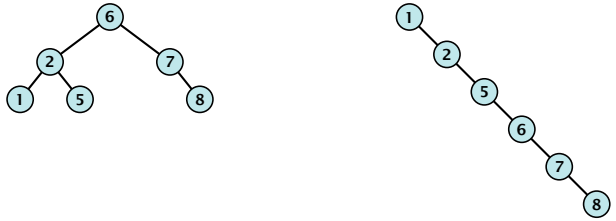
7.1 Binary Search Trees

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:



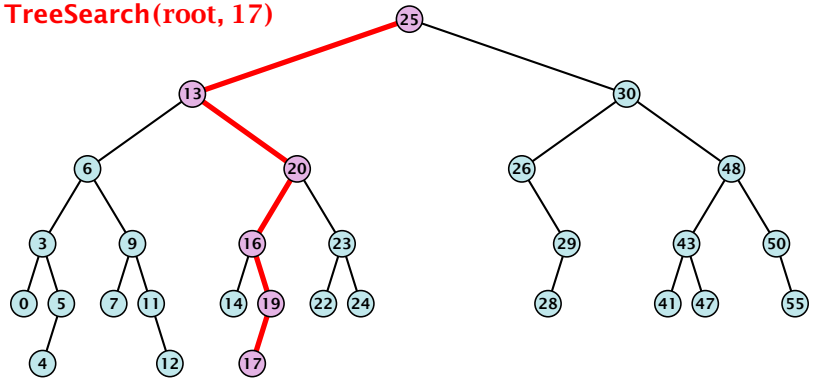
7.1 Binary Search Trees

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

- T.insert(x)
- T.delete(x)
- T.search(k)
- T.successor(x)
- T.predecessor(x)
- T.minimum()
- T.maximum()

Binary Search Trees: Searching

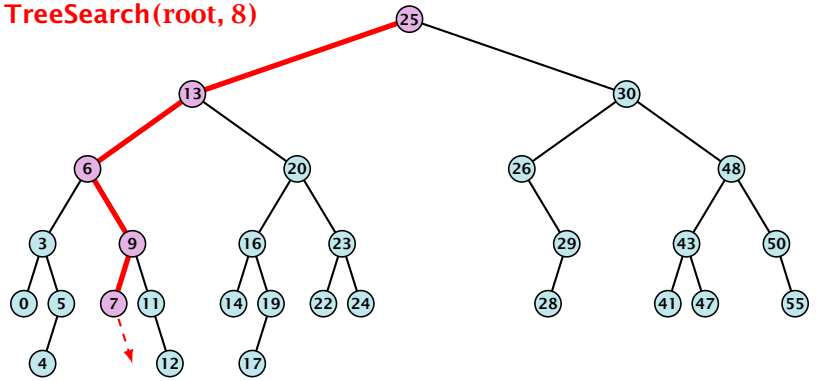
TreeSearch(root, 17)



Algorithm 5 TreeSearch(x, k)
1: **if** $x = \text{null}$ **or** $k = \text{key}[x]$ **return** x
2: **if** $k < \text{key}[x]$ **return** TreeSearch$(\text{left}[x], k)$
3: **else return** TreeSearch$(\text{right}[x], k)$

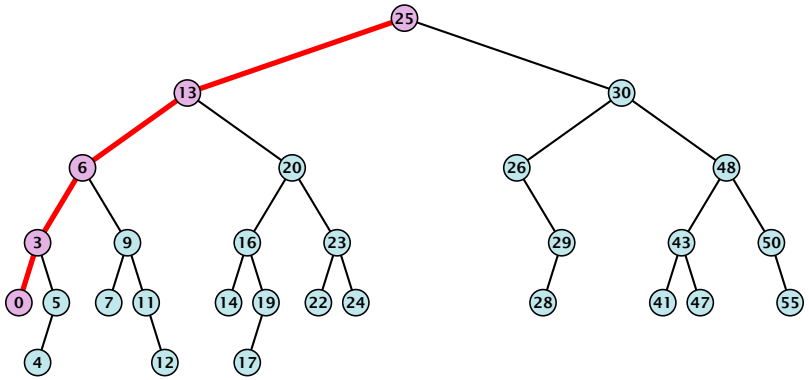
Binary Search Trees: Searching

TreeSearch(root, 8)



