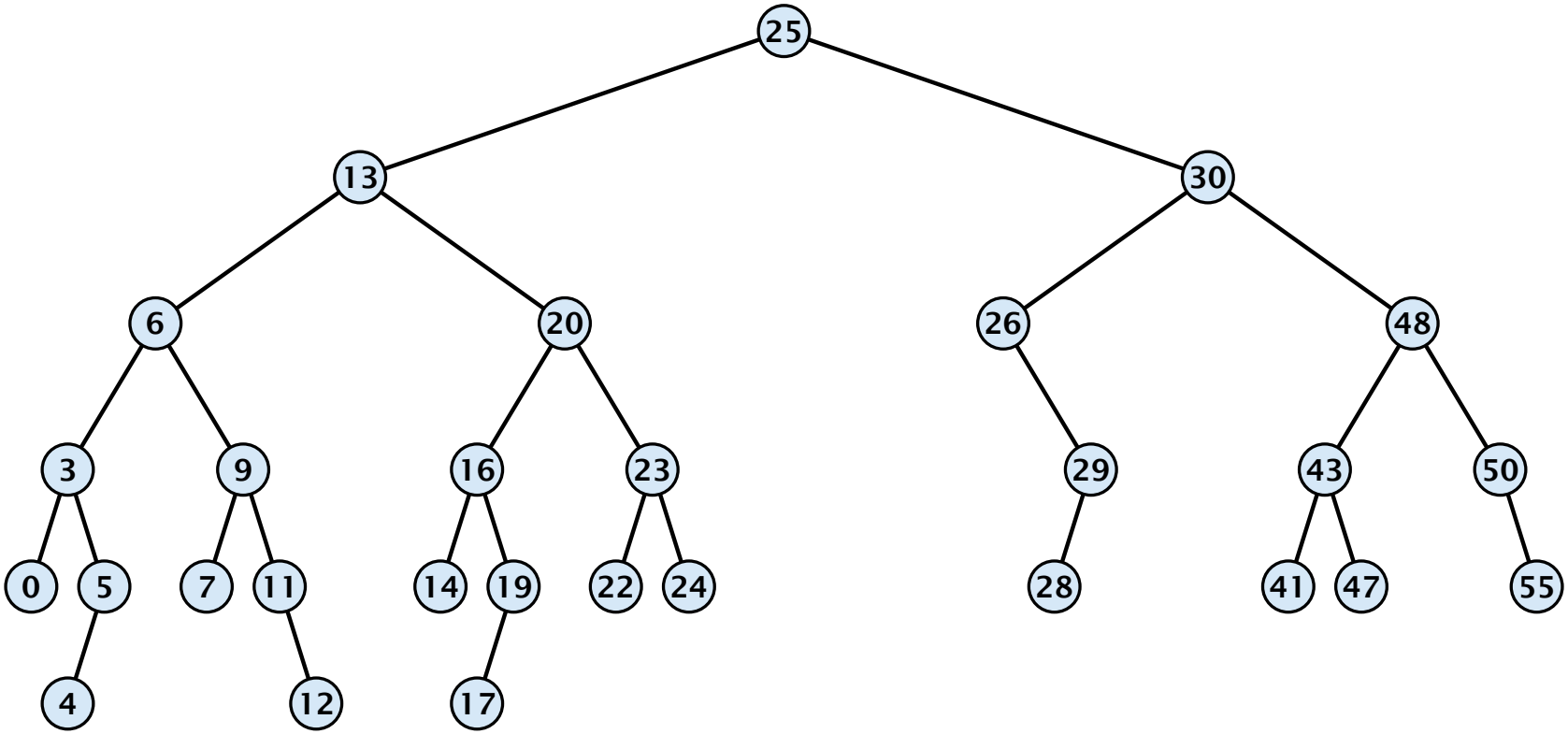
Binäre Suchbäume: Suchen

TreeSearch(root, 8)



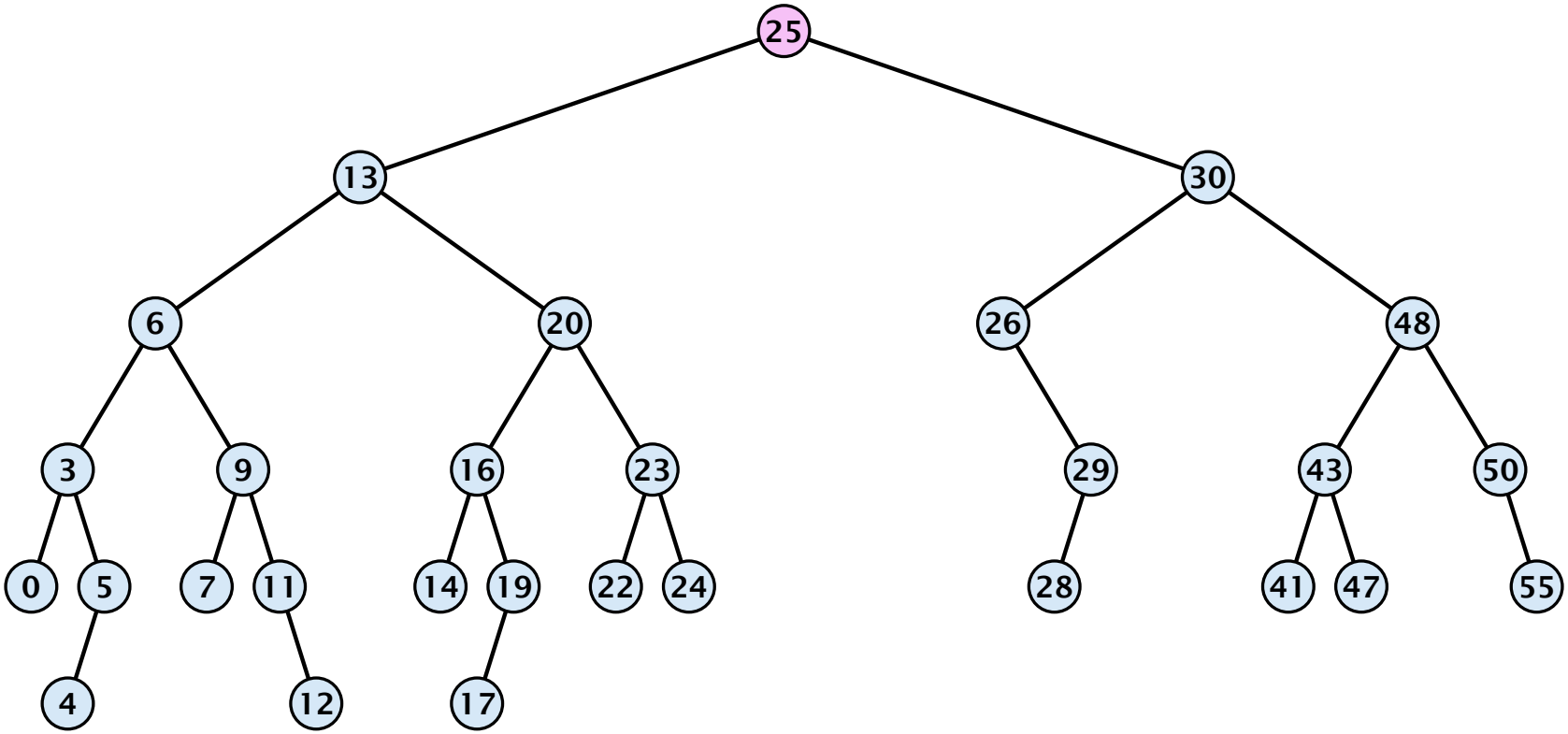

```
1 TreeSearch(h,k)
2     if (h == NULL || k == h->key)
3         return h;
4     if (k < h->key)
5         return TreeSearch(h->left,k);
6     else
7         return TreeSearch(h->right,k);
```

Binäre Suchbäume: Minimum



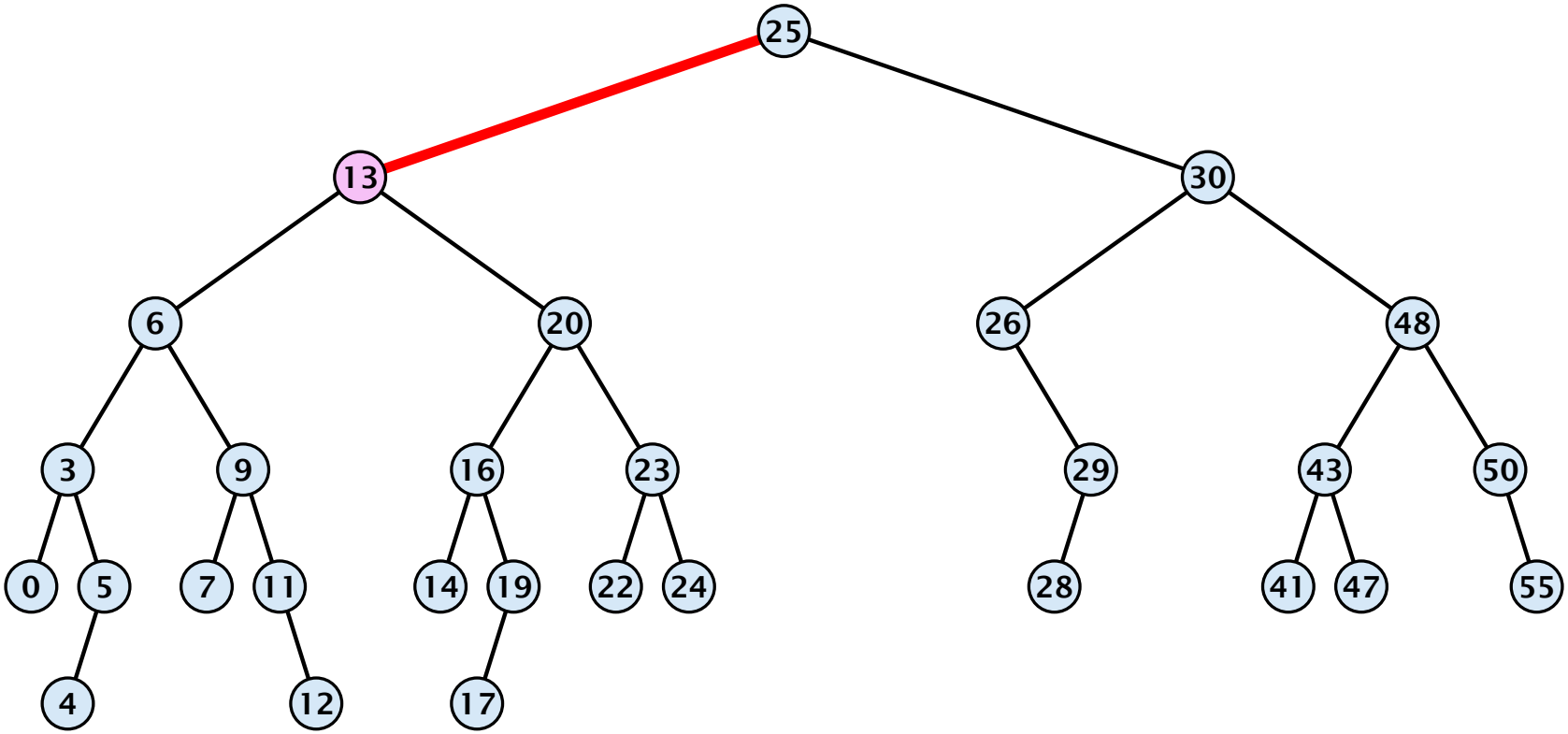

```
1 TreeMin(h)
2     if (h == NULL || h->left == NULL)
3         return h;
4     return TreeMin(h->left);
```

Binäre Suchbäume: Minimum



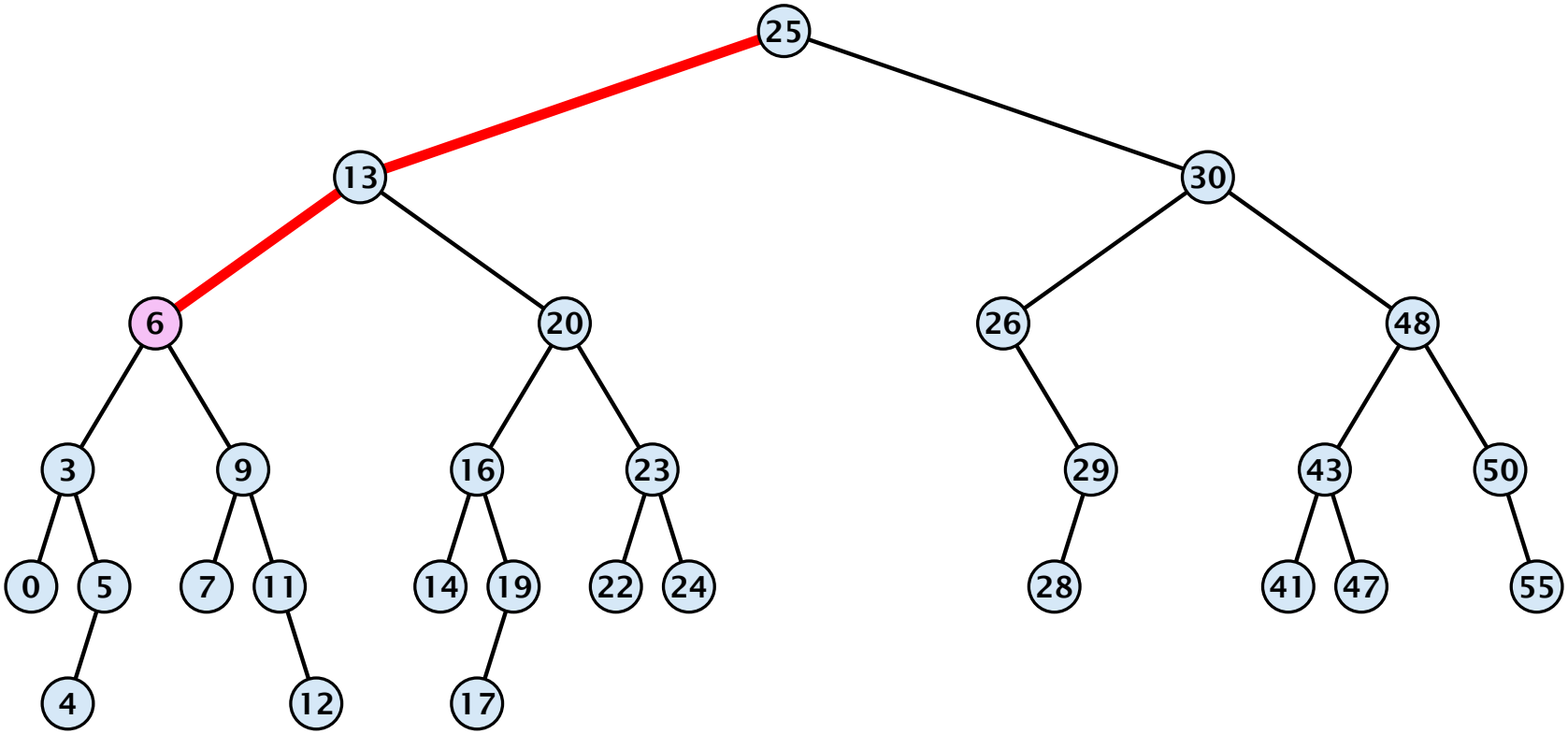

```
1 TreeMin(h)
2     if (h == NULL || h->left == NULL)
3         return h;
4     return TreeMin(h->left);
```

