6.5 Transformation of the Recurrence

Example 10

$$f_0 = 1$$
$$f_1 = 2$$
$$f_n = f_{n-1} \cdot f_{n-2} \text{ for } n \geq 2 .$$

6.5 Transformation of the Recurrence

Example 10

$$f_0 = 1$$
$$f_1 = 2$$
$$f_n = f_{n-1} \cdot f_{n-2} \text{ for } n \geq 2 \,.$$

Define

$$g_n := \log f_n \,.$$

6.5 Transformation of the Recurrence

Example 10

$$f_0 = 1$$
$$f_1 = 2$$
$$f_n = f_{n-1} \cdot f_{n-2} \text{ for } n \geq 2 \, .$$

$$g_n = \log f_n = \log(f_{n-1} \cdot f_{n-2}) = (\log f_{n-1}) + \log(f_{n-2})$$

$$= g_{n-1} + g_{n-2}$$

Define

$$g_n := \log f_n \, .$$

Then

$$g_n = g_{n-1} + g_{n-2} \text{ for } n \geq 2$$

6.5 Transformation of the Recurrence

Example 10

$$f_0 = 1$$
$$f_1 = 2$$
$$f_n = f_{n-1} \cdot f_{n-2} \text{ for } n \geq 2 \,.$$

Define

$$g_n := \log f_n \,.$$

Then

$$g_n = g_{n-1} + g_{n-2} \text{ for } n \geq 2$$
$$g_1 = \log 2 = 1 (\text{for } \log = \log_2),\; g_0 = 0$$

6.5 Transformation of the Recurrence

Example 10

$$f_0 = 1$$
$$f_1 = 2$$
$$f_n = f_{n-1} \cdot f_{n-2} \text{ for } n \geq 2 \text{ .}$$

Define

$$g_n := \log f_n \text{ .}$$

Then

$$g_n = g_{n-1} + g_{n-2} \text{ for } n \geq 2$$
$$g_1 = \log 2 = 1 (\text{for } \log = \log_2),\ g_0 = 0$$
$$g_n = F_n \text{ (} n\text{-th Fibonacci number)}$$

6.5 Transformation of the Recurrence

Example 10

$$f_0 = 1$$
$$f_1 = 2$$
$$f_n = f_{n-1} \cdot f_{n-2} \text{ for } n \geq 2 \, .$$

Define

$$g_n := \log f_n \, .$$

Then

$$g_n = g_{n-1} + g_{n-2} \text{ for } n \geq 2$$
$$g_1 = \log 2 = 1 (\text{for } \log = \log_2), \; g_0 = 0$$
$$g_n = F_n \; (n\text{-th Fibonacci number})$$
$$f_n = 2^{F_n}$$

6.5 Transformation of the Recurrence

Example 11

$$f_1 = 1$$
$$f_n = 3f_{\frac{n}{2}} + n; \text{ for } n = 2^k, k \geq 1 \;;$$

6.5 Transformation of the Recurrence

Example 11

$$f_1 = 1$$
$$f_n = 3f_{\frac{n}{2}} + n;\ \text{for } n = 2^k,\ k \geq 1\ ;$$

Define

$$g_k := f_{2^k}\ .$$

6.5 Transformation of the Recurrence

Example 11

$$f_1 = 1$$
$$f_n = 3f_{\frac{n}{2}} + n;\ \text{for } n = 2^k,\ k \geq 1\ ;$$

Define

$$g_k := f_{2^k}\ .$$

Then:

$$g_0 = 1$$

6.5 Transformation of the Recurrence

Example 11

$$f_1 = 1$$
$$f_n = 3f_{\frac{n}{2}} + n; \text{ for } n = 2^k,\ k \geq 1\ ;$$

$$g_n = f_{2^k} = 3f_{2^{k-1}} + 2^k = 3g_{k-1} + 2^k$$

Define

$$g_k := f_{2^k}\ .$$

Then:

$$g_0 = 1$$
$$g_k = 3g_{k-1} + 2^k,\ k \geq 1$$

6 Recurrences

We get

$$g_k = 3\left[g_{k-1}\right] + 2^k$$

6 Recurrences

We get

$$\begin{aligned} g_k &= 3\left[g_{k-1}\right] + 2^k \\ &= 3\left[3g_{k-2} + 2^{k-1}\right] + 2^k \end{aligned}$$

6 Recurrences

We get

$$\begin{aligned} g_k &= 3\left[g_{k-1}\right] + 2^k \\ &= 3\left[3g_{k-2} + 2^{k-1}\right] + 2^k \\ &= 3^2\left[g_{k-2}\right] + 32^{k-1} + 2^k \end{aligned}$$

6 Recurrences

We get

$$\begin{aligned} g_k &= 3\left[g_{k-1}\right] + 2^k \\ &= 3\left[3g_{k-2} + 2^{k-1}\right] + 2^k \\ &= 3^2\left[g_{k-2}\right] + 32^{k-1} + 2^k \\ &= 3^2\left[3g_{k-3} + 2^{k-2}\right] + 32^{k-1} + 2^k \end{aligned}$$

6 Recurrences

We get

$$
\begin{aligned}
g_k &= 3\left[g_{k-1}\right] + 2^k \\
&= 3\left[3g_{k-2} + 2^{k-1}\right] + 2^k \\
&= 3^2\left[g_{k-2}\right] + 32^{k-1} + 2^k \\
&= 3^2\left[3g_{k-3} + 2^{k-2}\right] + 32^{k-1} + 2^k \\
&= 3^3 g_{k-3} + 3^2 2^{k-2} + 32^{k-1} + 2^k
\end{aligned}
$$

6 Recurrences

We get $3^k + 3^{k-1} \cdot 2 + 3^{k-2} \cdot 2^2 + \ldots + 3 \cdot 2^{k-1} + 2^k$

$$\begin{aligned} g_k &= 3\left[g_{k-1}\right] + 2^k \\ &= 3\left[3g_{k-2} + 2^{k-1}\right] + 2^k \\ &= 3^2\left[g_{k-2}\right] + 32^{k-1} + 2^k \\ &= 3^2\left[3g_{k-3} + 2^{k-2}\right] + 32^{k-1} + 2^k \\ &= 3^3 g_{k-3} + 3^2 2^{k-2} + 32^{k-1} + 2^k \\ &= 2^k \cdot \sum_{i=0}^{k} \left(\frac{3}{2}\right)^i \end{aligned}$$

6 Recurrences

We get

$$
\begin{aligned}
g_k &= 3\left[g_{k-1}\right] + 2^k \\
&= 3\left[3g_{k-2} + 2^{k-1}\right] + 2^k \\
&= 3^2\left[g_{k-2}\right] + 32^{k-1} + 2^k \\
&= 3^2\left[3g_{k-3} + 2^{k-2}\right] + 32^{k-1} + 2^k \\
&= 3^3 g_{k-3} + 3^2 2^{k-2} + 32^{k-1} + 2^k \\
&= 2^k \cdot \sum_{i=0}^{k} \left(\frac{3}{2}\right)^i \\
&= 2^k \cdot \frac{(\frac{3}{2})^{k+1} - 1}{1/2}
\end{aligned}
$$