Algorithm 5 TreeSearch(x, k)
1: **if** $x = \text{null}$ **or** $k = \text{key}[x]$ **return** x
2: **if** $k < \text{key}[x]$ **return** TreeSearch$(\text{left}[x], k)$
3: **else return** TreeSearch$(\text{right}[x], k)$

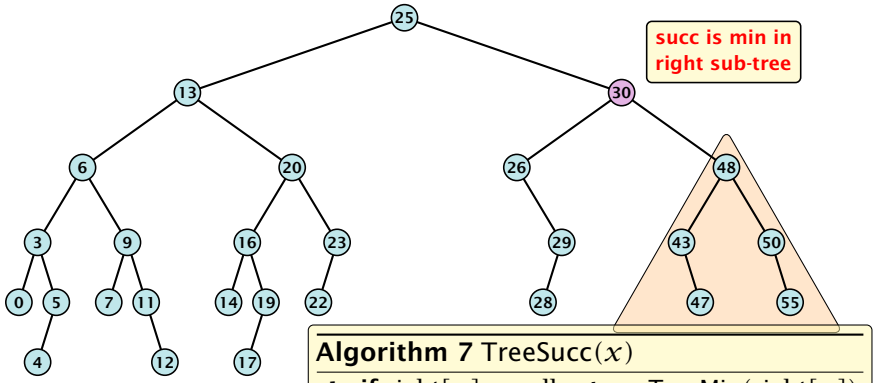
Binary Search Trees: Minimum



Algorithm 6 TreeMin(x)

```
1: if x = null or left[x] = null return x
2: if k < key[x] return TreeSearch(left[x], k)
3: else return TreeMin(left[x])
```

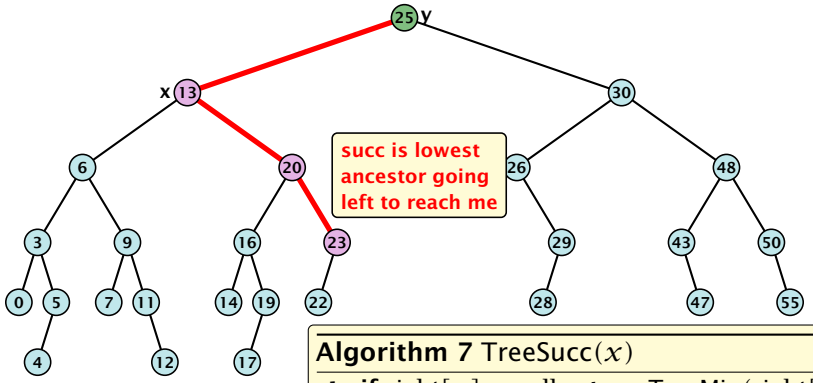

Binary Search Trees: Successor



Algorithm 7 TreeSucc(x)

```
1: if right[x] ≠ null return TreeMin(right[x])
2: y ← parent[x]
3: while y ≠ null and x = right[y] do
4:       x ← y; y ← parent[x]
5: return y;
```


Binary Search Trees: Successor



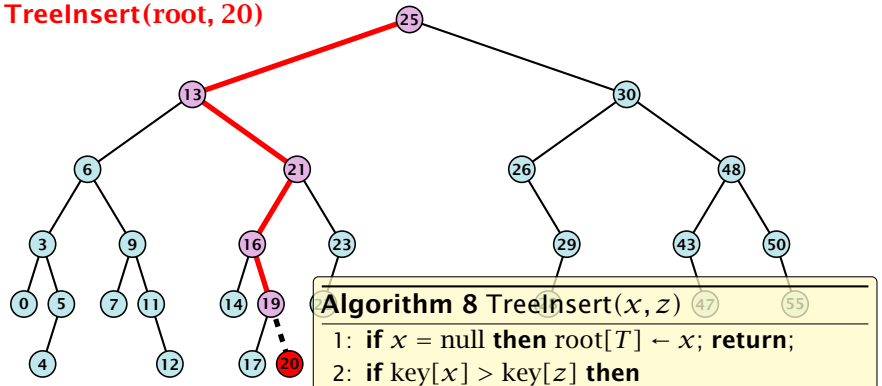
Algorithm 7 TreeSucc(x)

```
1: if right[x] ≠ null return TreeMin(right[x])
2: y ← parent[x]
3: while y ≠ null and x = right[y] do
4:       x ← y; y ← parent[x]
5: return y;
```


Binary Search Trees: Insert

Insert element **not** in the tree.