Binäre Suchbäume: Minimum



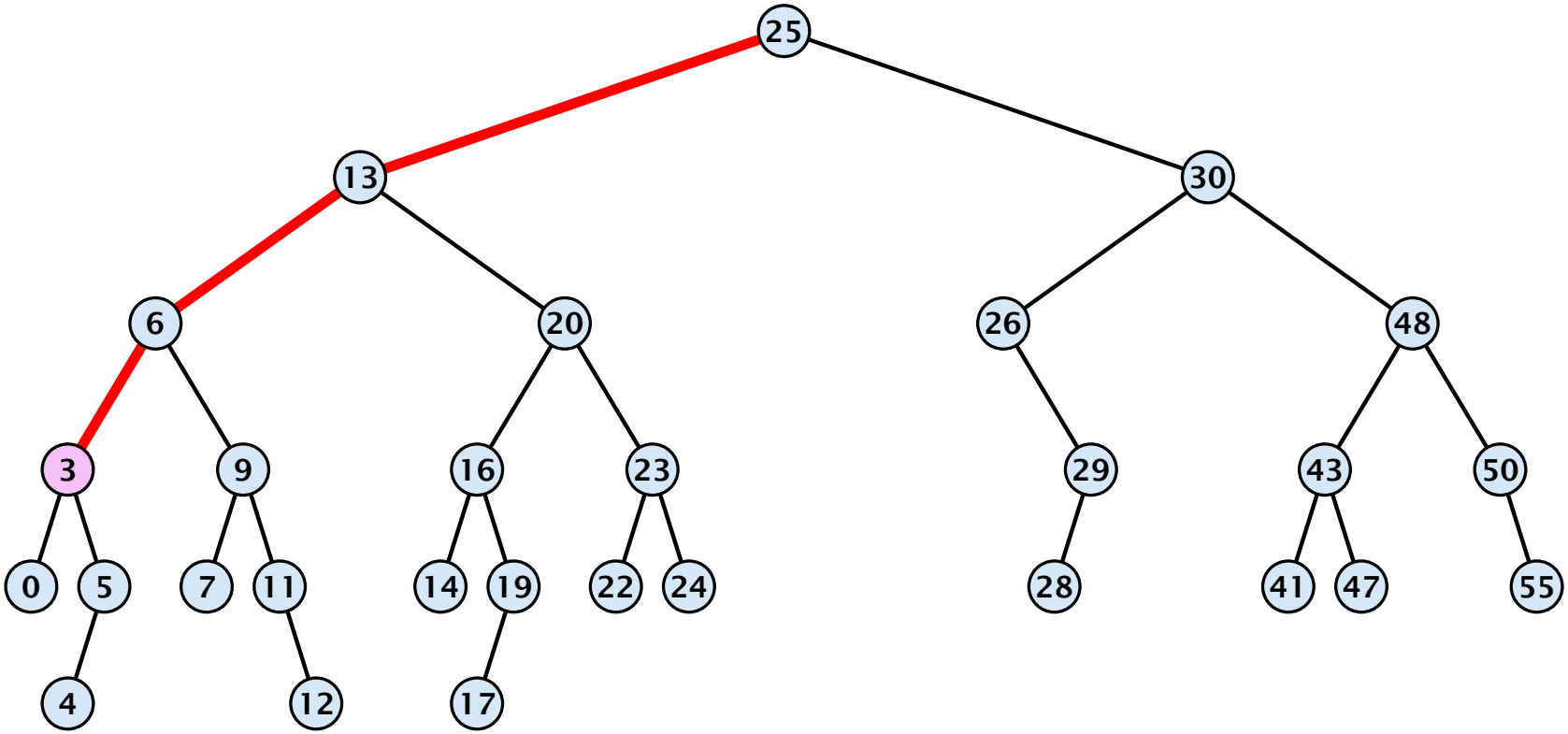

```
1 TreeMin(h)
2     if (h == NULL || h->left == NULL)
3         return h;
4     return TreeMin(h->left);
```

Binäre Suchbäume: Minimum



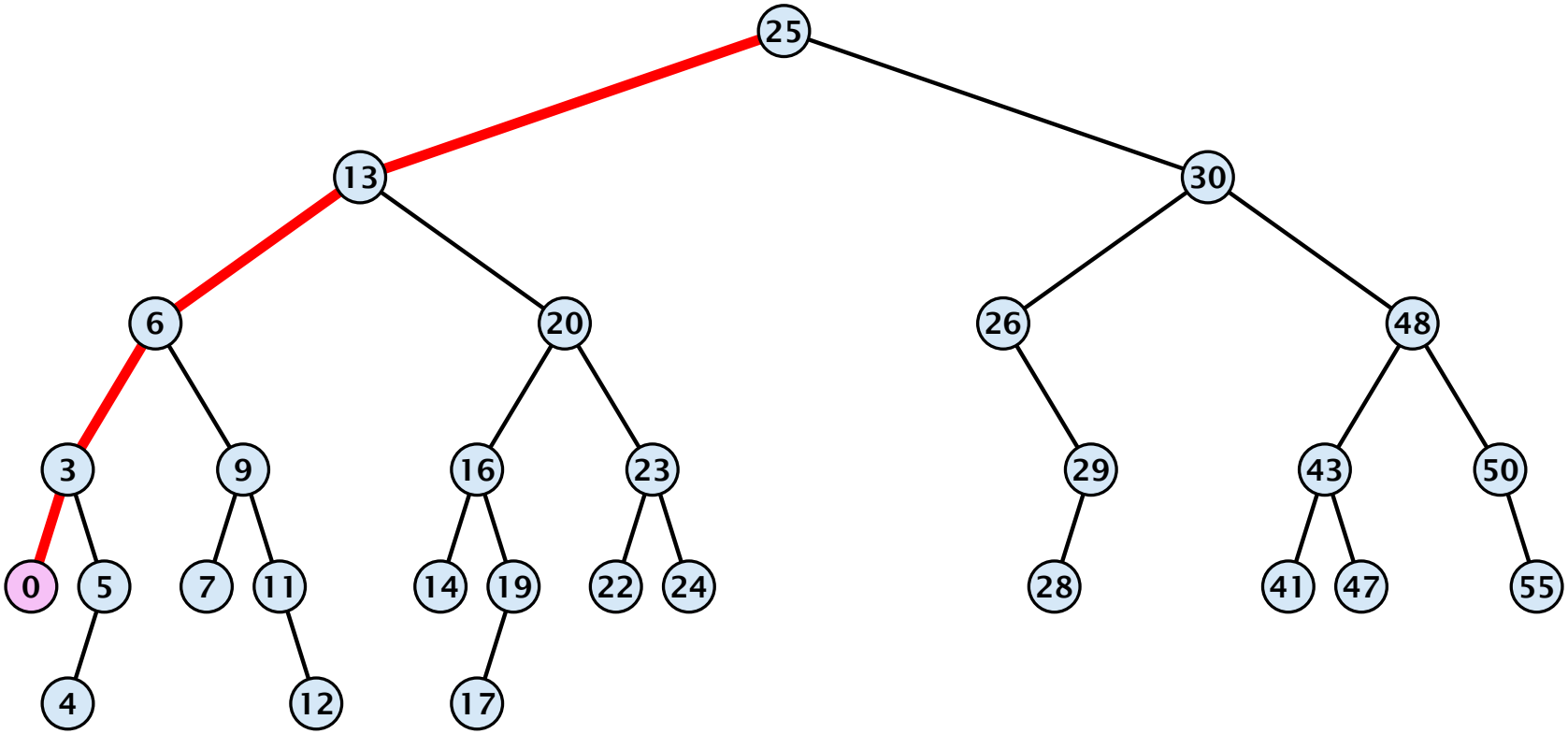

```
1 TreeMin(h)
2     if (h == NULL || h->left == NULL)
3         return h;
4     return TreeMin(h->left);
```

Binäre Suchbäume: Minimum



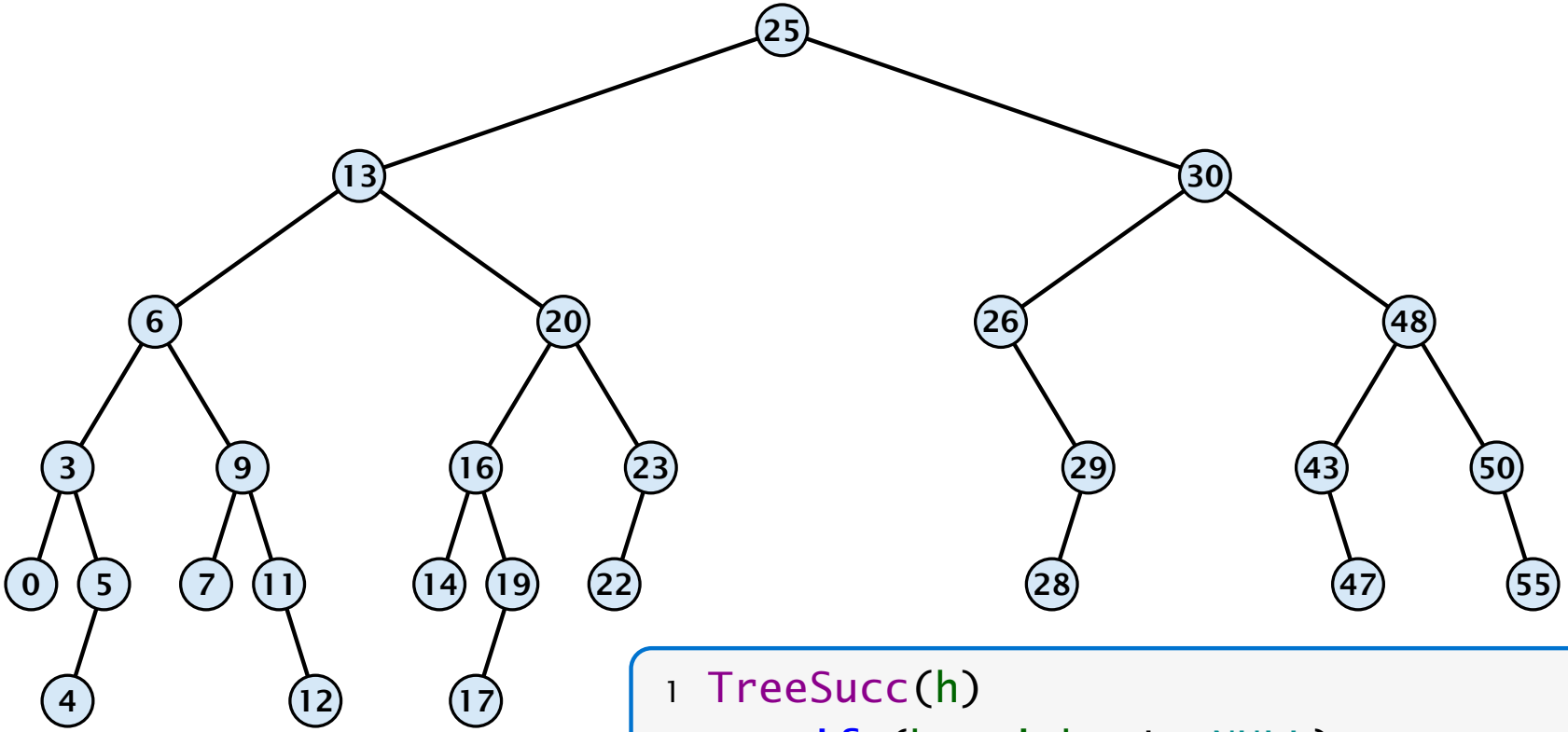

```
1 TreeMin(h)
2     if (h == NULL || h->left == NULL)
3         return h;
4     return TreeMin(h->left);
```

Binäre Suchbäume: Minimum



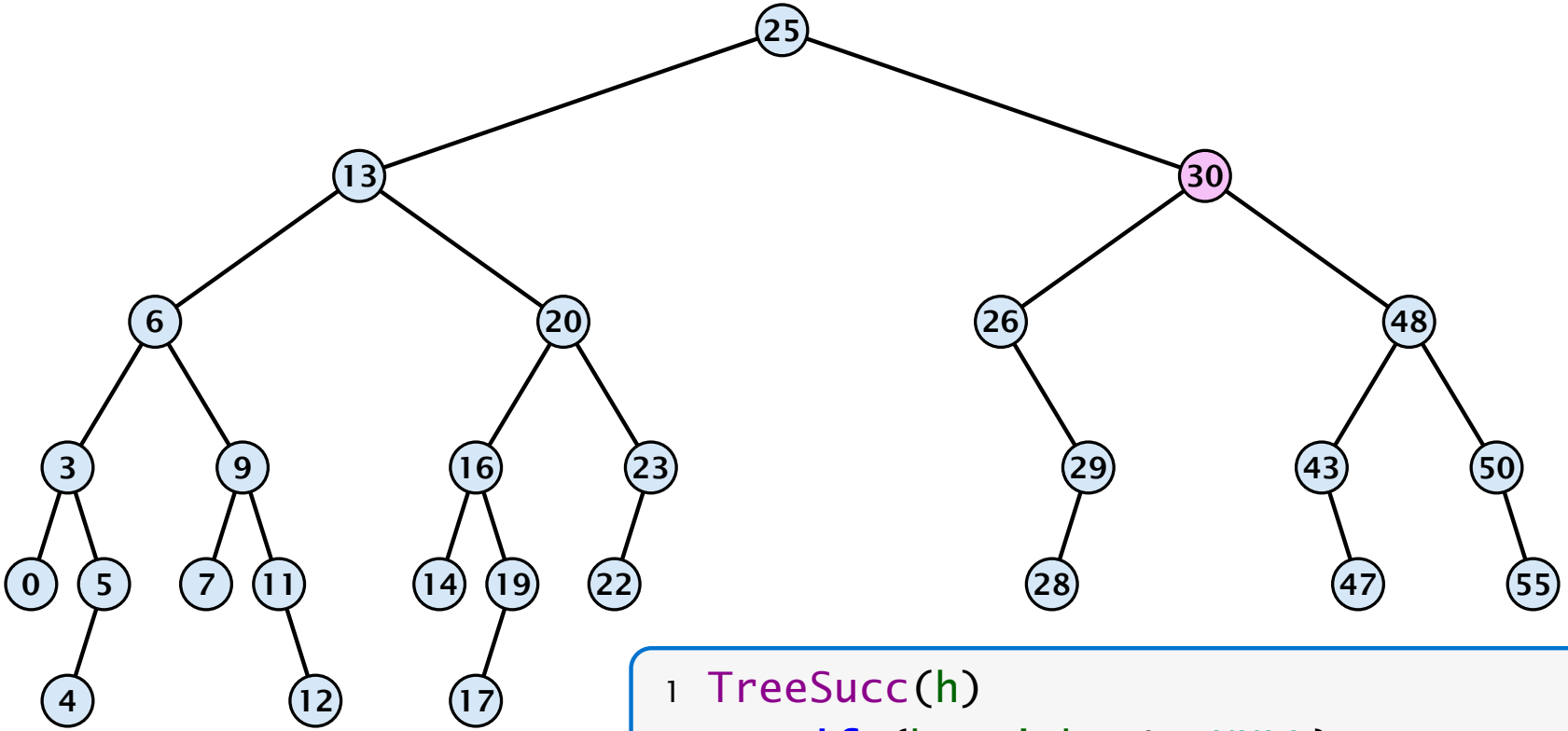

```
1 TreeMin(h)
2     if (h == NULL || h->left == NULL)
3         return h;
4     return TreeMin(h->left);
```

Binäre Suchbäume: Nachfolger



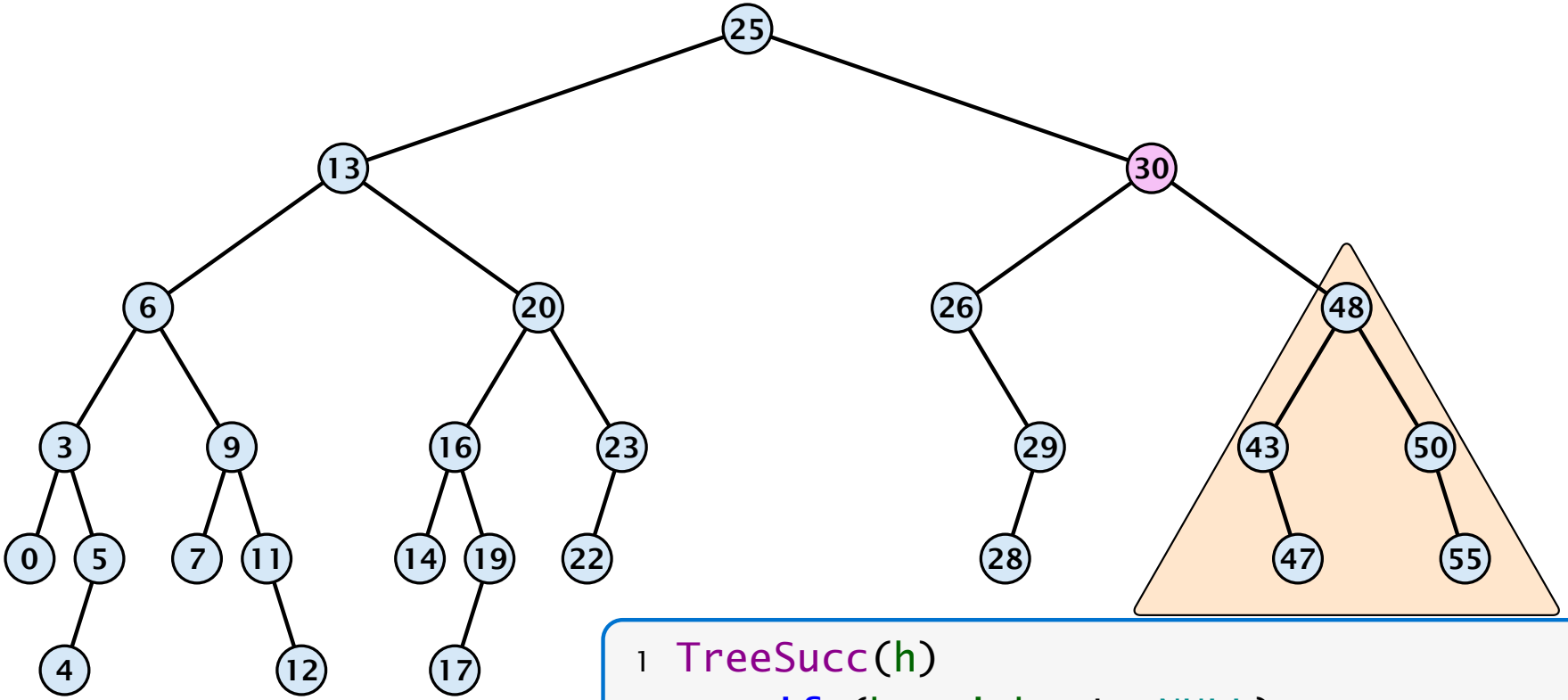

```
1 TreeSucc(h)
2     if (h->right != NULL)
3         return TreeMin(h->right);
4     y = h->parent;
5     while (y != NULL && h == y->right)
6         h = y; y = h->parent;
7     return y;
```