6 Recurrences

We get

$$
\begin{aligned}
g_k &= 3\left[g_{k-1}\right] + 2^k \\
&= 3\left[3g_{k-2} + 2^{k-1}\right] + 2^k \\
&= 3^2\left[g_{k-2}\right] + 32^{k-1} + 2^k \\
&= 3^2\left[3g_{k-3} + 2^{k-2}\right] + 32^{k-1} + 2^k \\
&= 3^3 g_{k-3} + 3^2 2^{k-2} + 32^{k-1} + 2^k \\
&= 2^k \cdot \sum_{i=0}^{k} \left(\frac{3}{2}\right)^i \\
&= 2^k \cdot \frac{(\frac{3}{2})^{k+1} - 1}{1/2} = 3^{k+1} - 2^{k+1}
\end{aligned}
$$

6 Recurrences

Let $n = 2^k$:

$$g_k = 3^{k+1} - 2^{k+1}, \text{ hence}$$
$$f_n = 3 \cdot 3^k - 2 \cdot 2^k$$

6 Recurrences

Let $n = 2^k$:

$$
\begin{aligned}
g_k &= 3^{k+1} - 2^{k+1}, \text{ hence} \\
f_n &= 3 \cdot 3^k - 2 \cdot 2^k \\
&= 3(2^{\log 3})^k - 2 \cdot 2^k
\end{aligned}
$$

6 Recurrences

Let $n = 2^k$:

$$
\begin{aligned}
g_k &= 3^{k+1} - 2^{k+1}, \text{ hence} \\
f_n &= 3 \cdot 3^k - 2 \cdot 2^k \\
&= 3(2^{\log 3})^k - 2 \cdot 2^k \\
&= 3(2^k)^{\log 3} - 2 \cdot 2^k
\end{aligned}
$$

6 Recurrences

Let $n = 2^k$:

$$
\begin{aligned}
g_k &= 3^{k+1} - 2^{k+1}, \text{ hence} \\
f_n &= 3 \cdot 3^k - 2 \cdot 2^k \\
&= 3(2^{\log 3})^k - 2 \cdot 2^k \\
&= 3(2^k)^{\log 3} - 2 \cdot 2^k \\
&= 3n^{\log 3} - 2n \ .
\end{aligned}
$$

$\Theta(n^{\log 3})$

Part III

Data Structures

Abstract Data Type

An abstract data type (ADT) is defined by an interface of operations or methods that can be performed and that have a defined behavior.

The data types in this lecture all operate on objects that are represented by a [key, value] pair.

- The key comes from a totally ordered set, and we assume that there is an efficient comparison function.
- The value can be anything; it usually carries satellite information important for the application that uses the ADT.

Dynamic Set Operations

- $S.\mathbf{search}(k)$: Returns pointer to object x from S with $\mathrm{key}[x] = k$ or null.
- $S.\mathbf{insert}(x)$: Inserts object x into set S. $\mathrm{key}[x]$ must not currently exist in the data-structure.
- $S.\mathbf{delete}(x)$: Given pointer to object x from S, delete x from the set.
- $S.\mathbf{minimum}()$: Return pointer to object with smallest key-value in S.
- $S.\mathbf{maximum}()$: Return pointer to object with largest key-value in S.
- $S.\mathbf{successor}(x)$: Return pointer to the next larger element in S or null if x is maximum.
- $S.\mathbf{predecessor}(x)$: Return pointer to the next smaller element in S or null if x is minimum.

Dynamic Set Operations

- $S.\mathbf{search}(k)$: Returns pointer to object x from S with $\text{key}[x] = k$ or null.
- $S.\mathbf{insert}(x)$: Inserts object x into set S. $\text{key}[x]$ must not currently exist in the data-structure.
- $S.\mathbf{delete}(x)$: Given pointer to object x from S, delete x from the set.
- $S.\mathbf{minimum}()$: Return pointer to object with smallest key-value in S.
- $S.\mathbf{maximum}()$: Return pointer to object with largest key-value in S.
- $S.\mathbf{successor}(x)$: Return pointer to the next larger element in S or null if x is maximum.
- $S.\mathbf{predecessor}(x)$: Return pointer to the next smaller element in S or null if x is minimum.

Dynamic Set Operations

- $\boldsymbol{S.\,\mathrm{search}(k)}$: Returns pointer to object x from S with $\mathrm{key}[x] = k$ or null.
- $\boldsymbol{S.\,\mathrm{insert}(x)}$: Inserts object x into set S. $\mathrm{key}[x]$ must not currently exist in the data-structure.
- $\boldsymbol{S.\,\mathrm{delete}(x)}$: Given pointer to object x from S, delete x from the set.
- $\boldsymbol{S.\,\mathrm{minimum}()}$: Return pointer to object with smallest key-value in S.
- $\boldsymbol{S.\,\mathrm{maximum}()}$: Return pointer to object with largest key-value in S.
- $\boldsymbol{S.\,\mathrm{successor}(x)}$: Return pointer to the next larger element in S or null if x is maximum.
- $\boldsymbol{S.\,\mathrm{predecessor}(x)}$: Return pointer to the next smaller element in S or null if x is minimum.

Dynamic Set Operations

- $S.\mathbf{search}(k)$: Returns pointer to object x from S with $\mathrm{key}[x] = k$ or null.
- $S.\mathbf{insert}(x)$: Inserts object x into set S. $\mathrm{key}[x]$ must not currently exist in the data-structure.
- $S.\mathbf{delete}(x)$: Given pointer to object x from S, delete x from the set.
- $S.\mathbf{minimum}()$: Return pointer to object with smallest key-value in S.
- $S.\mathbf{maximum}()$: Return pointer to object with largest key-value in S.
- $S.\mathbf{successor}(x)$: Return pointer to the next larger element in S or null if x is maximum.
- $S.\mathbf{predecessor}(x)$: Return pointer to the next smaller element in S or null if x is minimum.

Dynamic Set Operations

- $\boldsymbol{S}.\,\mathbf{search}(\boldsymbol{k})$: Returns pointer to object x from S with $\mathrm{key}[x] = k$ or null.
- $\boldsymbol{S}.\,\mathbf{insert}(\boldsymbol{x})$: Inserts object x into set S. $\mathrm{key}[x]$ must not currently exist in the data-structure.
- $\boldsymbol{S}.\,\mathbf{delete}(\boldsymbol{x})$: Given pointer to object x from S, delete x from the set.
- $\boldsymbol{S}.\,\mathbf{minimum}()$: Return pointer to object with smallest key-value in S.
- $\boldsymbol{S}.\,\mathbf{maximum}()$: Return pointer to object with largest key-value in S.
- $\boldsymbol{S}.\,\mathbf{successor}(\boldsymbol{x})$: Return pointer to the next larger element in S or null if x is maximum.
- $\boldsymbol{S}.\,\mathbf{predecessor}(\boldsymbol{x})$: Return pointer to the next smaller element in S or null if x is minimum.

