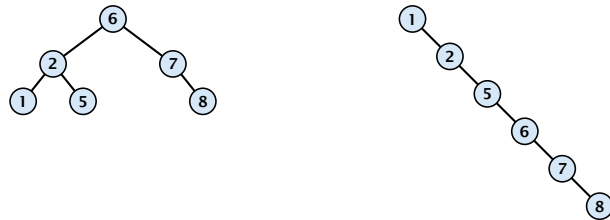


7.1 Binary Search Trees

An (internal) **binary search tree** stores the elements in a binary tree. Each tree-node corresponds to an element. All elements in the left sub-tree of a node v have a smaller key-value than $\text{key}[v]$ and elements in the right sub-tree have a larger-key value. We assume that all key-values are different.

(External Search Trees store objects only at leaf-vertices)

Examples:



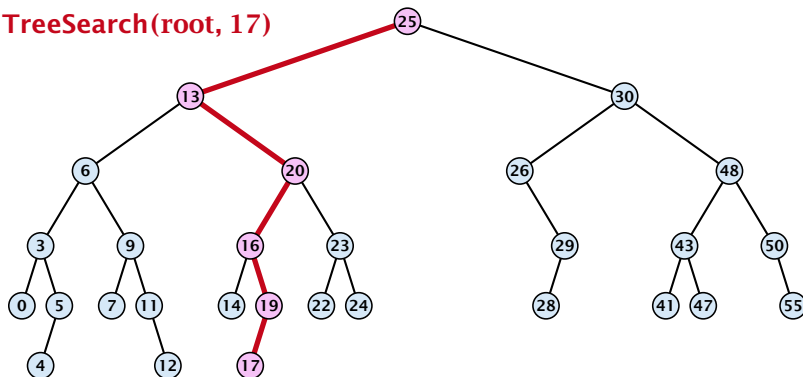
7.1 Binary Search Trees

We consider the following operations on binary search trees. Note that this is a super-set of the dictionary-operations.

- ▶ $T.\text{insert}(x)$
- ▶ $T.\text{delete}(x)$
- ▶ $T.\text{search}(k)$
- ▶ $T.\text{successor}(x)$
- ▶ $T.\text{predecessor}(x)$
- ▶ $T.\text{minimum}()$
- ▶ $T.\text{maximum}()$

Binary Search Trees: Searching

TreeSearch(root, 17)

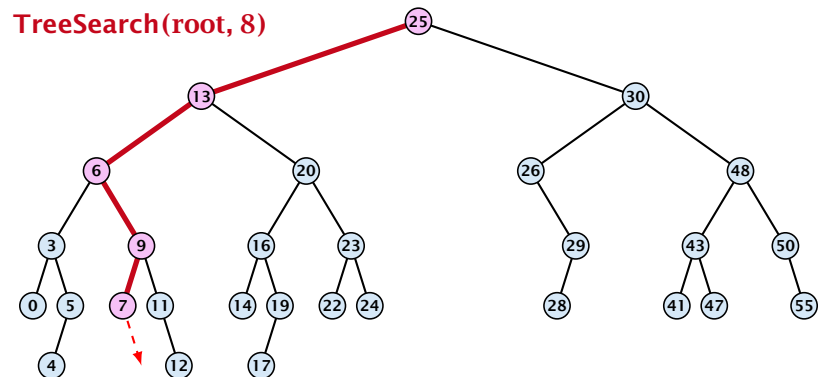


Algorithm 1 $\text{TreeSearch}(x, k)$

- 1: **if** $x = \text{null}$ **or** $k = \text{key}[x]$ **return** x
- 2: **if** $k < \text{key}[x]$ **return** $\text{TreeSearch}(\text{left}[x], k)$
- 3: **else return** $\text{TreeSearch}(\text{right}[x], k)$