Binäre Suchbäume: Nachfolger



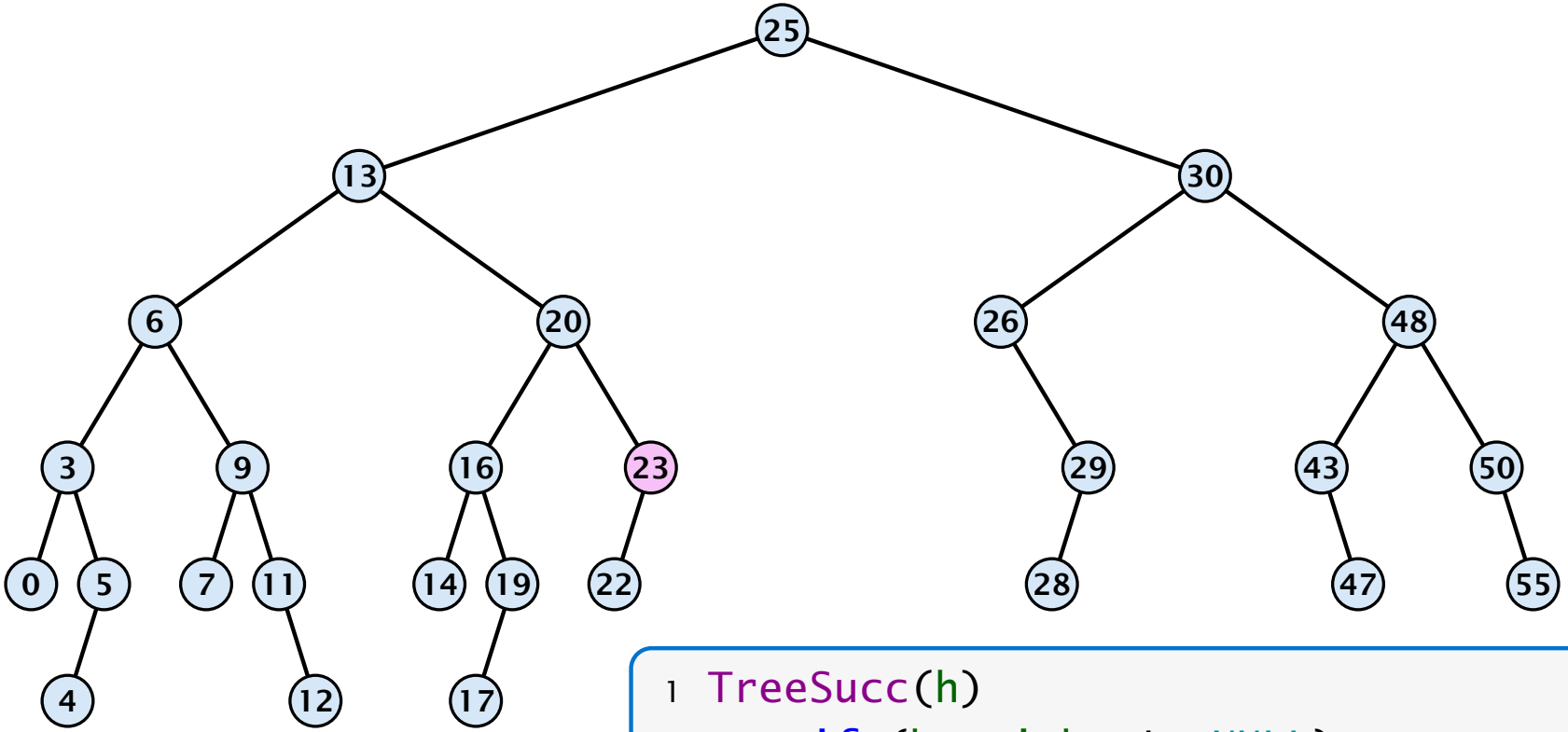

```
1 TreeSucc(h)
2     if (h->right != NULL)
3         return TreeMin(h->right);
4     y = h->parent;
5     while (y != NULL && h == y->right)
6         h = y; y = h->parent;
7     return y;
```

Binäre Suchbäume: Nachfolger



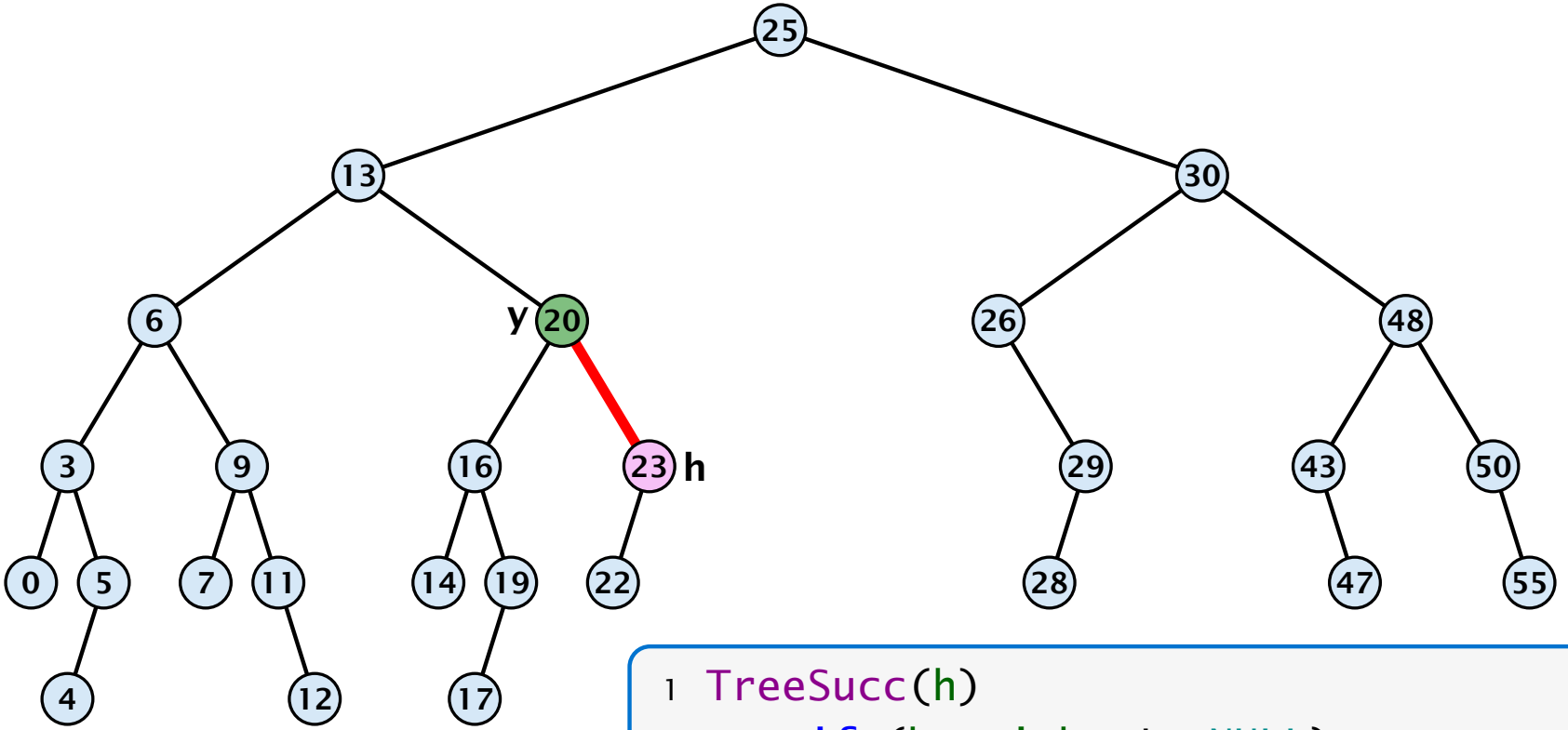

```
1  TreeSucc(h)
2      if (h->right != NULL)
3          return TreeMin(h->right);
4      y = h->parent;
5      while (y != NULL && h == y->right)
6          h = y; y = h->parent;
7      return y;
```

Binäre Suchbäume: Nachfolger



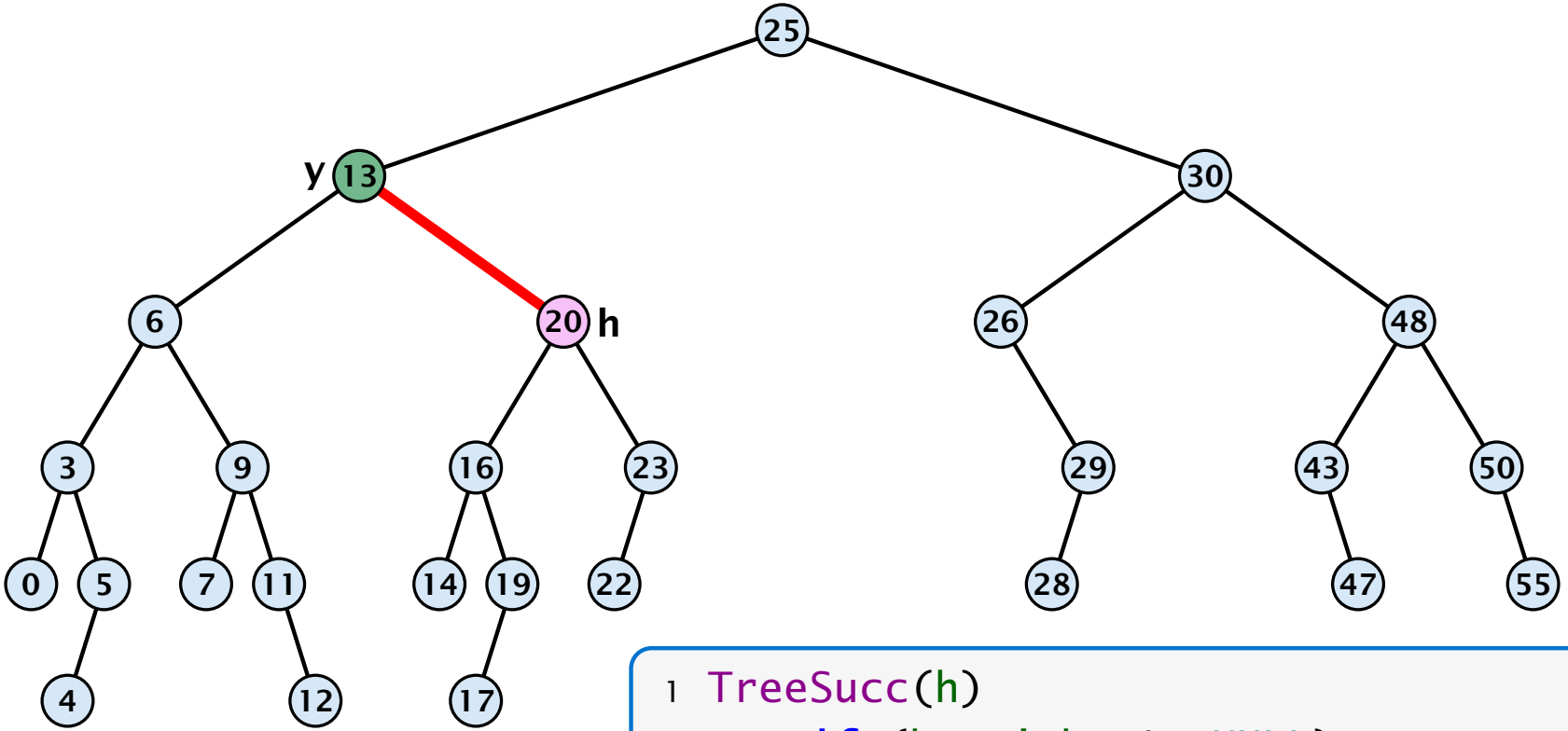

```
1 TreeSucc(h)
2     if (h->right != NULL)
3         return TreeMin(h->right);
4     y = h->parent;
5     while (y != NULL && h == y->right)
6         h = y; y = h->parent;
7     return y;
```

Binäre Suchbäume: Nachfolger



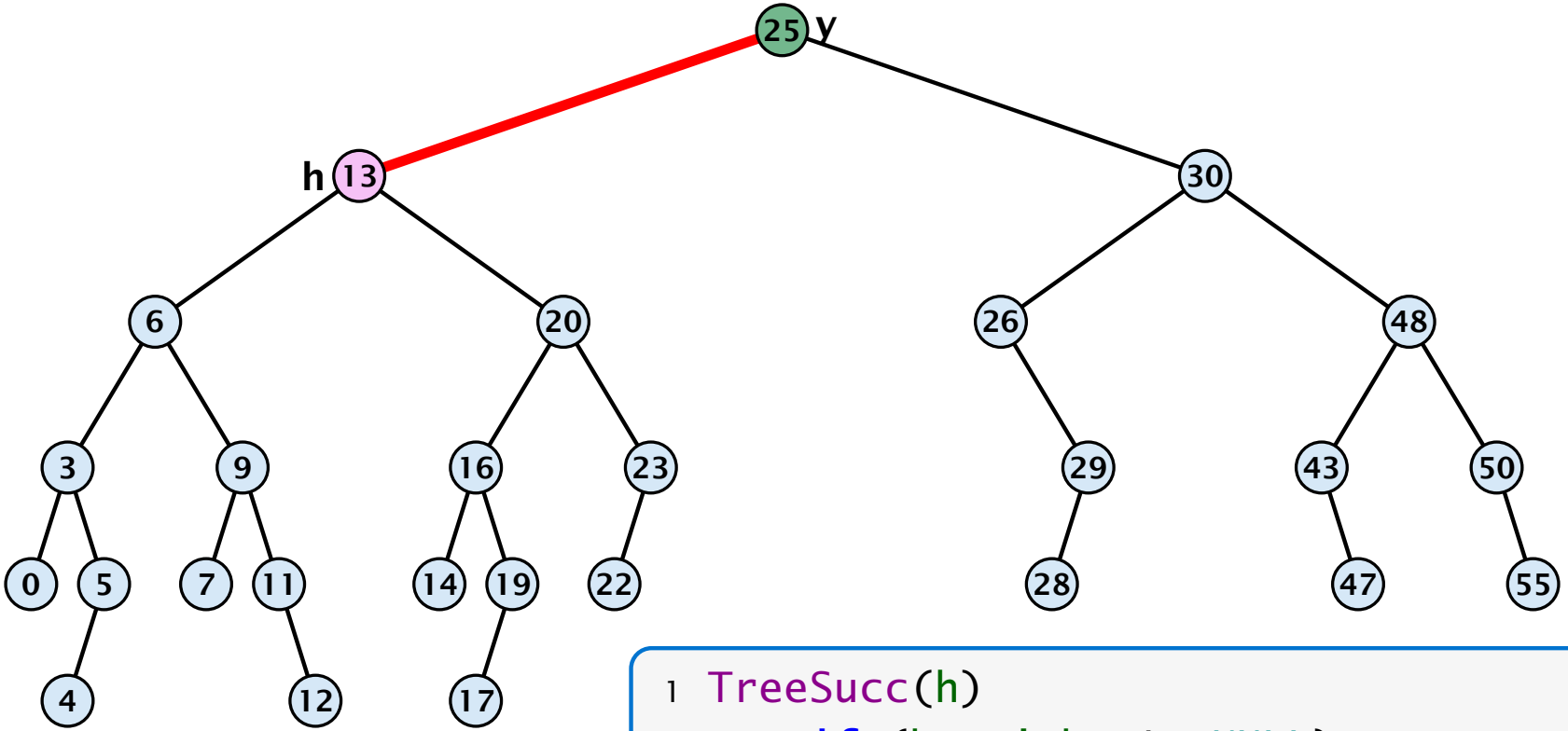

```
1 TreeSucc(h)
2     if (h->right != NULL)
3         return TreeMin(h->right);
4     y = h->parent;
5     while (y != NULL && h == y->right)
6         h = y; y = h->parent;
7     return y;
```