Dynamic Set Operations

- $\boldsymbol{S.\,\mathrm{search}(k)}$: Returns pointer to object x from S with $\mathrm{key}[x] = k$ or null.
- $\boldsymbol{S.\,\mathrm{insert}(x)}$: Inserts object x into set S. $\mathrm{key}[x]$ must not currently exist in the data-structure.
- $\boldsymbol{S.\,\mathrm{delete}(x)}$: Given pointer to object x from S, delete x from the set.
- $\boldsymbol{S.\,\mathrm{minimum}()}$: Return pointer to object with smallest key-value in S.
- $\boldsymbol{S.\,\mathrm{maximum}()}$: Return pointer to object with largest key-value in S.
- $\boldsymbol{S.\,\mathrm{successor}(x)}$: Return pointer to the next larger element in S or null if x is maximum.
- $\boldsymbol{S.\,\mathrm{predecessor}(x)}$: Return pointer to the next smaller element in S or null if x is minimum.

Dynamic Set Operations

- $\boldsymbol{S.\,\text{search}(k)}$: Returns pointer to object x from S with $\text{key}[x] = k$ or null.
- $\boldsymbol{S.\,\text{insert}(x)}$: Inserts object x into set S. $\text{key}[x]$ must not currently exist in the data-structure.
- $\boldsymbol{S.\,\text{delete}(x)}$: Given pointer to object x from S, delete x from the set.
- $\boldsymbol{S.\,\text{minimum}()}$: Return pointer to object with smallest key-value in S.
- $\boldsymbol{S.\,\text{maximum}()}$: Return pointer to object with largest key-value in S.
- $\boldsymbol{S.\,\text{successor}(x)}$: Return pointer to the next larger element in S or null if x is maximum.
- $\boldsymbol{S.\,\text{predecessor}(x)}$: Return pointer to the next smaller element in S or null if x is minimum.

Dynamic Set Operations

- $S.\textbf{union}(S')$: Sets $S := S \cup S'$. The set S' is destroyed.
- $S.\textbf{merge}(S')$: Sets $S := S \cup S'$. Requires $S \cap S' = \emptyset$.
- $S.\textbf{split}(k, S')$:
  $S := \{x \in S \mid \text{key}[x] \leq k\}$, $S' := \{x \in S \mid \text{key}[x] > k\}$.
- $S.\textbf{concatenate}(S')$: $S := S \cup S'$.
  Requires $\text{key}[S.\text{maximum}()] \leq \text{key}[S'.\text{minimum}()]$.
- $S.\textbf{decrease-key}(x, k)$: Replace $\text{key}[x]$ by $k \leq \text{key}[x]$.

Dynamic Set Operations

- **S. union(S')**: Sets $S := S \cup S'$. The set S' is destroyed.
- **S. merge(S')**: Sets $S := S \cup S'$. Requires $S \cap S' = \emptyset$.
- **S. split(k, S')**:
  $S := \{x \in S \mid \text{key}[x] \leq k\}$, $S' := \{x \in S \mid \text{key}[x] > k\}$.
- **S. concatenate(S')**: $S := S \cup S'$.
  Requires $\text{key}[S.\text{maximum}()] \leq \text{key}[S'.\text{minimum}()]$.
- **S. decrease-key(x, k)**: Replace $\text{key}[x]$ by $k \leq \text{key}[x]$.

Dynamic Set Operations

- $\boldsymbol{S.}\,\textbf{union}(\boldsymbol{S'})$: Sets $S := S \cup S'$. The set S' is destroyed.
- $\boldsymbol{S.}\,\textbf{merge}(\boldsymbol{S'})$: Sets $S := S \cup S'$. Requires $S \cap S' = \emptyset$.
- $\boldsymbol{S.}\,\textbf{split}(\boldsymbol{k}, \boldsymbol{S'})$:
  $S := \{x \in S \mid \text{key}[x] \le k\}$, $S' := \{x \in S \mid \text{key}[x] > k\}$.
- $\boldsymbol{S.}\,\textbf{concatenate}(\boldsymbol{S'})$: $S := S \cup S'$.
  Requires $\text{key}[S.\text{maximum}()] \le \text{key}[S'.\text{minimum}()]$.
- $\boldsymbol{S.}\,\textbf{decrease-key}(\boldsymbol{x}, \boldsymbol{k})$: Replace $\text{key}[x]$ by $k \le \text{key}[x]$.

Dynamic Set Operations

- $S.\text{union}(S')$: Sets $S := S \cup S'$. The set S' is destroyed.
- $S.\text{merge}(S')$: Sets $S := S \cup S'$. Requires $S \cap S' = \emptyset$.
- $S.\text{split}(k, S')$:
  $S := \{x \in S \mid \text{key}[x] \leq k\}$, $S' := \{x \in S \mid \text{key}[x] > k\}$.
- $S.\text{concatenate}(S')$: $S := S \cup S'$.
  Requires $\text{key}[S.\text{maximum}()] \leq \text{key}[S'.\text{minimum}()]$.
- $S.\text{decrease-key}(x, k)$: Replace $\text{key}[x]$ by $k \leq \text{key}[x]$.

Dynamic Set Operations

- $S.\mathbf{union}(S')$: Sets $S := S \cup S'$. The set S' is destroyed.
- $S.\mathbf{merge}(S')$: Sets $S := S \cup S'$. Requires $S \cap S' = \emptyset$.
- $S.\mathbf{split}(k, S')$:
  $S := \{x \in S \mid \text{key}[x] \le k\}$, $S' := \{x \in S \mid \text{key}[x] > k\}$.
- $S.\mathbf{concatenate}(S')$: $S := S \cup S'$.
  Requires $\text{key}[S.\text{maximum}()] \le \text{key}[S'.\text{minimum}()]$.

- $S.\mathbf{decrease\text{-}key}(x, k)$: Replace $\text{key}[x]$ by $k \le \text{key}[x]$.

S. change-key (x, k) . " " " k

Examples of ADTs

Stack:

- $S.\mathbf{push}(x)$: Insert an element.
- $S.\mathbf{pop}()$: Return the element from S that was inserted most recently; delete it from S.
- $S.\mathbf{empty}()$: Tell if S contains any object.

Queue:

- $S.\mathbf{enqueue}(x)$: Insert an element.
- $S.\mathbf{dequeue}()$: Return the element that is longest in the structure; delete it from S.
- $S.\mathbf{empty}()$: Tell if S contains any object.

Priority-Queue:

- $S.\mathbf{insert}(x)$: Insert an element.
- $S.\mathbf{delete\text{-}min}()$: Return the element with lowest key-value; delete it from S.

Examples of ADTs

Stack:

- $S.\mathrm{push}(x)$: Insert an element.
- $S.\mathrm{pop}()$: Return the element from S that was inserted most recently; delete it from S.
- $S.\mathrm{empty}()$: Tell if S contains any object.