Binary Search Trees: Searching

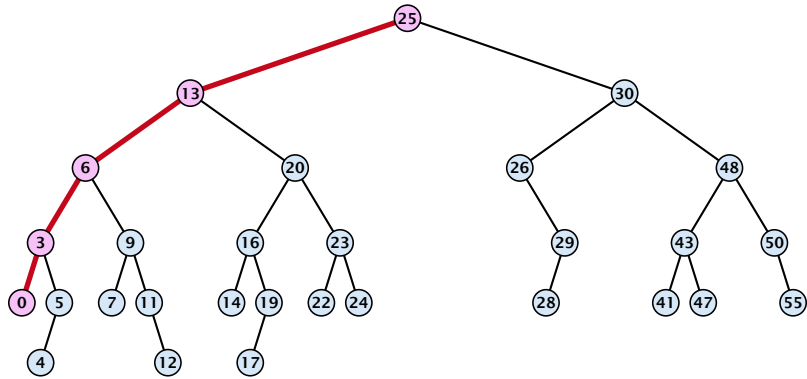
TreeSearch(root, 8)



Algorithm 1 $\text{TreeSearch}(x, k)$

- 1: **if** $x = \text{null}$ **or** $k = \text{key}[x]$ **return** x
- 2: **if** $k < \text{key}[x]$ **return** $\text{TreeSearch}(\text{left}[x], k)$
- 3: **else return** $\text{TreeSearch}(\text{right}[x], k)$

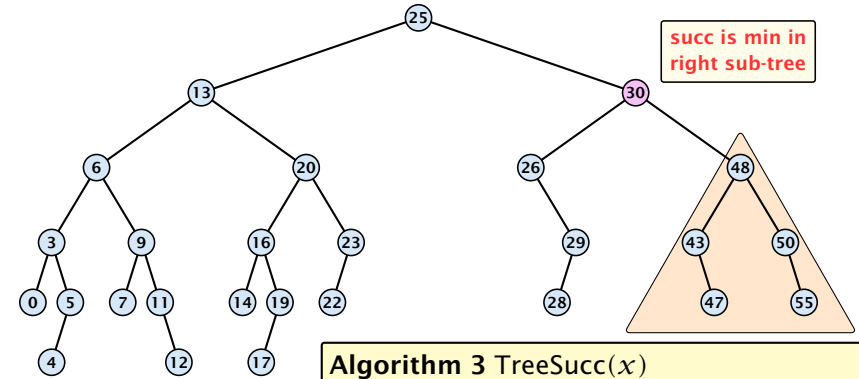
Binary Search Trees: Minimum



Algorithm 2 TreeMin(x)

- 1: if $x = \text{null}$ or $\text{left}[x] = \text{null}$ return x
- 2: return $\text{TreeMin}(\text{left}[x])$

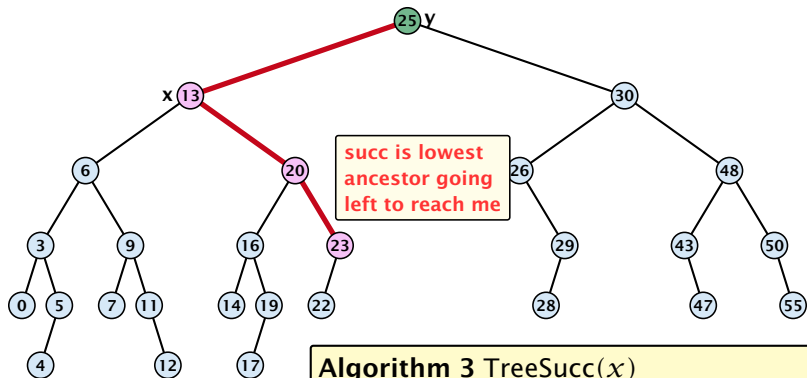
Binary Search Trees: Successor



Algorithm 3 TreeSucc(x)

- 1: if $\text{right}[x] \neq \text{null}$ return $\text{TreeMin}(\text{right}[x])$
- 2: $y \leftarrow \text{parent}[x]$
- 3: while $y \neq \text{null}$ and $x = \text{right}[y]$ do
- 4: $x \leftarrow y; y \leftarrow \text{parent}[x]$
- 5: return y ;