Binäre Suchbäume: Nachfolger



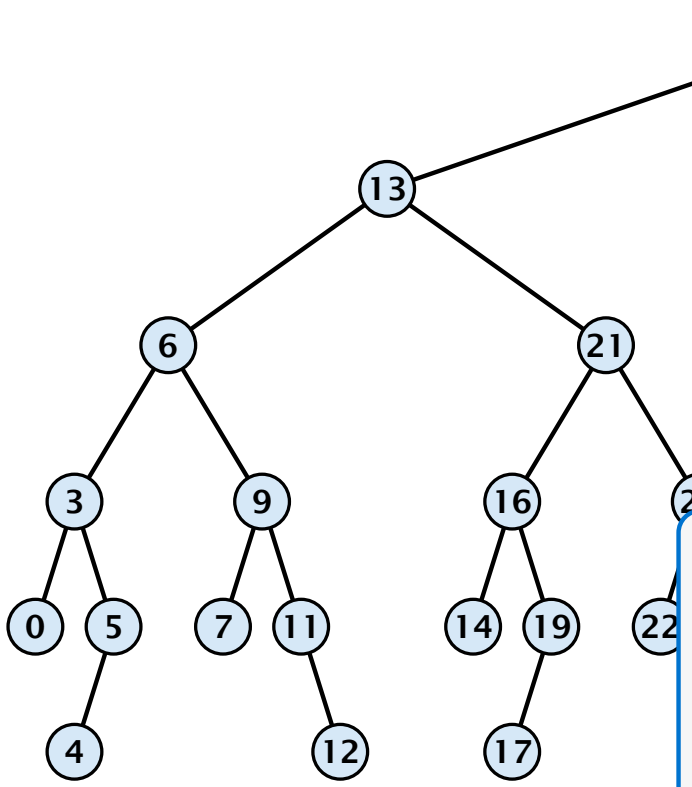

```
1 TreeSucc(h)
2     if (h->right != NULL)
3         return TreeMin(h->right);
4     y = h->parent;
5     while (y != NULL && h == y->right)
6         h = y; y = h->parent;
7     return y;
```

Binäre Suchbäume: Nachfolger




```
1  TreeSucc(h)
2      if (h->right != NULL)
3          return TreeMin(h->right);
4      y = h->parent;
5      while (y != NULL && h == y->right)
6          h = y; y = h->parent;
7      return y;
```

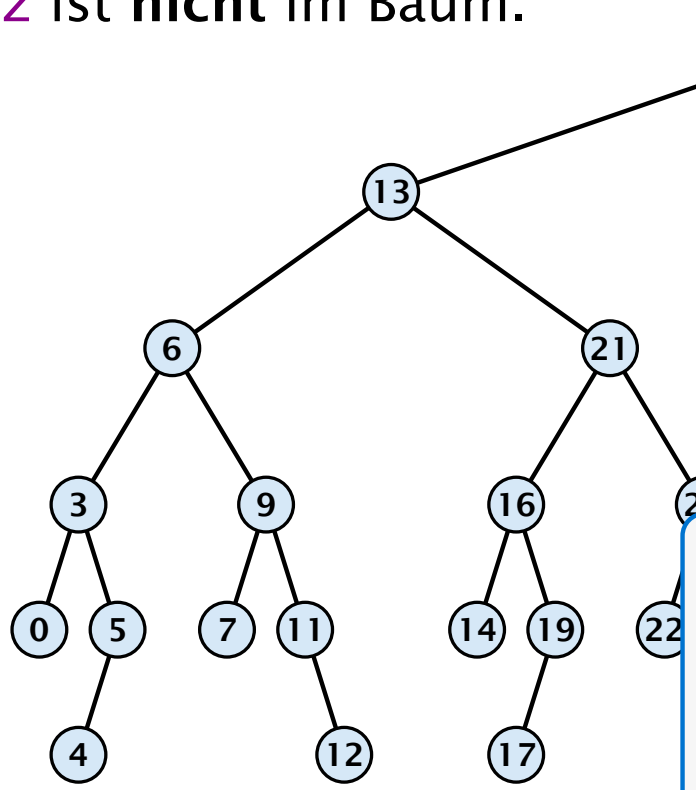
Binäre Suchbäume: Einfügen




```
TreeInsert(h,z)
    if (z == NULL)
        T->root = z; z->parent = NULL;
        return;
    if (h->key > z->key)
        if (h->left == NULL)
            h->left = z; z->parent = h;
        else TreeInsert(h->left,z);
    else
        if (h->right == NULL)
            h->right = z; z->parent = h;
        else TreeInsert(h->right,z);
```

Binäre Suchbäume: Einfügen

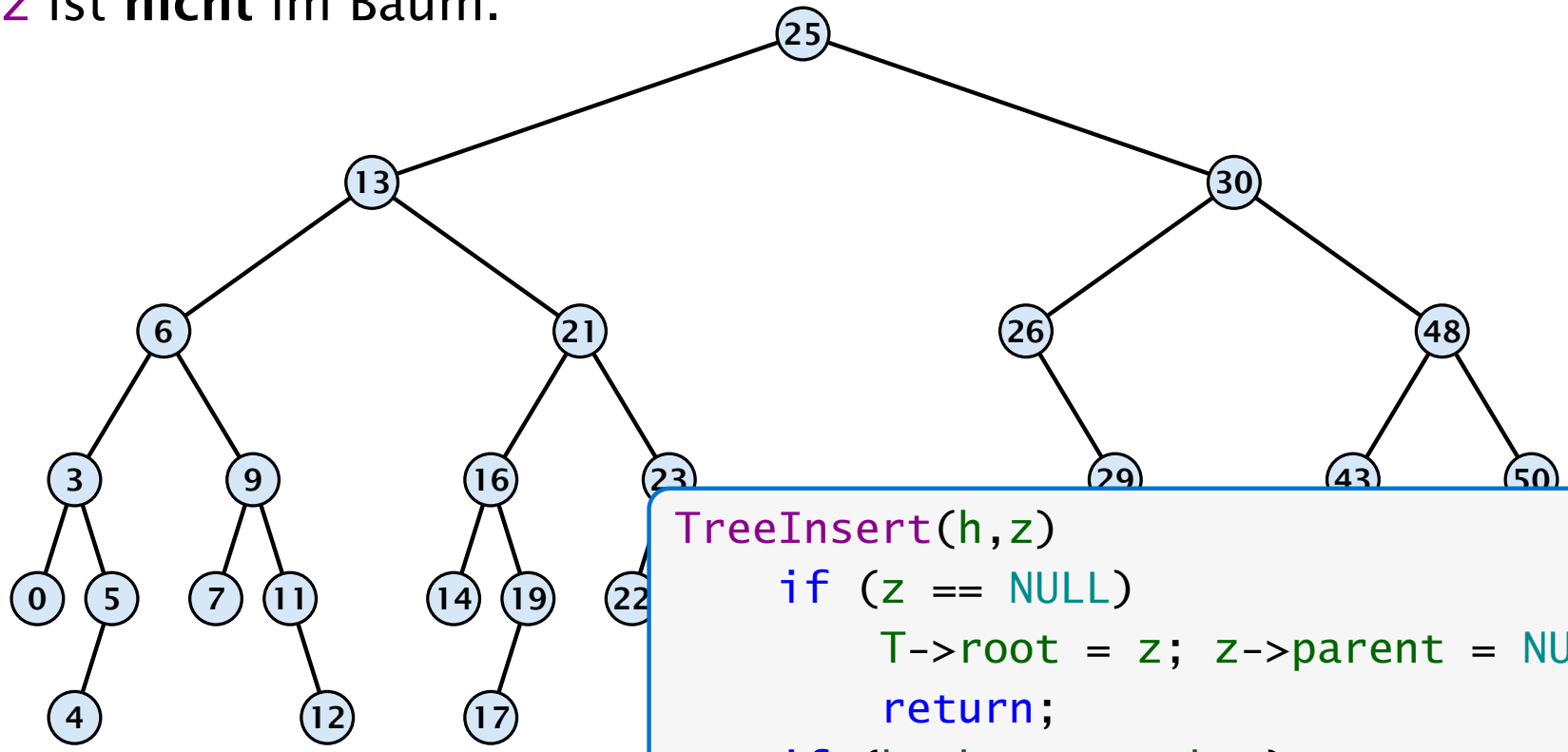
z ist **nicht** im Baum.




```
TreeInsert(h,z)
    if (z == NULL)
        T->root = z; z->parent = NULL;
        return;
    if (h->key > z->key)
        if (h->left == NULL)
            h->left = z; z->parent = h;
        else TreeInsert(h->left,z);
    else
        if (h->right == NULL)
            h->right = z; z->parent = h;
        else TreeInsert(h->right,z);
```

Binäre Suchbäume: Einfügen

z ist **nicht** im Baum.



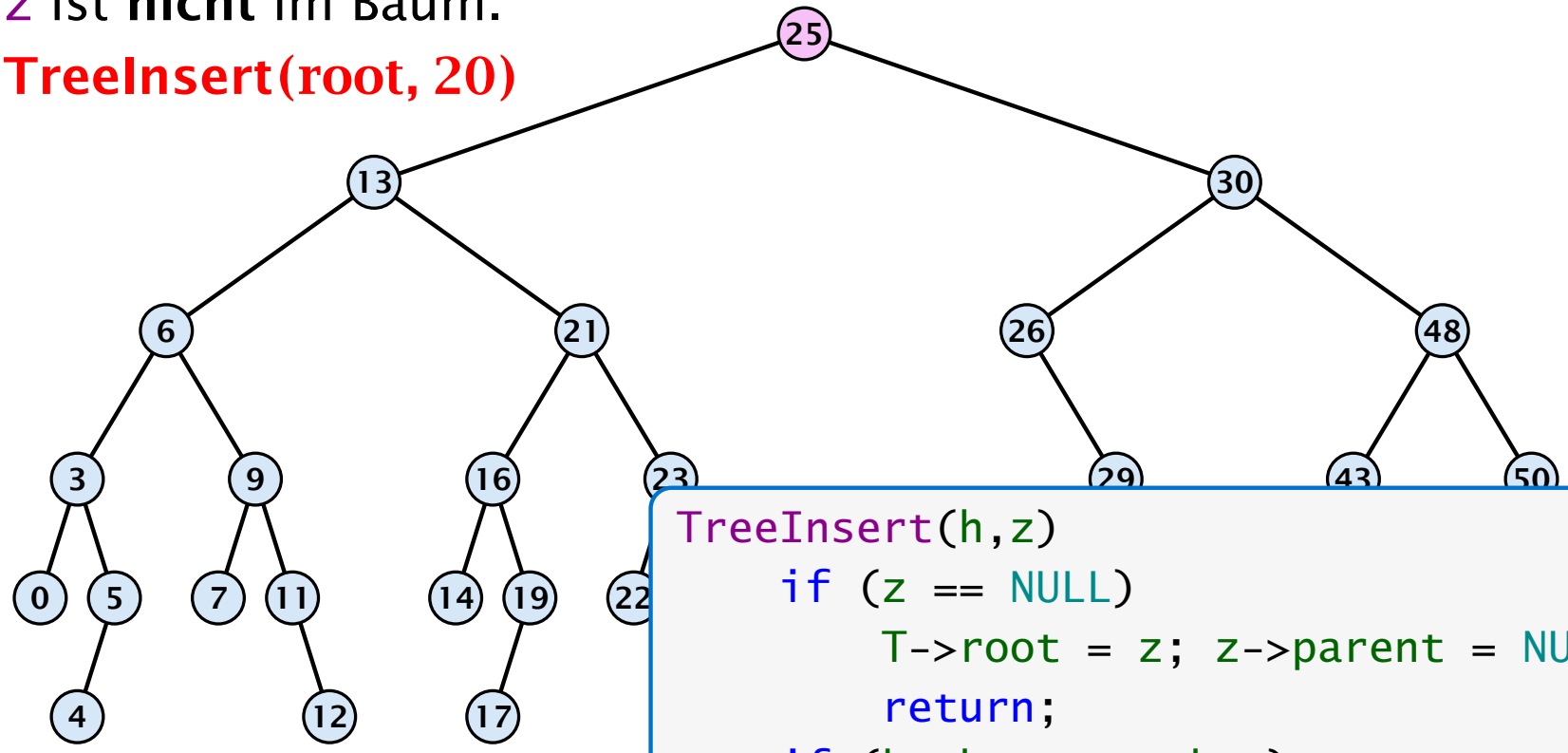
Suche nach z. Suche endet an NULL-Zeiger. Hier wird z eingefügt.

```
TreeInsert(h,z)
    if (z == NULL)
        T->root = z; z->parent = NULL;
        return;
    if (h->key > z->key)
        if (h->left == NULL)
            h->left = z; z->parent = h;
        else TreeInsert(h->left,z);
    else
        if (h->right == NULL)
            h->right = z; z->parent = h;
        else TreeInsert(h->right,z);
```

Binäre Suchbäume: Einfügen

z ist **nicht** im Baum.

TreeInsert(root, 20)



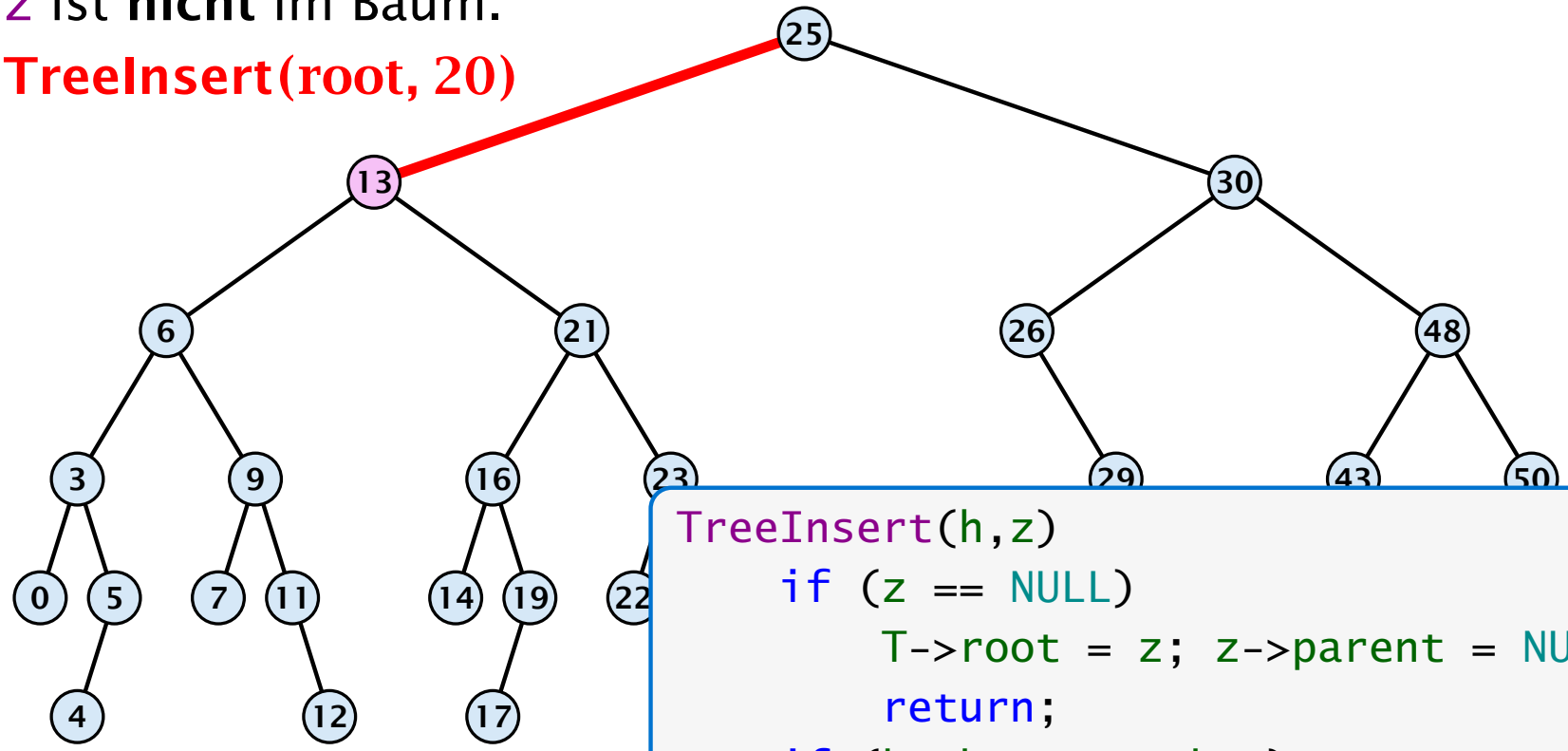
Suche nach z. Suche endet an NULL-Zeiger. Hier wird z eingefügt.

```
TreeInsert(h,z)
    if (z == NULL)
        T->root = z; z->parent = NULL;
        return;
    if (h->key > z->key)
        if (h->left == NULL)
            h->left = z; z->parent = h;
        else TreeInsert(h->left,z);
    else
        if (h->right == NULL)
            h->right = z; z->parent = h;
        else TreeInsert(h->right,z);
```

Binäre Suchbäume: Einfügen

z ist **nicht** im Baum.