Queue:

- $S.\mathrm{enqueue}(x)$: Insert an element.
- $S.\mathrm{dequeue}()$: Return the element that is longest in the structure; delete it from S.
- $S.\mathrm{empty}()$: Tell if S contains any object.

Priority-Queue:

- $S.\mathrm{insert}(x)$: Insert an element.
- $S.\mathrm{delete\text{-}min}()$: Return the element with lowest key-value; delete it from S.

Examples of ADTs

Stack:

- $S.\textbf{push}(x)$: Insert an element.
- $S.\textbf{pop}()$: Return the element from S that was inserted most recently; delete it from S.
- $S.\textbf{empty}()$: Tell if S contains any object.

Queue:

- $S.\textbf{enqueue}(x)$: Insert an element.
- $S.\textbf{dequeue}()$: Return the element that is longest in the structure; delete it from S.
- $S.\textbf{empty}()$: Tell if S contains any object.

Priority-Queue:

- $S.\textbf{insert}(x)$: Insert an element.
- $S.\textbf{delete-min}()$: Return the element with lowest key-value; delete it from S.

7 Dictionary

Dictionary:

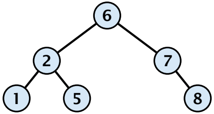
- $\boldsymbol{S.\,\mathrm{insert}(x)}$: Insert an element x.
- $\boldsymbol{S.\,\mathrm{delete}(x)}$: Delete the element pointed to by x.
- $\boldsymbol{S.\,\mathrm{search}(k)}$: Return a pointer to an element e with $\mathrm{key}[e] = k$ in S if it exists; otherwise return null.

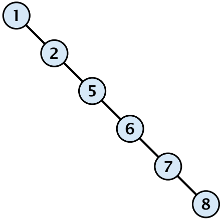
7.1 Binary Search Trees

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:



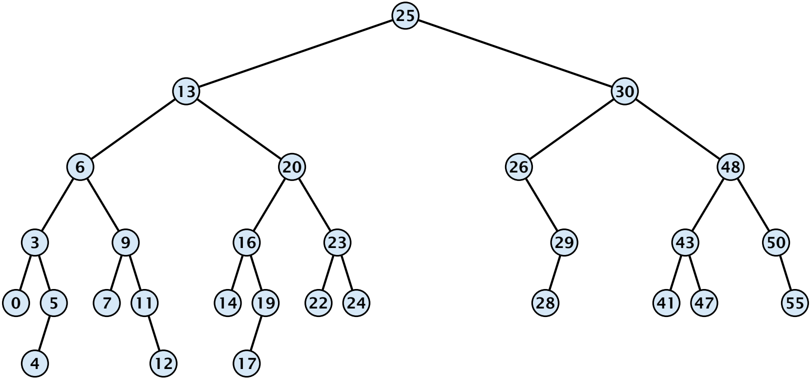


7.1 Binary Search Trees

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



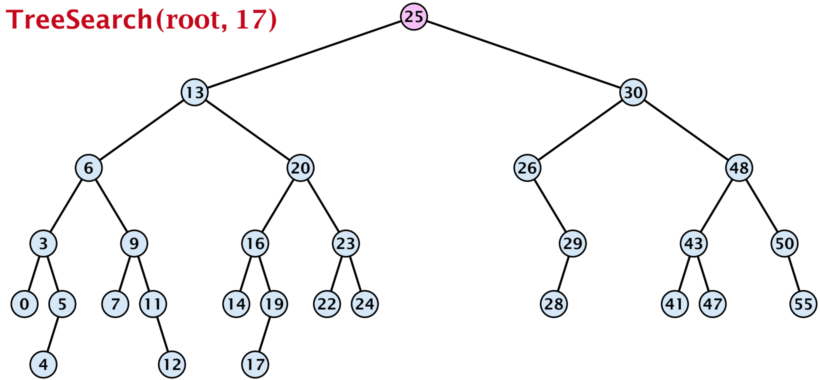
Algorithm 5 TreeSearch(x, k)

if $x = \text{null}$ **or** $k = \text{key}[x]$ **return** x

if $k < \text{key}[x]$ **return** TreeSearch$(\text{left}[x], k)$

else return TreeSearch$(\text{right}[x], k)$

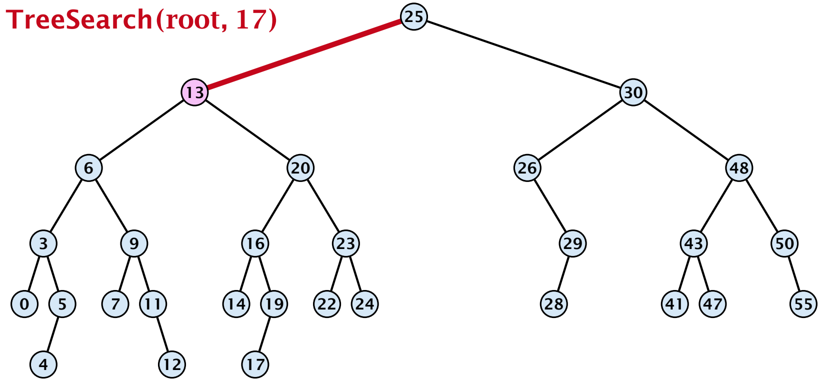
Binary Search Trees: Searching



Algorithm 5 TreeSearch(x, k)

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

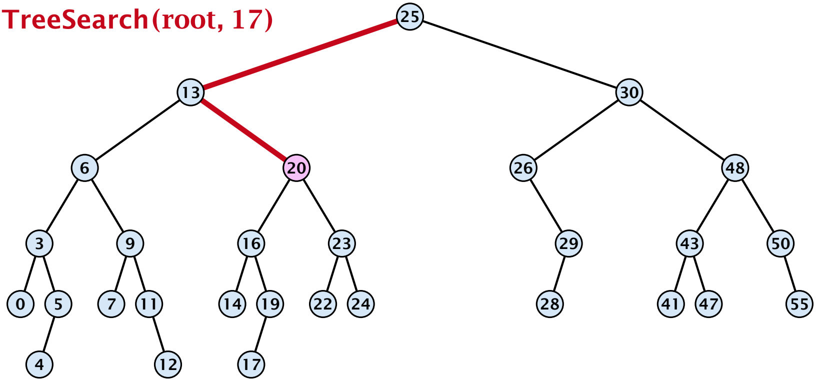
Binary Search Trees: Searching



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



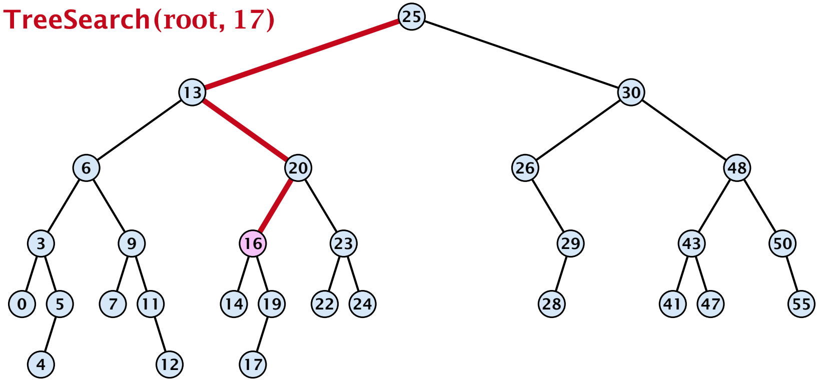
Algorithm 5 TreeSearch(x, k)