Binary Search Trees: Successor



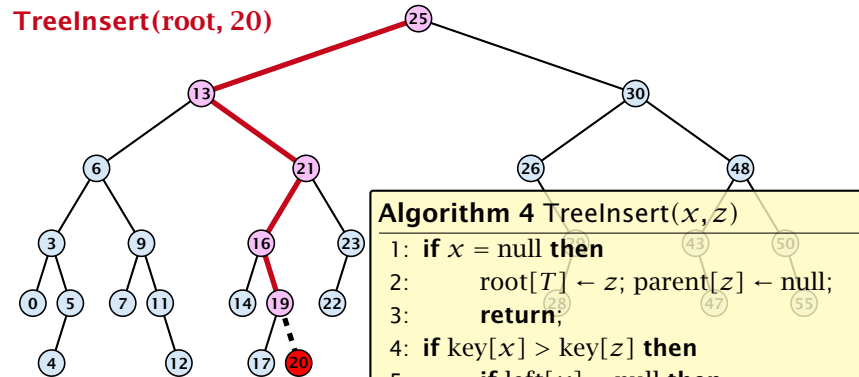
Algorithm 3 TreeSucc(x)

- 1: if $\text{right}[x] \neq \text{null}$ return $\text{TreeMin}(\text{right}[x])$
- 2: $y \leftarrow \text{parent}[x]$
- 3: while $y \neq \text{null}$ and $x = \text{right}[y]$ do
- 4: $x \leftarrow y; y \leftarrow \text{parent}[x]$
- 5: return y ;

Binary Search Trees: Insert

Insert element **not** in the tree.

TreeInsert(root, 20)

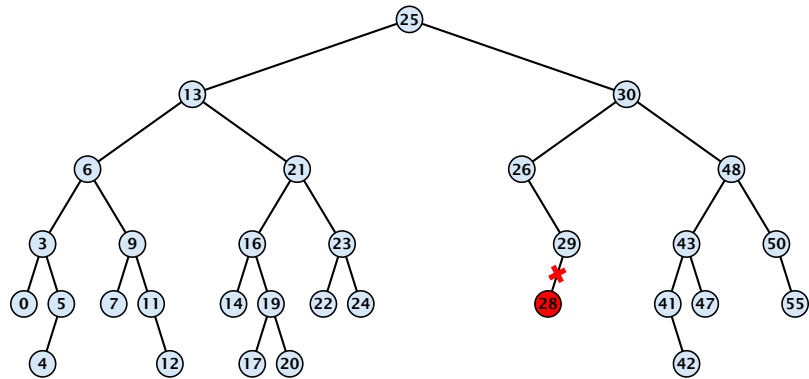


Algorithm 4 TreeInsert(x, z)

- 1: if $x = \text{null}$ then
- 2: $\text{root}[T] \leftarrow z; \text{parent}[z] \leftarrow \text{null};$
- 3: return;
- 4: if $\text{key}[x] > \text{key}[z]$ then
- 5: if $\text{left}[x] = \text{null}$ then
- 6: $\text{left}[x] \leftarrow z; \text{parent}[z] \leftarrow x;$
- 7: else $\text{TreeInsert}(\text{left}[x], z);$
- 8: else
- 9: if $\text{right}[x] = \text{null}$ then
- 10: $\text{right}[x] \leftarrow z; \text{parent}[z] \leftarrow x;$
- 11: else $\text{TreeInsert}(\text{right}[x], z);$

Search for z . At some point the search stops at a null-pointer. This is the place to insert z .