TreeInsert(root, 20)

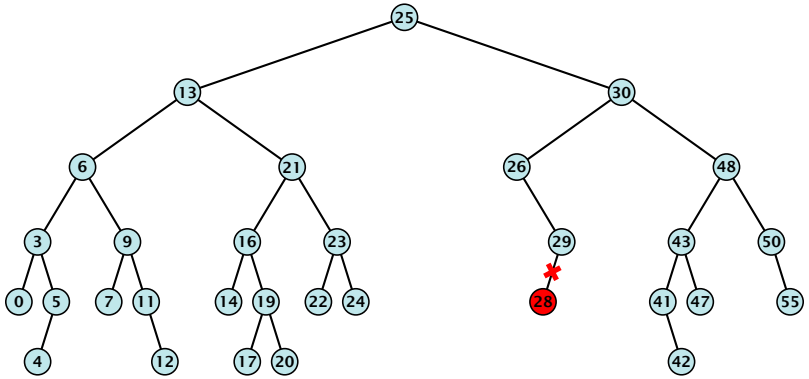


Search for z. At some point the search stops at a null-pointer. This is the place to insert z.

Algorithm 8 TreeInsert(x, z)

```
1: if x = null then root[T] ← x; return;
2: if key[x] > key[z] then
3:       if left[x] = null then left[x] ← z;
4:       else TreeInsert(left[x], z);
5: else
6:       if right[x] = null then right[x] ← z;
7:       else TreeInsert(right[x], z);
8: return
```

Binary Search Trees: Delete

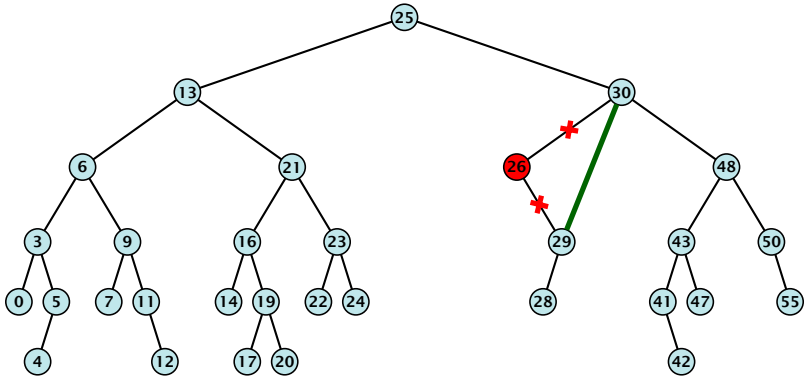


Case 1:

Element does not have any children

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

Binary Search Trees: Delete

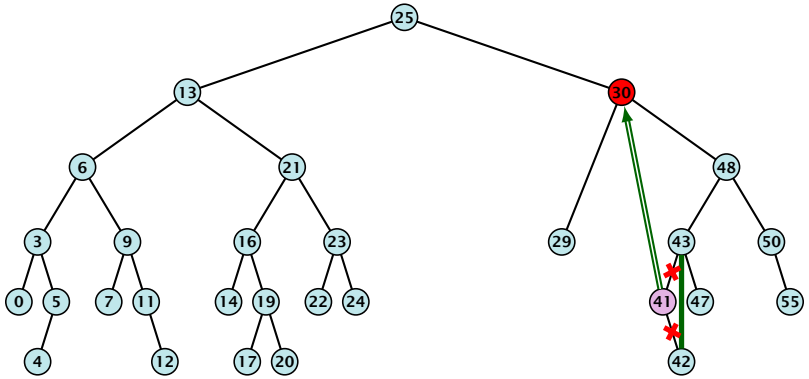


Case 2:

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 ← z else y ← TreeSucc(z);        select y to splice out
 3: if left[y] ≠ null
 4:       then x ← left[y] else x ← right[y]; x is child of y (or null)
 5: if x ≠ null then parent[x] ← parent[y];       parent[x] is correct
 6: if parent[y] = null then                      } fix pointer to x
 7:       root[T] ← x                             }
 8: else                                          }
 9:       if y = left[parent[x]] then             }
10:            left[parent[y]] ← x                }
11:       else                                    }
12:            right[parent[y]] ← x               }
13: if y ≠ 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.