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$(\text{right}[x], k)$

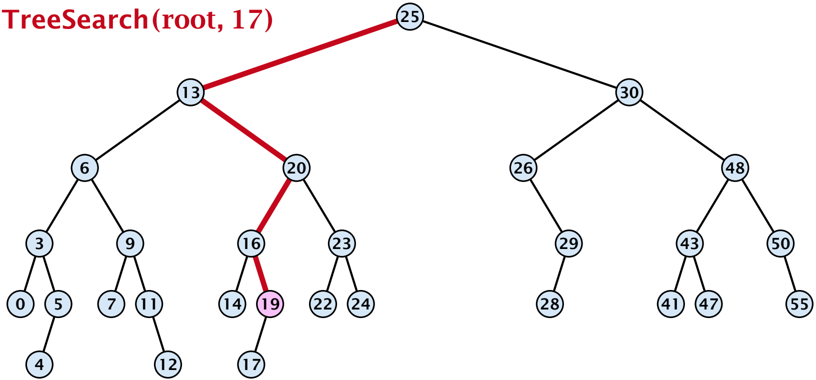
Binary Search Trees: Searching



Algorithm 5 TreeSearch(x, k)

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

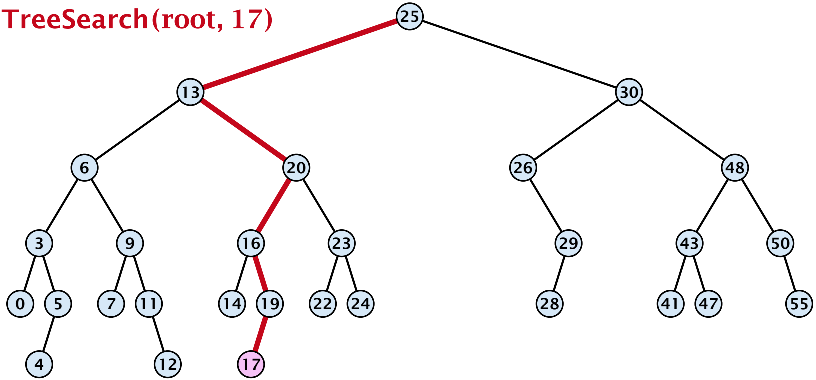
Binary Search Trees: Searching



Algorithm 5 TreeSearch(x, k)

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

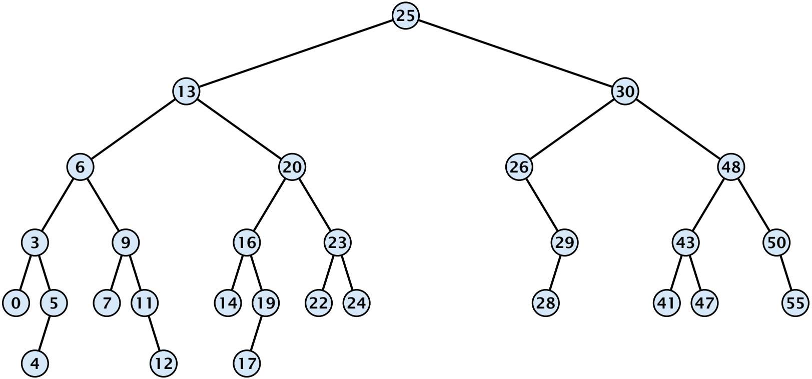
Binary Search Trees: Searching



Algorithm 5 TreeSearch(x, k)

1: **if** x = 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

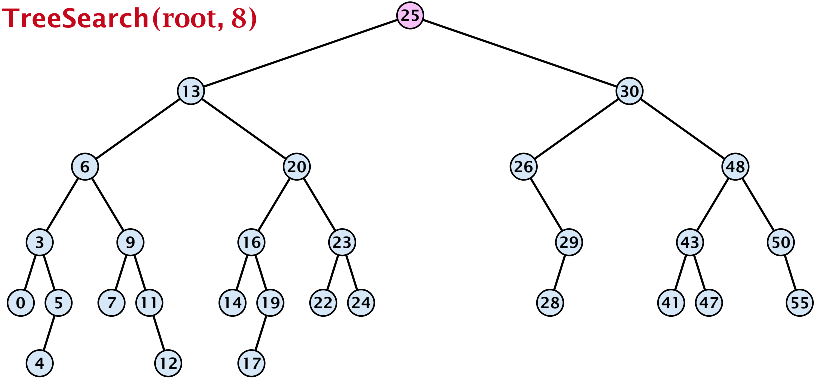


Algorithm 5 TreeSearch(x, k)

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

Binary Search Trees: Searching

TreeSearch(root, 8)

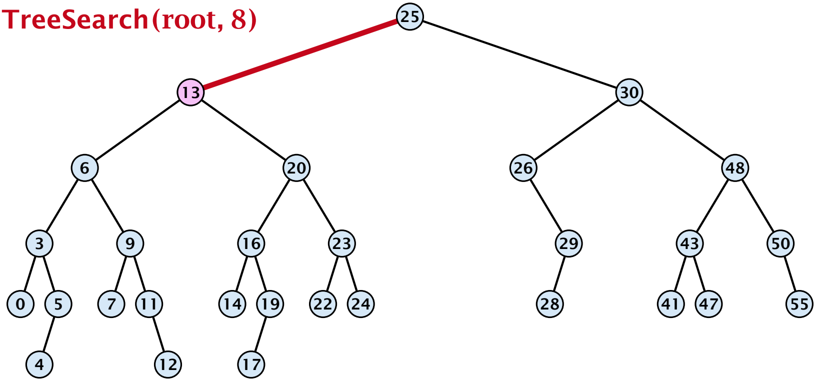


Algorithm 5 TreeSearch(x, k)

1: **if** x = 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

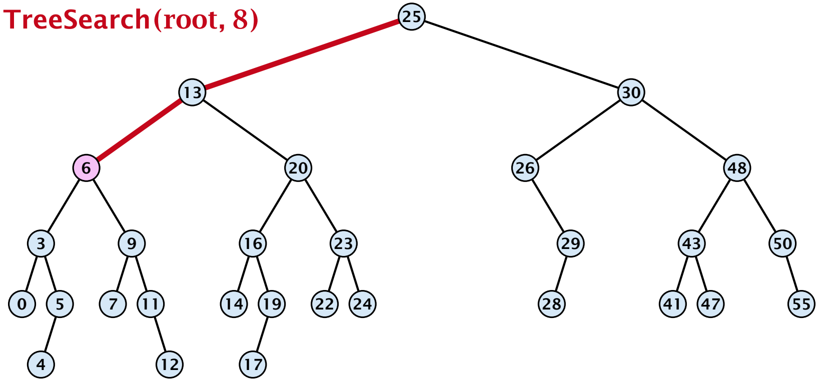
TreeSearch(root, 8)