Binary Search Trees: Delete

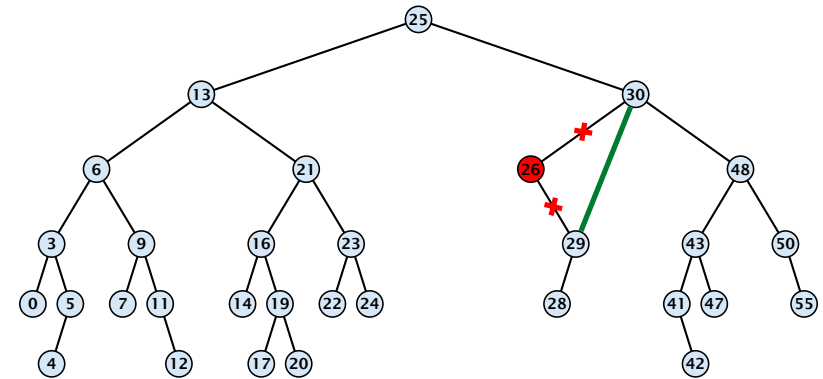


Case 1:

Element does not have any children

- ▶ Simply go to the parent and set the corresponding pointer to **null**.

Binary Search Trees: Delete

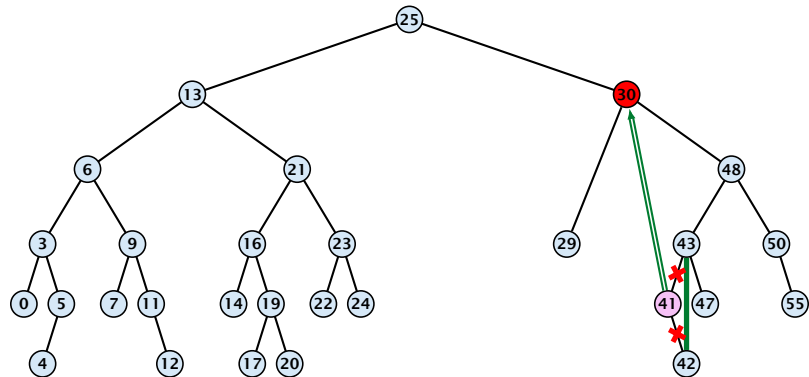


Case 2:

Element has exactly one child

- ▶ Splice the element out of the tree by connecting its parent to its successor.

Binary Search Trees: Delete



Case 3:

Element has two children

- ▶ Find the successor of the element
- ▶ Splice successor out of the tree
- ▶ Replace content of element by content of successor

Binary Search Trees: Delete

Algorithm 9 TreeDelete(z)

```

1: if left[z] = null or right[z] = null
2:   then  $y \leftarrow z$  else  $y \leftarrow \text{TreeSucc}(z)$ ;   select  $y$  to splice out
3: if left[y]  $\neq$  null
4:   then  $x \leftarrow \text{left}[y]$  else  $x \leftarrow \text{right}[y]$ ;  $x$  is child of  $y$  (or null)
5: if  $x \neq \text{null}$  then parent[x]  $\leftarrow$  parent[y];   parent[x] is correct
6: if parent[y] = null then
7:   root[T]  $\leftarrow$   $x$ 
8: else
9:   if  $y = \text{left}[\text{parent}[y]]$  then
10:    left[parent[y]]  $\leftarrow$   $x$ 
11:   else
12:    right[parent[y]]  $\leftarrow$   $x$ 
13: if  $y \neq z$  then copy  $y$ -data to  $z$ 
    
```

Balanced Binary Search Trees

All operations on a binary search tree can be performed in time $\mathcal{O}(h)$, where h denotes the height of the tree.

However the height of the tree may become as large as $\Theta(n)$.

Balanced Binary Search Trees

With each insert- and delete-operation perform **local** adjustments to guarantee a height of $\mathcal{O}(\log n)$.

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

similar: SPLAY trees.

Binary Search Trees (BSTs)

Bibliography

[MS08] Kurt Mehlhorn, Peter Sanders:
Algorithms and Data Structures — The Basic Toolbox,
Springer, 2008

[CLRS90] Thomas H. Cormen, Charles E. Leiserson, Ron L. Rivest, Clifford Stein:
Introduction to Algorithms (3rd ed.),
MIT Press and McGraw-Hill, 2009

Binary search trees can be found in every standard text book. For example Chapter 7.1 in [MS08] and Chapter 12 in [CLRS90].