TreeInsert(root, 20)



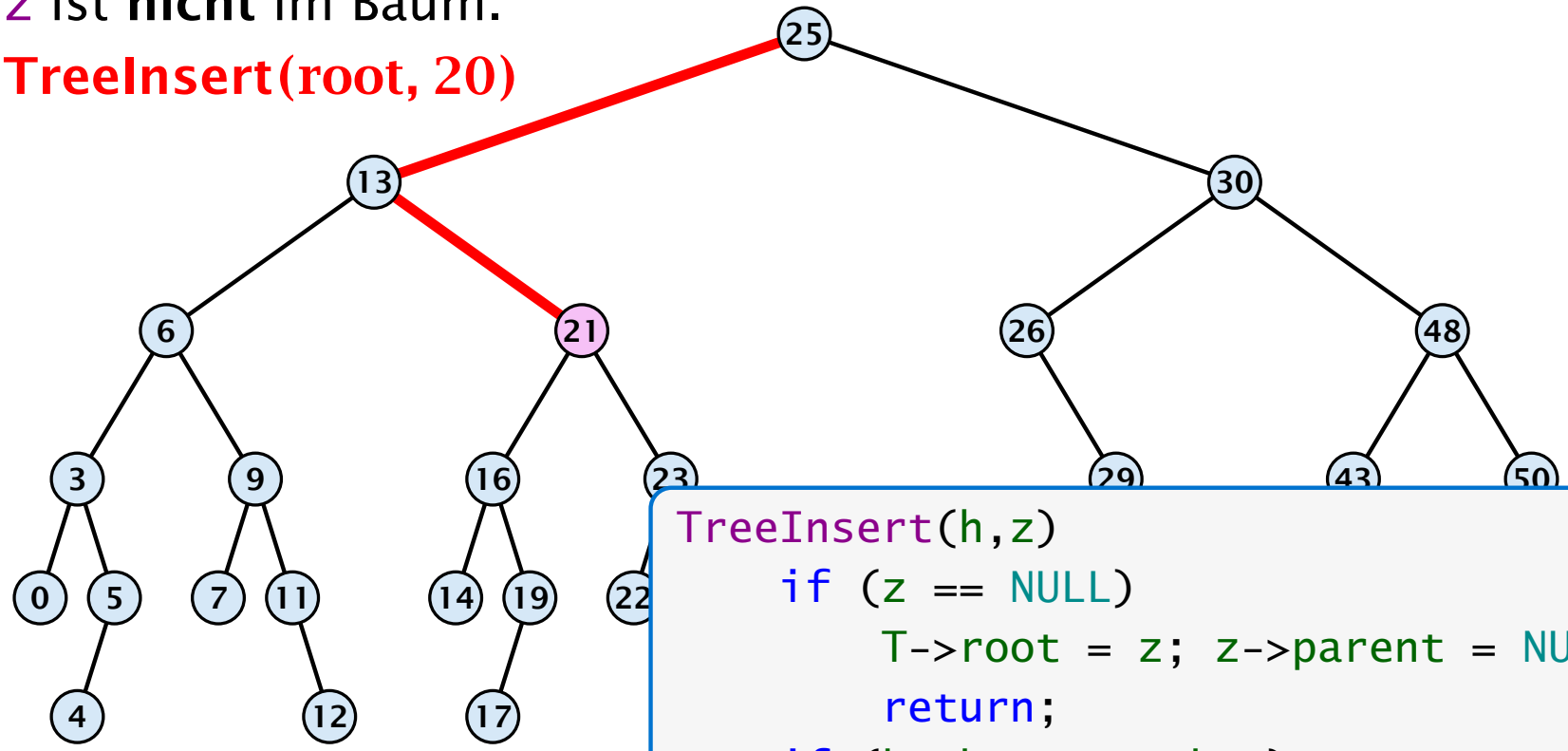
Suche nach z. Suche endet an NULL-Zeiger. Hier wird z eingefügt.

```
TreeInsert(h,z)
    if (z == NULL)
        T->root = z; z->parent = NULL;
        return;
    if (h->key > z->key)
        if (h->left == NULL)
            h->left = z; z->parent = h;
        else TreeInsert(h->left,z);
    else
        if (h->right == NULL)
            h->right = z; z->parent = h;
        else TreeInsert(h->right,z);
```

Binäre Suchbäume: Einfügen

z ist **nicht** im Baum.

TreeInsert(root, 20)



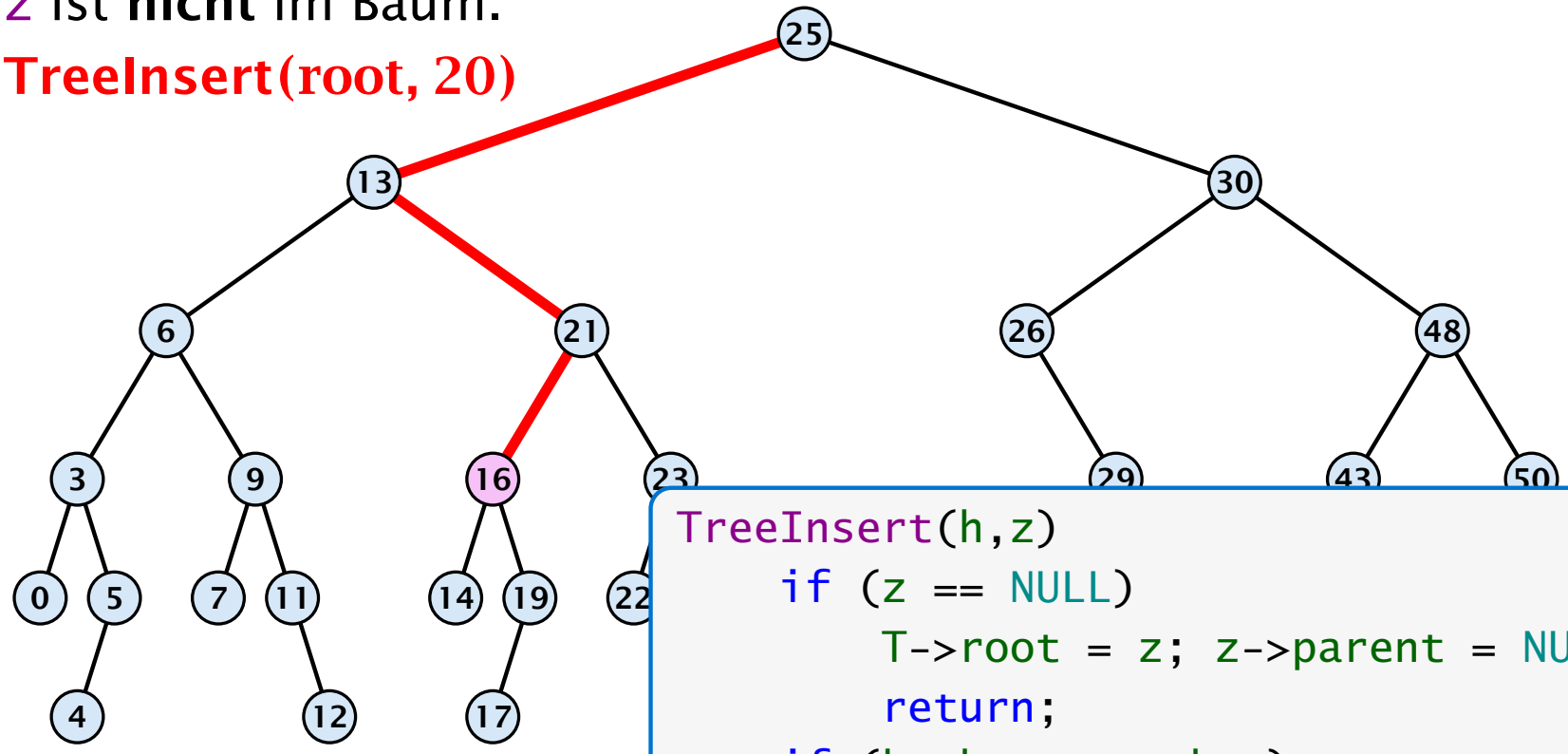
Suche nach z. Suche endet an NULL-Zeiger. Hier wird z eingefügt.

```
TreeInsert(h,z)
    if (z == NULL)
        T->root = z; z->parent = NULL;
        return;
    if (h->key > z->key)
        if (h->left == NULL)
            h->left = z; z->parent = h;
        else TreeInsert(h->left,z);
    else
        if (h->right == NULL)
            h->right = z; z->parent = h;
        else TreeInsert(h->right,z);
```

Binäre Suchbäume: Einfügen

z ist **nicht** im Baum.

TreeInsert(root, 20)



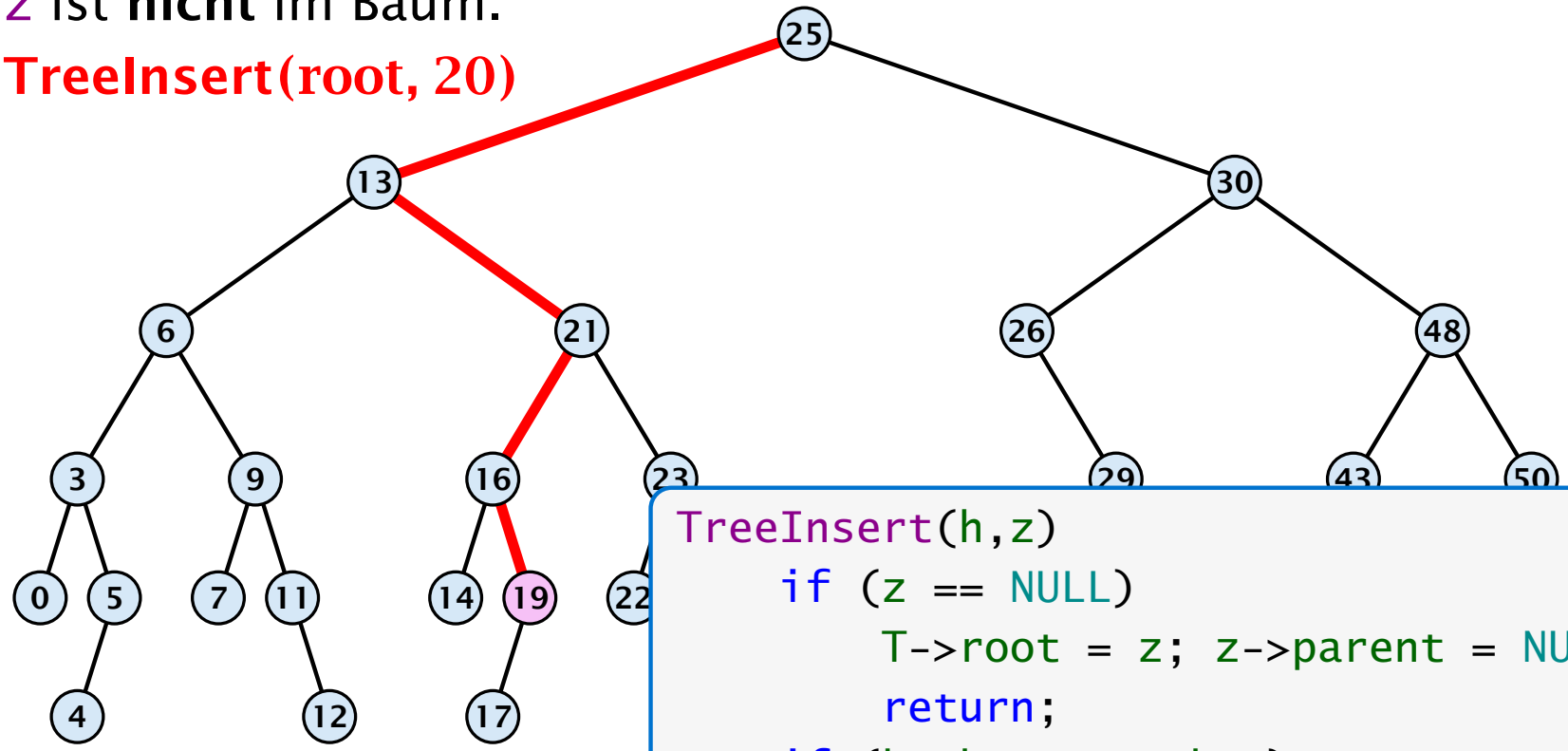
Suche nach z. Suche endet an NULL-Zeiger. Hier wird z eingefügt.

```
TreeInsert(h,z)
    if (z == NULL)
        T->root = z; z->parent = NULL;
        return;
    if (h->key > z->key)
        if (h->left == NULL)
            h->left = z; z->parent = h;
        else TreeInsert(h->left,z);
    else
        if (h->right == NULL)
            h->right = z; z->parent = h;
        else TreeInsert(h->right,z);
```

Binäre Suchbäume: Einfügen

z ist **nicht** im Baum.

TreeInsert(root, 20)



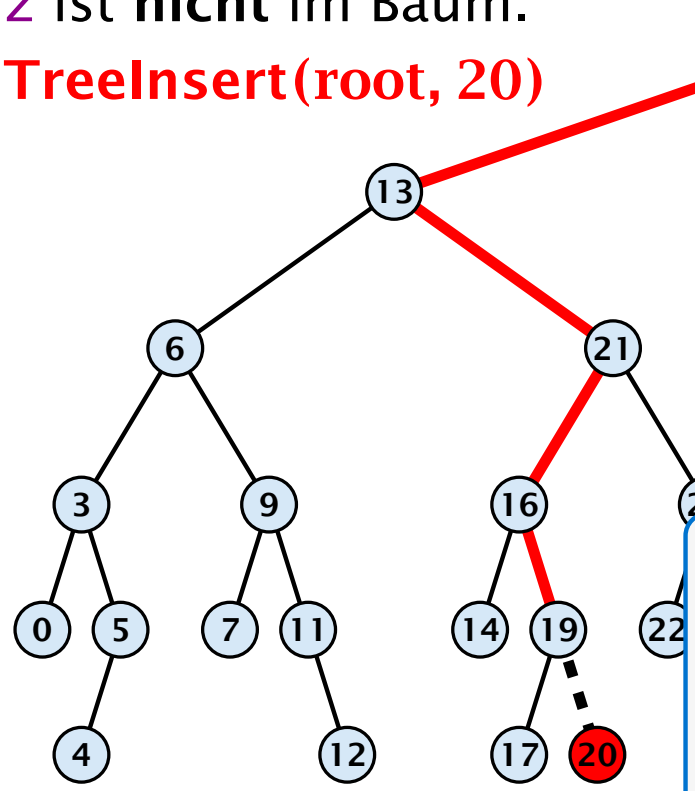
Suche nach z. Suche endet an NULL-Zeiger. Hier wird z eingefügt.

```
TreeInsert(h,z)
    if (z == NULL)
        T->root = z; z->parent = NULL;
        return;
    if (h->key > z->key)
        if (h->left == NULL)
            h->left = z; z->parent = h;
        else TreeInsert(h->left,z);
    else
        if (h->right == NULL)
            h->right = z; z->parent = h;
        else TreeInsert(h->right,z);
```

Binäre Suchbäume: Einfügen

z ist **nicht** im Baum.

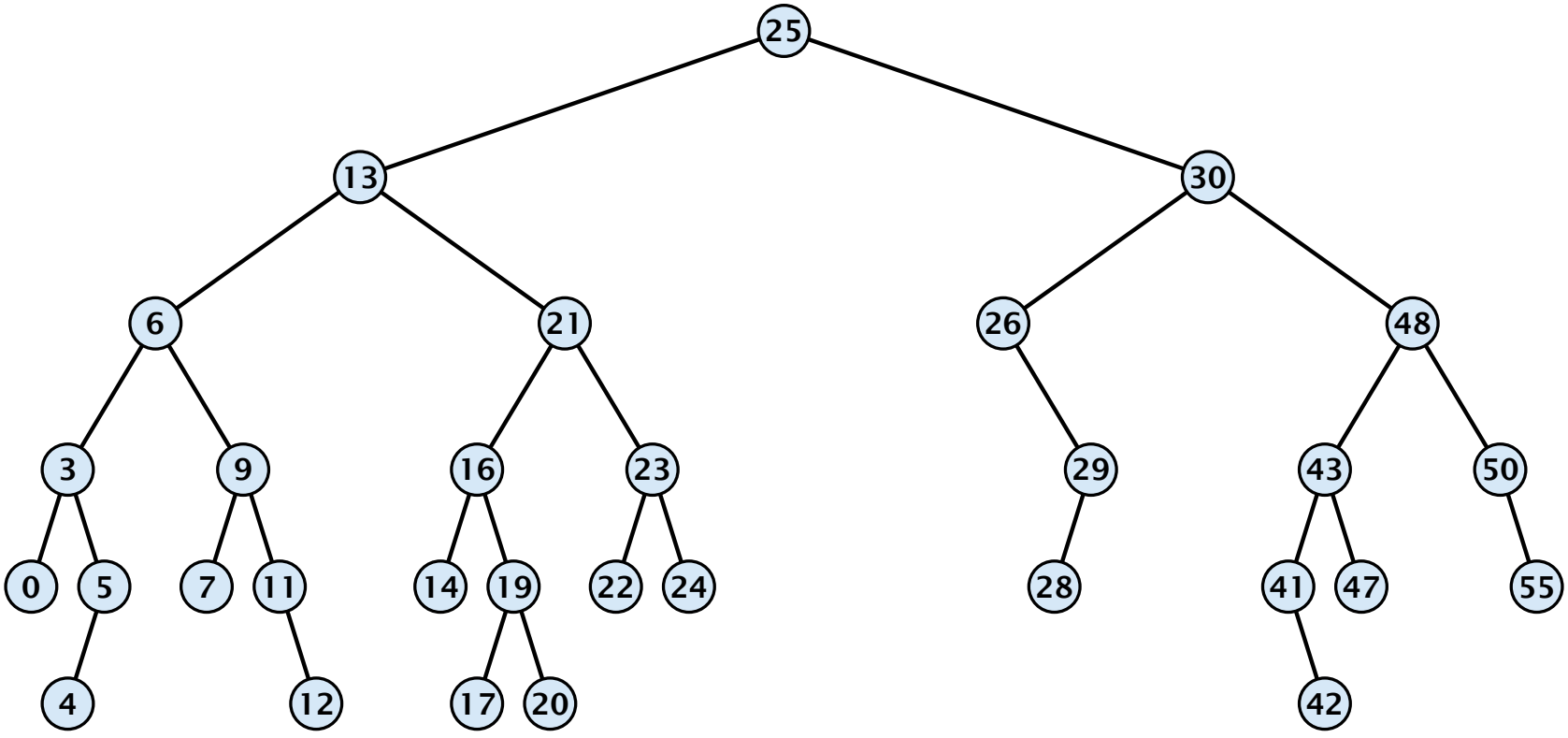
TreeInsert(root, 20)