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: Searching

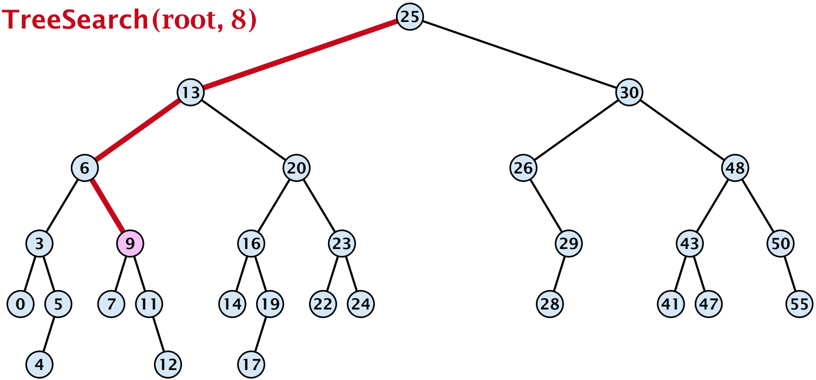


Algorithm 5 TreeSearch(x, k)

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

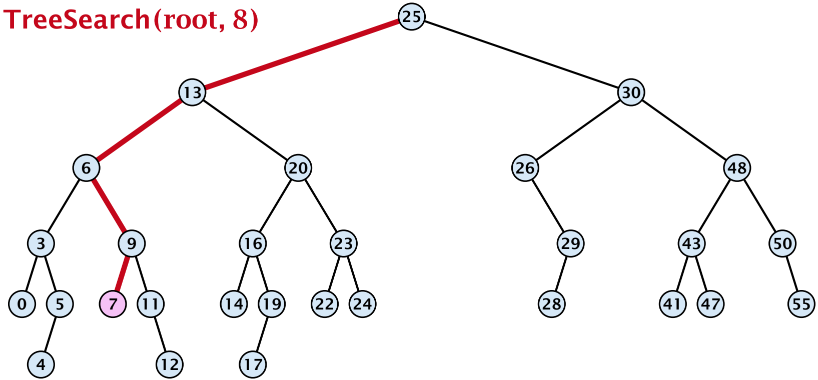
TreeSearch(root, 8)



Algorithm 5 TreeSearch(x, k)

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

Binary Search Trees: Searching

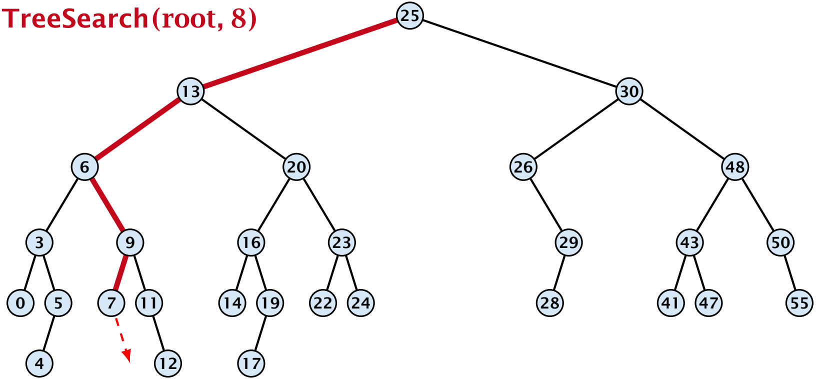


Algorithm 5 TreeSearch(x, k)

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

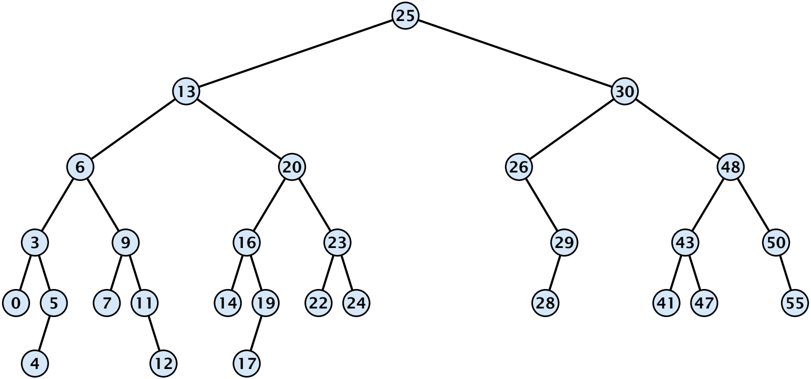
TreeSearch(root, 8)



Algorithm 5 TreeSearch(x, k)

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

Binary Search Trees: Minimum

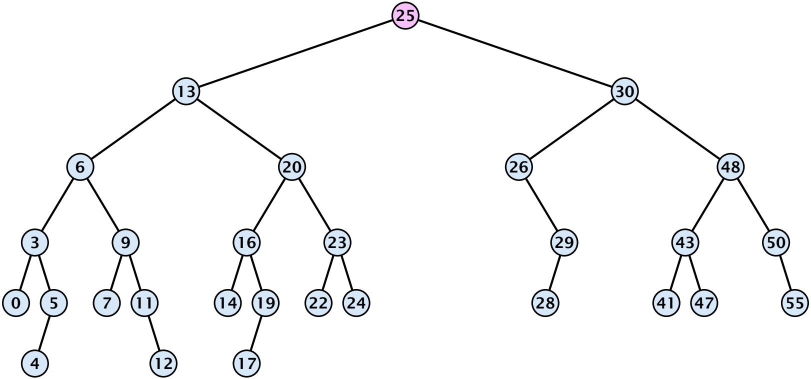


Algorithm 6 TreeMin(x)

1: **if** $x = \text{null}$ **or** $\text{left}[x] = \text{null}$ **return** x

2: **return** TreeMin($\text{left}[x]$)

Binary Search Trees: Minimum

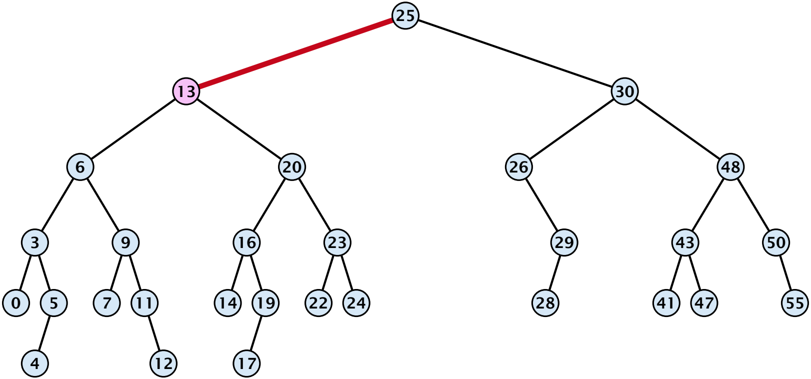


Algorithm 6 TreeMin(x)

1: **if** x = null **or** left[x] = null **return** x

2: **return** TreeMin(left[x])

Binary Search Trees: Minimum

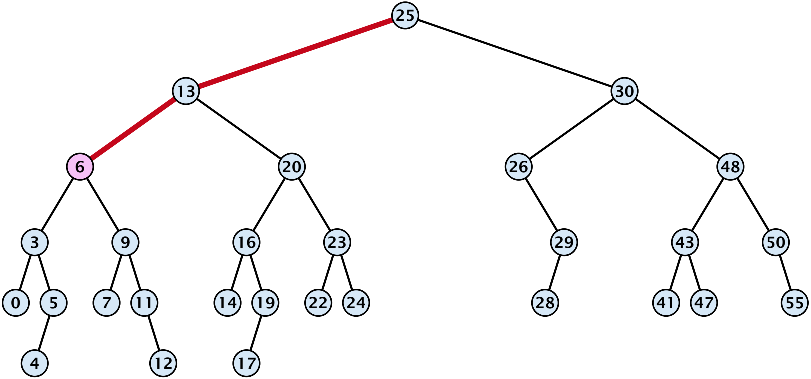


Algorithm 6 TreeMin(x)

1: **if** $x = \text{null}$ **or** $\text{left}[x] = \text{null}$ **return** x

2: **return** TreeMin($\text{left}[x]$)

Binary Search Trees: Minimum

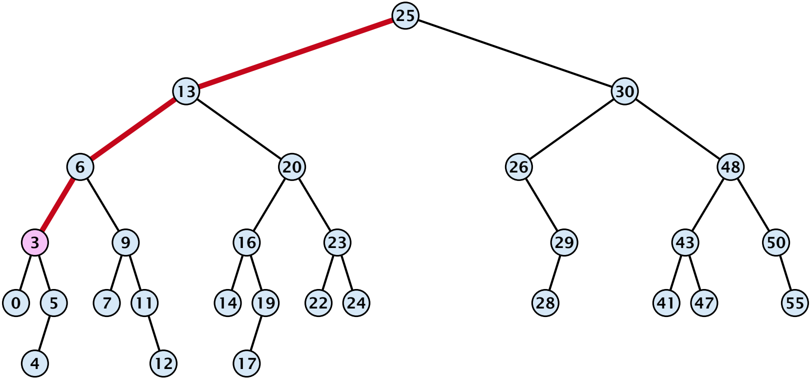


Algorithm 6 TreeMin(x)

1: **if** $x = \text{null}$ **or** $\text{left}[x] = \text{null}$ **return** x

2: **return** TreeMin($\text{left}[x]$)

Binary Search Trees: Minimum

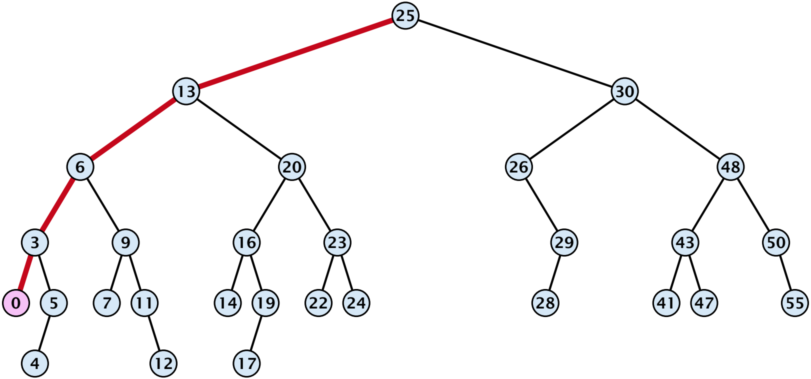


Algorithm 6 TreeMin(x)

1: **if** $x = \text{null}$ **or** $\text{left}[x] = \text{null}$ **return** x

2: **return** TreeMin($\text{left}[x]$)

Binary Search Trees: Minimum

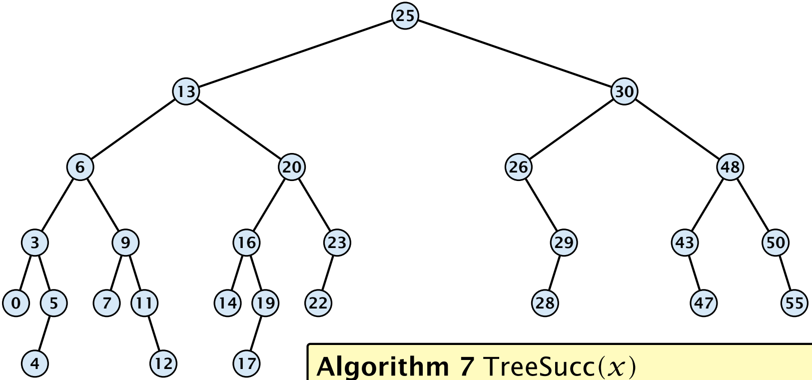


Algorithm 6 TreeMin(x)

1: **if** $x = \text{null}$ **or** $\text{left}[x] = \text{null}$ **return** x

2: **return** TreeMin($\text{left}[x]$)

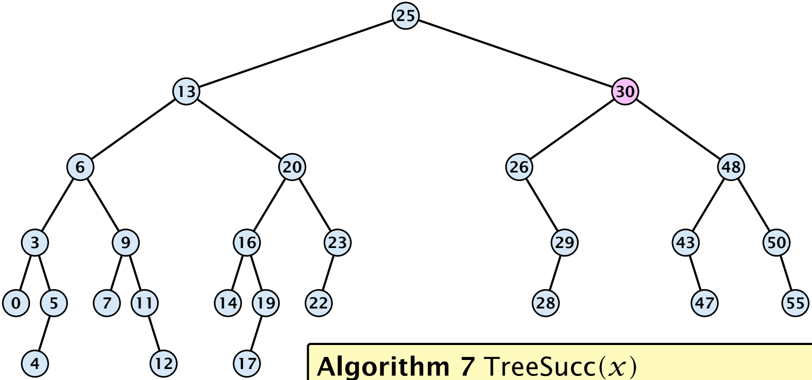
Binary Search Trees: Successor



Algorithm 7 TreeSucc(x)