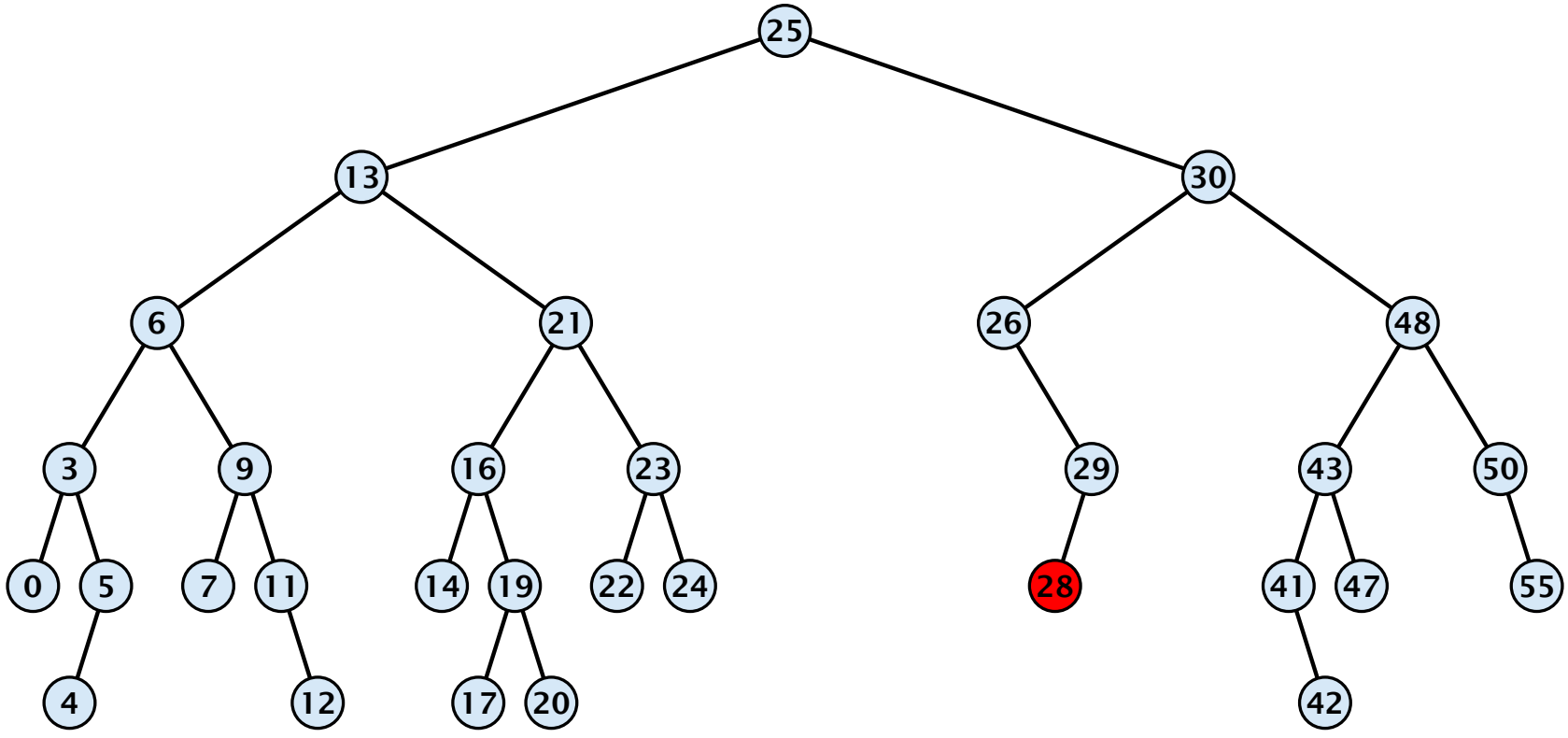
Suche nach z. Suche endet an NULL-Zeiger. Hier wird z eingefügt.

```
TreeInsert(h,z)
    if (z == NULL)
        T->root = z; z->parent = NULL;
        return;
    if (h->key > z->key)
        if (h->left == NULL)
            h->left = z; z->parent = h;
        else TreeInsert(h->left,z);
    else
        if (h->right == NULL)
            h->right = z; z->parent = h;
        else TreeInsert(h->right,z);
```

Binäre Suchbäume: Löschen



Binäre Suchbäume: Löschen

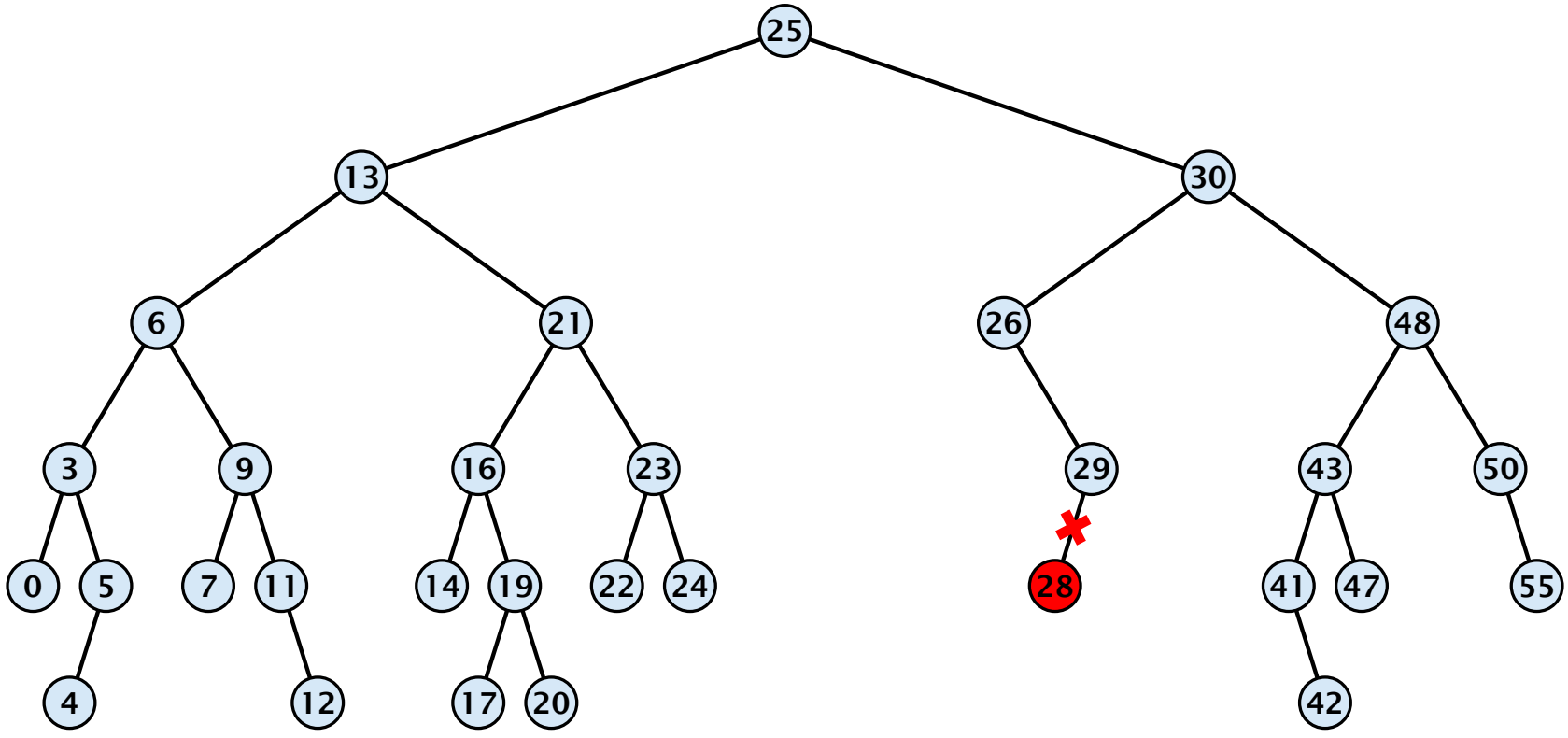


Fall 1:

Element hat keine Kinder

- Der Kindzeiger am Elternknoten wird auf NULL gesetzt.

Binäre Suchbäume: Löschen

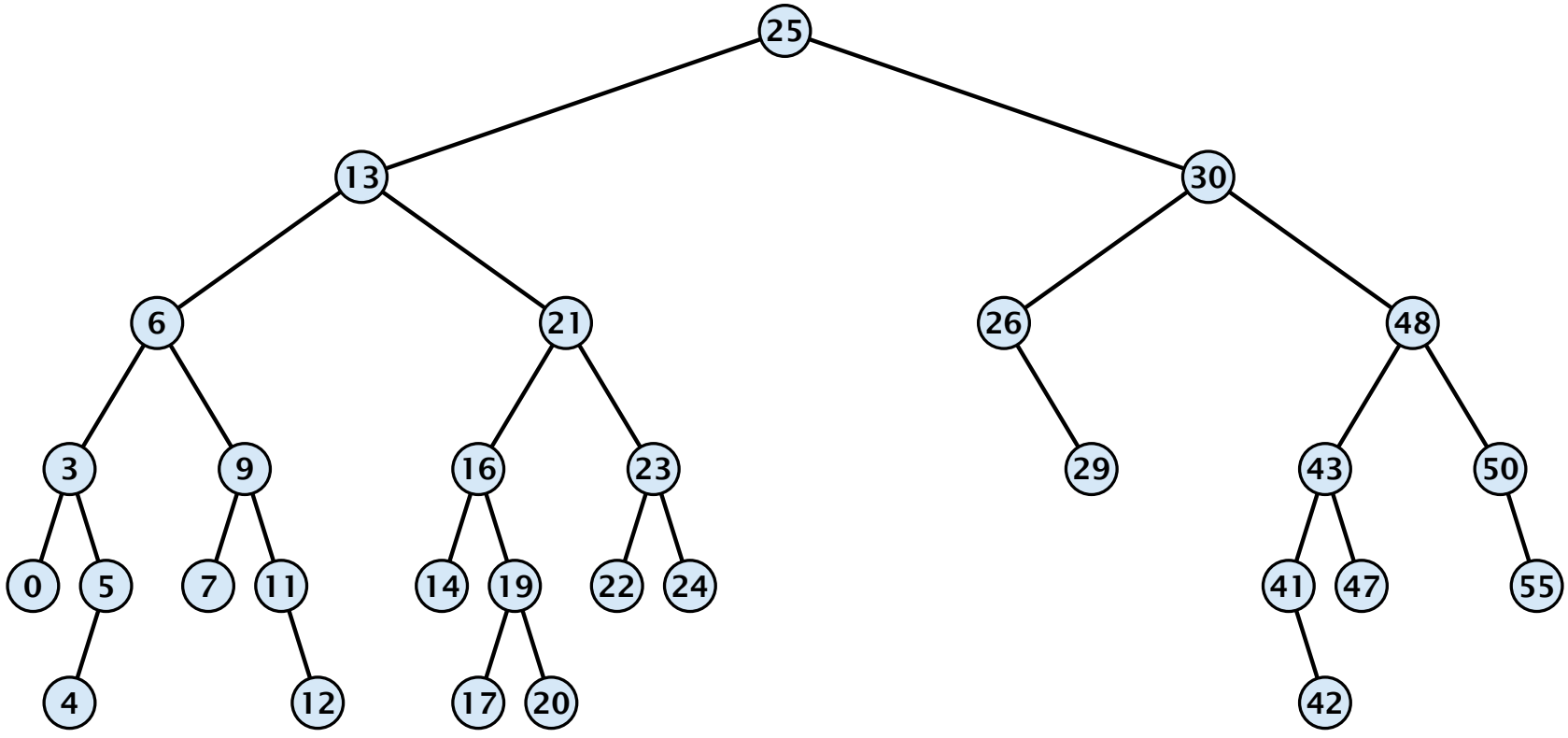


Fall 1:

Element hat keine Kinder

- Der Kindzeiger am Elternknoten wird auf NULL gesetzt.

Binäre Suchbäume: Löschen

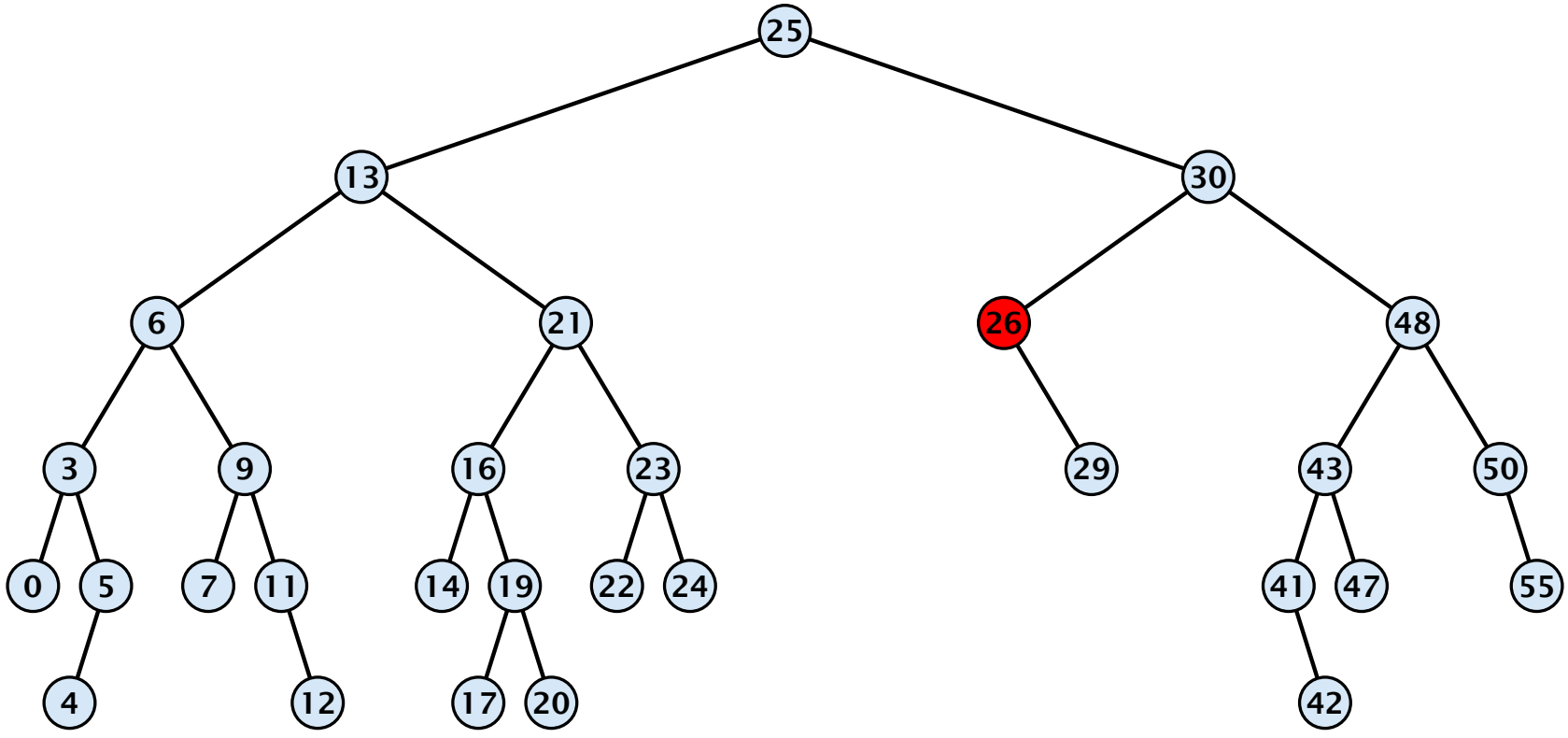


Fall 1:

Element hat keine Kinder

- Der Kindzeiger am Elternknoten wird auf NULL gesetzt.

Binäre Suchbäume: Löschen

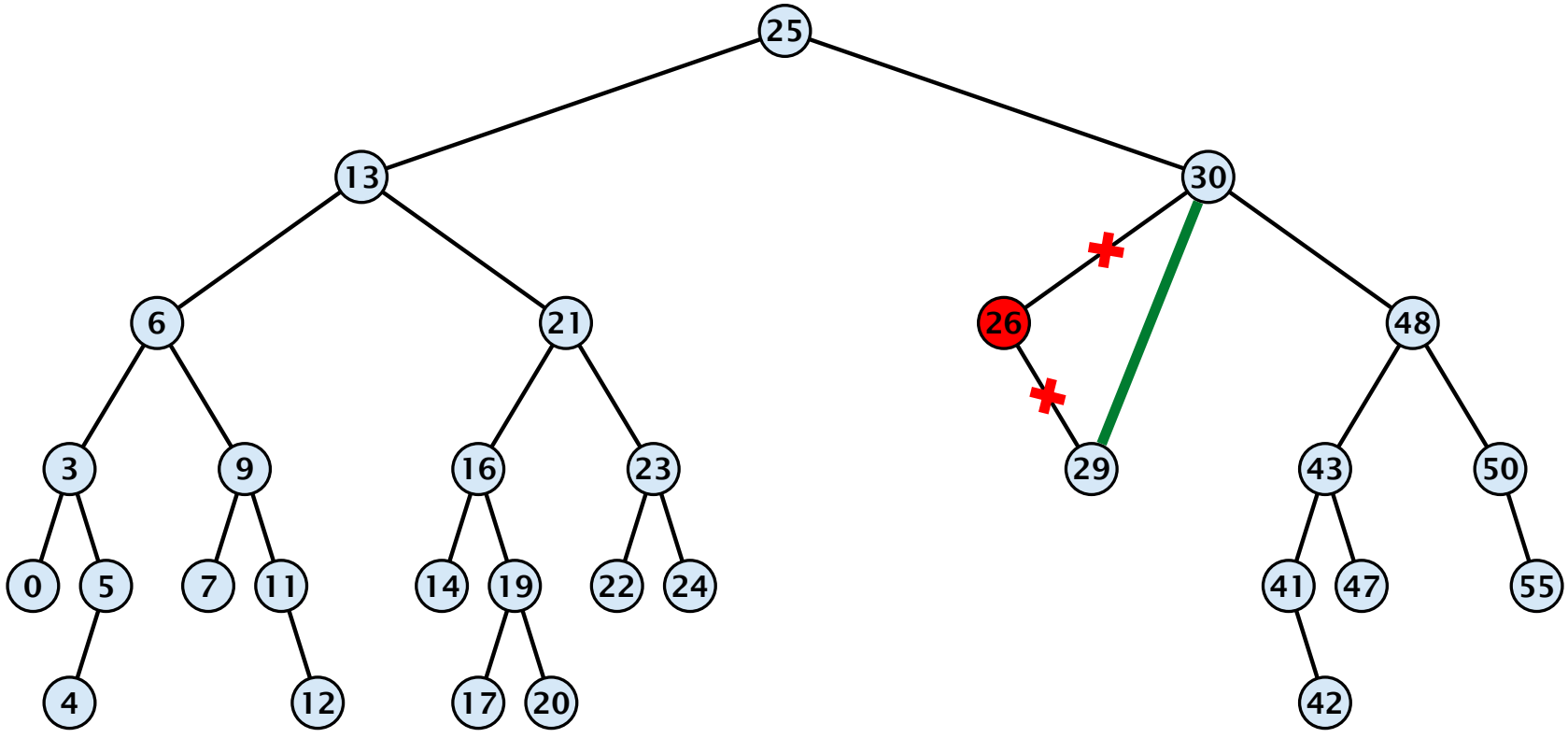


Fall 2:

Element hat genau ein Kind

- Überbrücke Element indem man Elternknoten mit Kind verbindet.

Binäre Suchbäume: Löschen

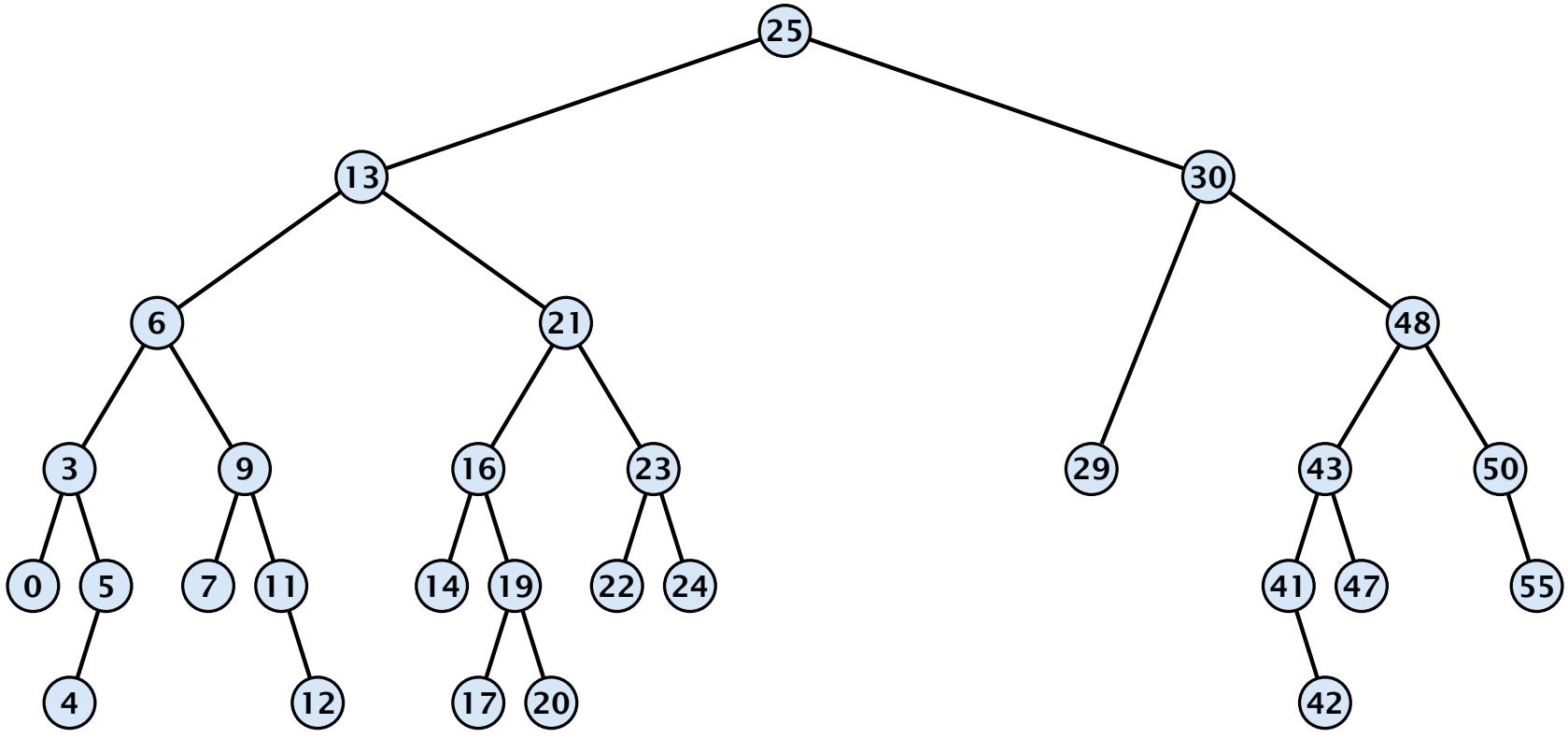


Fall 2:

Element hat genau ein Kind

- Überbrücke Element indem man Elternknoten mit Kind verbindet.

Binäre Suchbäume: Löschen

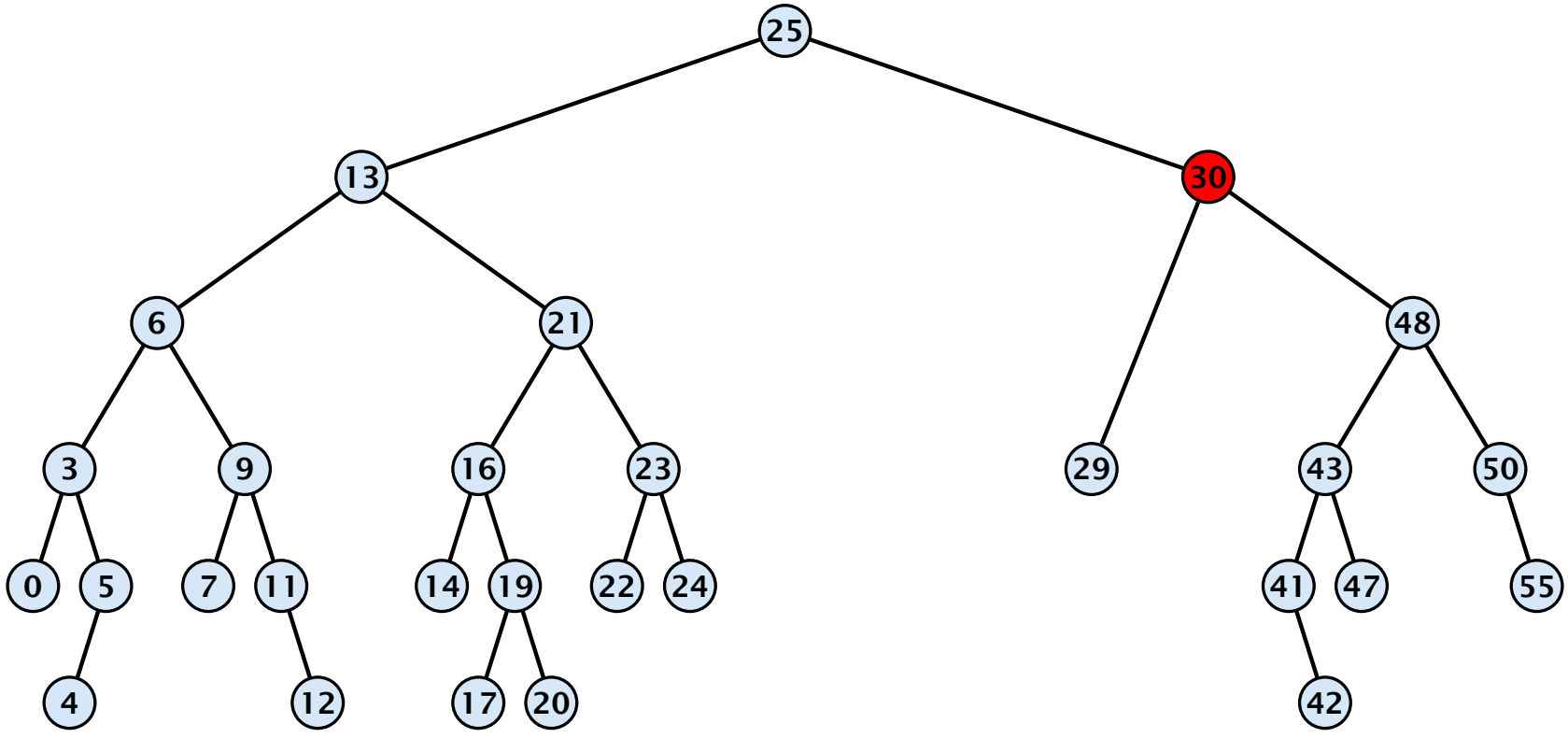


Fall 2:

Element hat genau ein Kind

- Überbrücke Element indem man Elternknoten mit Kind verbindet.

Binäre Suchbäume: Löschen

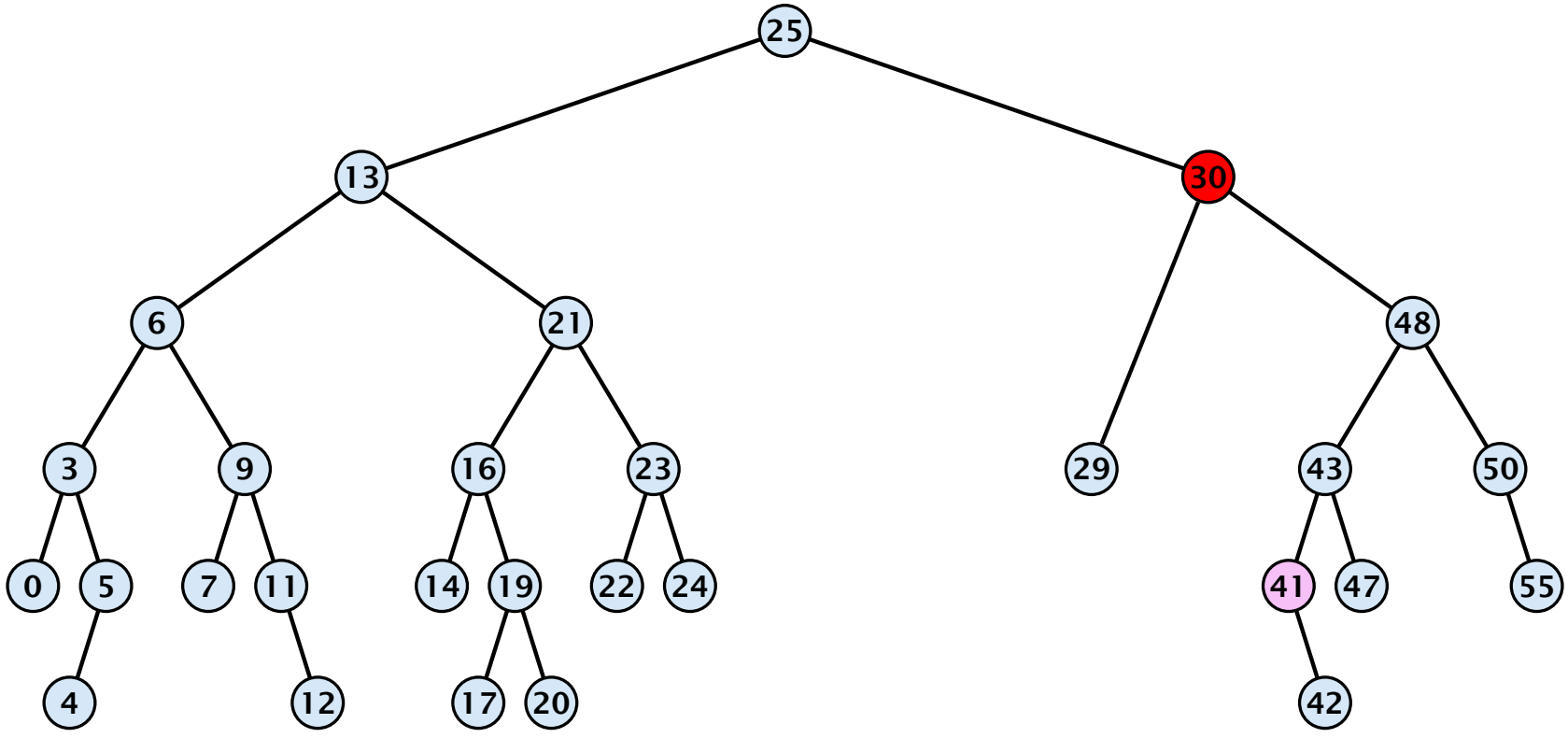


Case 3:

Element hat zwei Kinder

- Finde Nachfolger
- Überbrücke den Nachfolger
- Ersetze Element durch Nachfolger

Binäre Suchbäume: Löschen

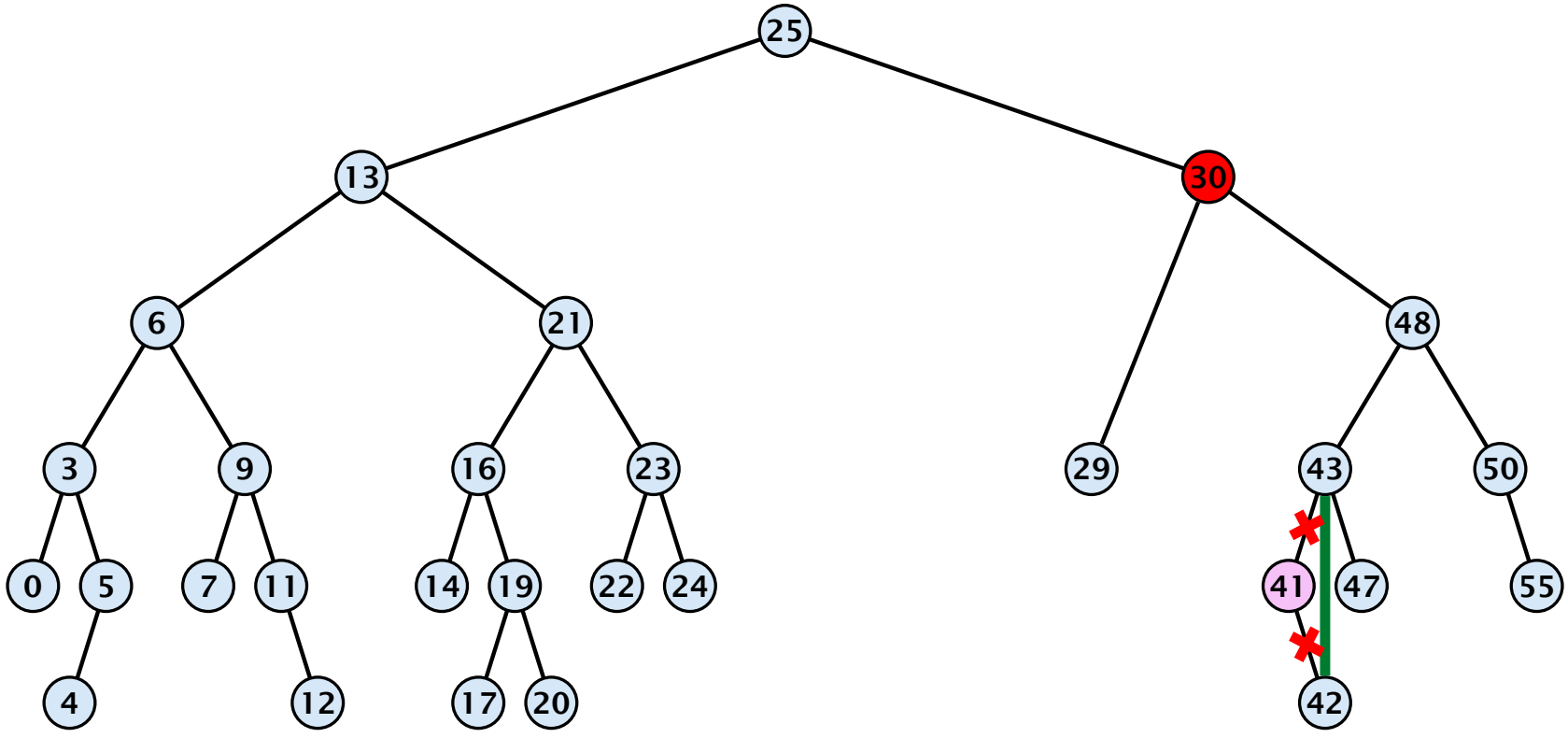


Case 3:

Element hat zwei Kinder

- Finde Nachfolger
- Überbrücke den Nachfolger
- Ersetze Element durch Nachfolger

Binäre Suchbäume: Löschen

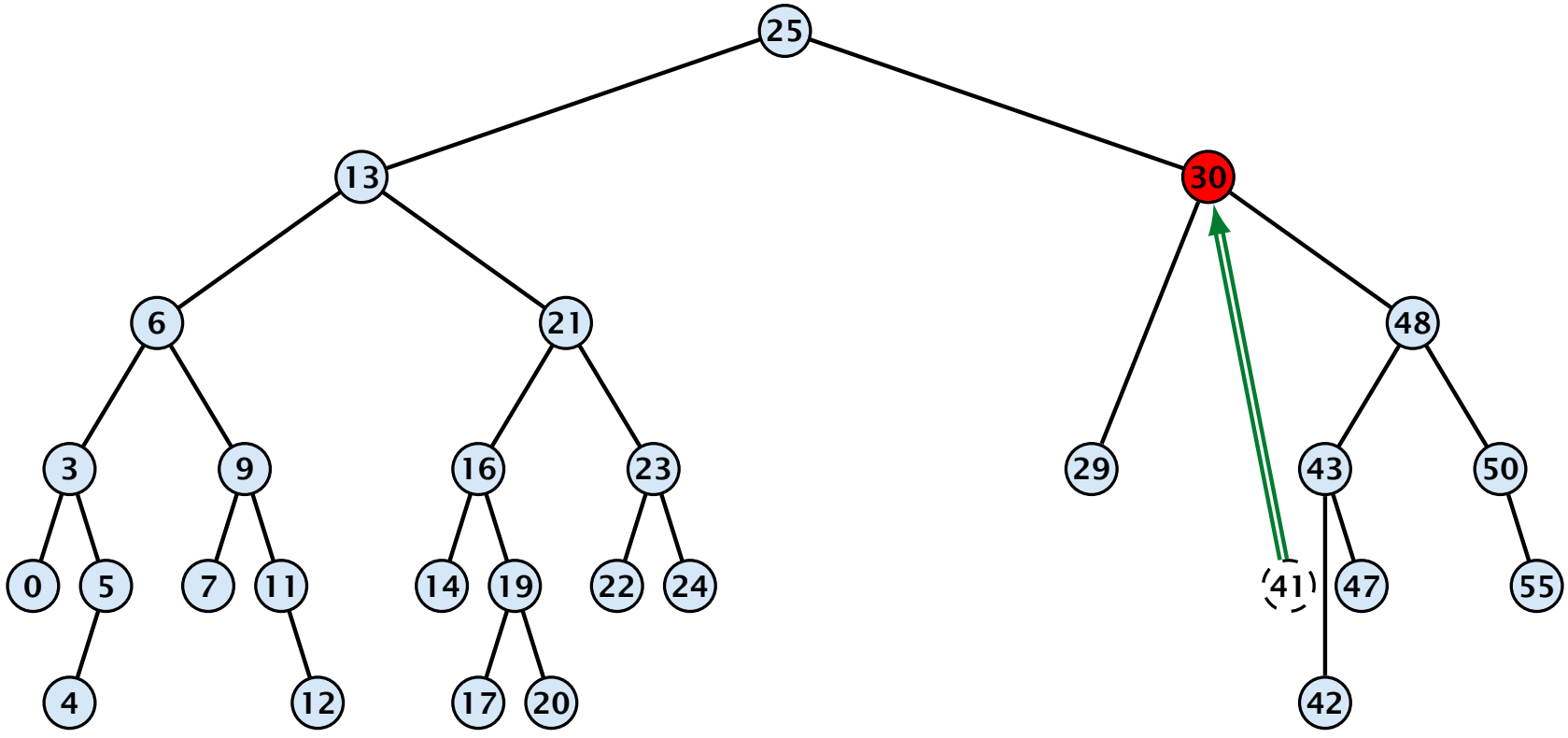


Case 3:

Element hat zwei Kinder

- Finde Nachfolger
- Überbrücke den Nachfolger
- Ersetze Element durch Nachfolger

Binäre Suchbäume: Löschen

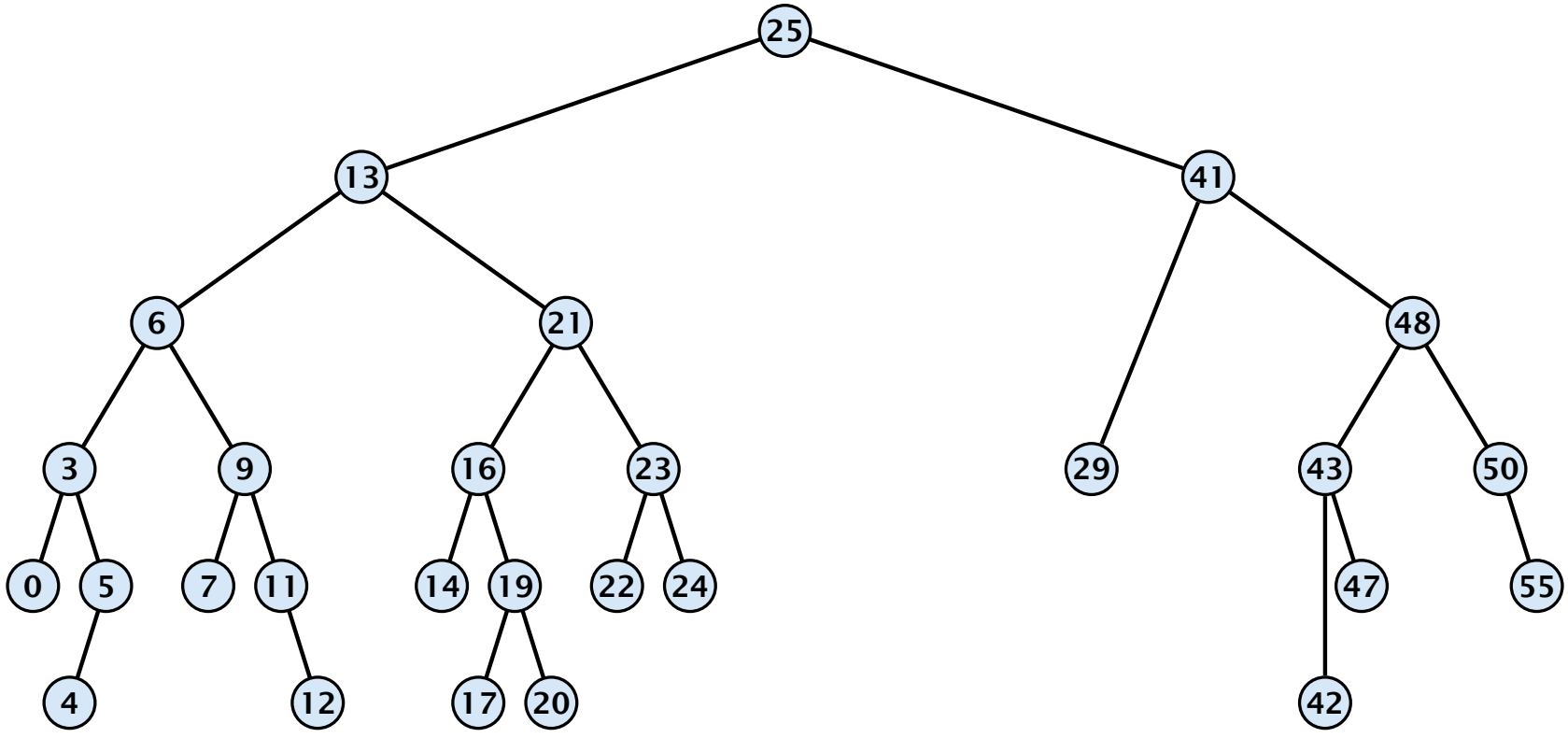


Case 3:

Element hat zwei Kinder

- Finde Nachfolger
- Überbrücke den Nachfolger
- Ersetze Element durch Nachfolger

Binäre Suchbäume: Löschen

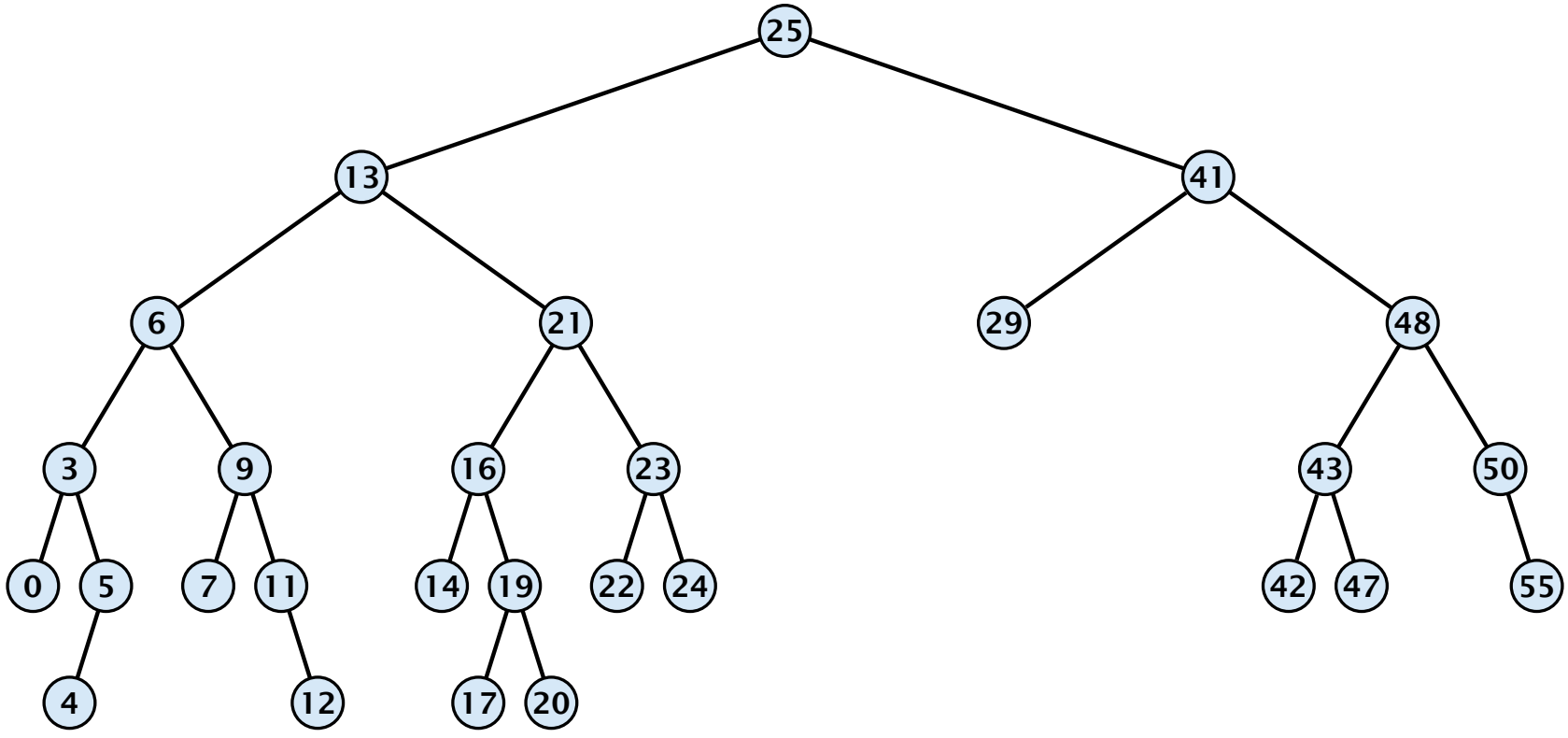


Case 3:

Element hat zwei Kinder

- Finde Nachfolger
- Überbrücke den Nachfolger
- Ersetze Element durch Nachfolger

Binäre Suchbäume: Löschen



Case 3:

Element hat zwei Kinder

- Finde Nachfolger
- Überbrücke den Nachfolger
- Ersetze Element durch Nachfolger

Binäre Suchbäume: Löschen

```
1  TreeDelete(z)
2      if (z->left == NULL || z->right == NULL)
3          y = z;
4      else y = TreeSucc(z);
5      if (y->left != NULL)
6          x = y->left;
7      else x = y->right;
8      if (x != NULL)
9          x->parent = y->parent;
10     if (y->parent == NULL)
11         T->root = x;
12     else
13         if (y->parent->left == y)
14             y->parent->left  = x;
15         else
16             y->parent->right = x;
17     if (y != z) replace z with y
```

Balancierte Suchbäume

Alle Operationen auf einem Binärbaum benötigen Zeit $\mathcal{O}(h)$, wobei h die Höhe des Baumes ist.

Die Höhe kann aber $\Theta(n)$ werden.

Balancierte Suchbäume
Bei jeder insert- oder delete-Operation werden zusätzlich lokale Änderungen durchgeführt, die garantieren dass die Höhe immer in $\mathcal{O}(\log n)$ ist.

AVL-trees, Red-black trees, Scapegoat trees, 2-3 trees, B-trees, AA trees, Treaps

ähnlich: SPLAY trees.

Balancierte Suchbäume

Alle Operationen auf einem Binärbaum benötigen Zeit $\mathcal{O}(h)$, wobei h die Höhe des Baumes ist.

Die Höhe kann aber $\Theta(n)$ werden.

Balancierte Suchbäume

Bei jeder insert- oder delete-Operation werden zusätzlich lokale Änderungen durchgeführt, die garantieren dass die Höhe immer in $\mathcal{O}(\log n)$ ist.

AVL-trees, Red-black trees, Scapegoat trees, 2-3 trees, B-trees, AA trees, Treaps

ähnlich: SPLAY trees.

Balancierte Suchbäume

Alle Operationen auf einem Binärbaum benötigen Zeit $\mathcal{O}(h)$, wobei h die Höhe des Baumes ist.

Die Höhe kann aber $\Theta(n)$ werden.

Balancierte Suchbäume

Bei jeder insert- oder delete-Operation werden zusätzlich lokale Änderungen durchgeführt, die garantieren dass die Höhe immer in $\mathcal{O}(\log n)$ ist.

AVL-trees, Red-black trees, Scapegoat trees, 2-3 trees, B-trees, AA trees, Treaps

ähnlich: SPLAY trees.

Balancierte Suchbäume

Alle Operationen auf einem Binärbaum benötigen Zeit $\mathcal{O}(h)$, wobei h die Höhe des Baumes ist.

Die Höhe kann aber $\Theta(n)$ werden.

Balancierte Suchbäume
Bei jeder insert- oder delete-Operation werden zusätzlich lokale Änderungen durchgeführt, die garantieren dass die Höhe immer in $\mathcal{O}(\log n)$ ist.

AVL-trees, Red-black trees, Scapegoat trees, 2-3 trees, B-trees, AA trees, Treaps

ähnlich: SPLAY trees.

Balancierte Suchbäume

Alle Operationen auf einem Binärbaum benötigen Zeit $\mathcal{O}(h)$, wobei h die Höhe des Baumes ist.

Die Höhe kann aber $\Theta(n)$ werden.

Balancierte Suchbäume
Bei jeder insert- oder delete-Operation werden zusätzlich lokale Änderungen durchgeführt, die garantieren dass die Höhe immer in $\mathcal{O}(\log n)$ ist.

AVL-trees, Red-black trees, Scapegoat trees, 2-3 trees, B-trees, AA trees, Treaps

ähnlich: SPLAY trees.