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

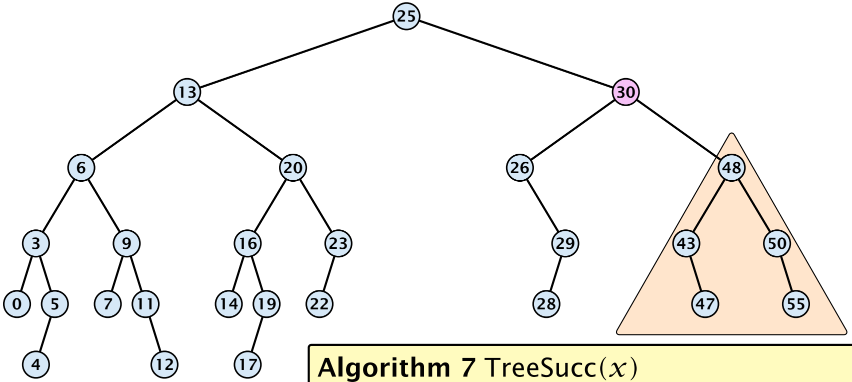
Binary Search Trees: Successor



Algorithm 7 TreeSucc(x)

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

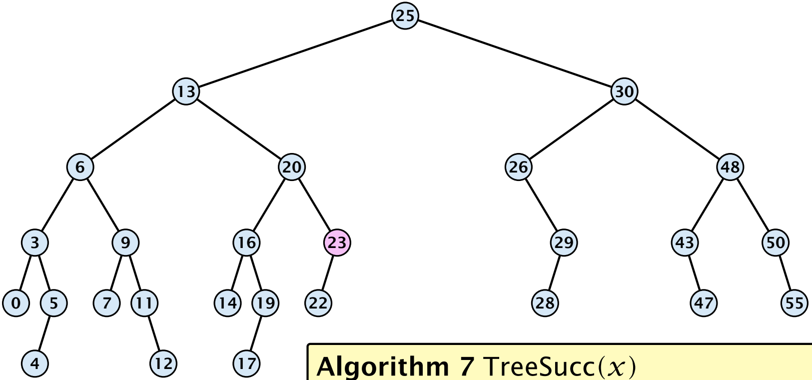
Binary Search Trees: Successor



Algorithm 7 TreeSucc(x)

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

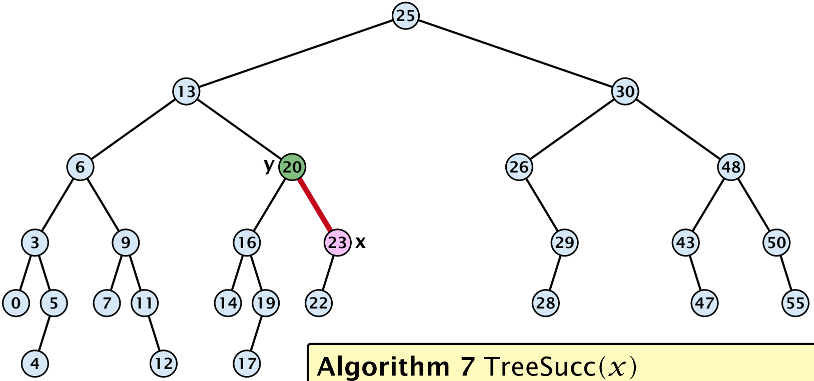
Binary Search Trees: Successor



Algorithm 7 TreeSucc(x)

if right[x] $\neq$ null **return** TreeMin(right[x])
$y \leftarrow$ parent[x]
while $y \neq$ null **and** $x =$ right[y] **do**
    $x \leftarrow y$; $y \leftarrow$ parent[x]
return y;

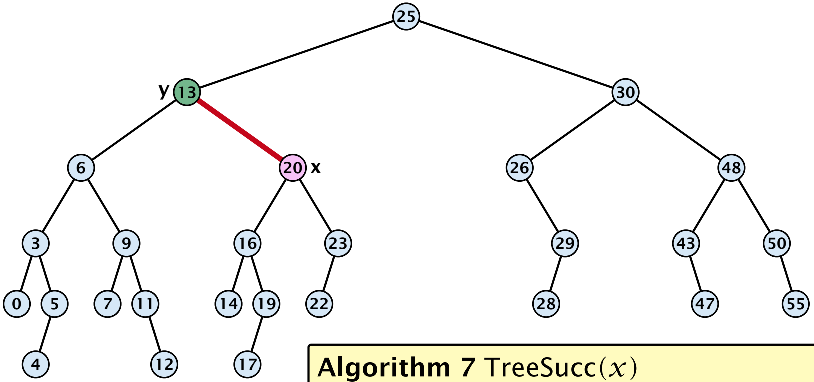
Binary Search Trees: Successor



Algorithm 7 TreeSucc(x)

```
if right[x] ≠ null return TreeMin(right[x])
y ← parent[x]
while y ≠ null and x = right[y] do
    x ← y; y ← parent[x]
return y;
```

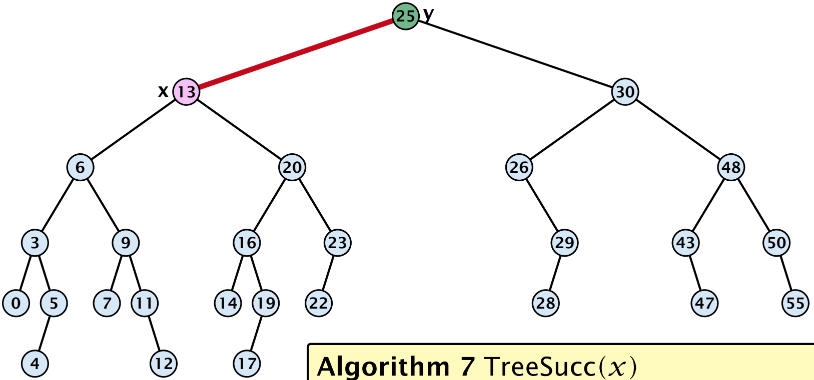
Binary Search Trees: Successor



Algorithm 7 TreeSucc(x)

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

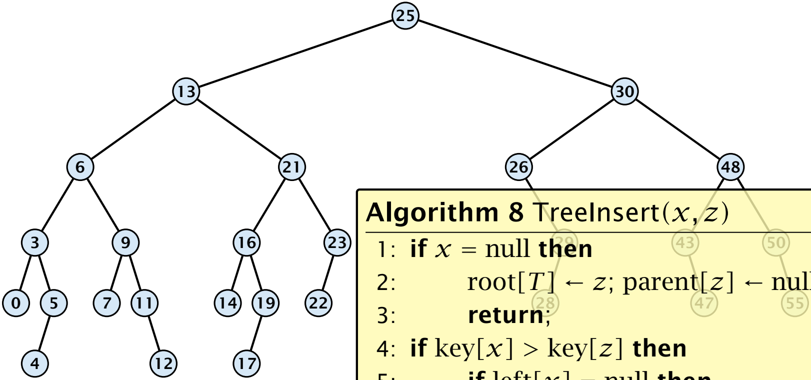
Binary Search Trees: Successor



Algorithm 7 TreeSucc(x)

if right[x] $\neq$ null **return** TreeMin(right[x])
$y \leftarrow$ parent[x]
while $y \neq$ null **and** $x =$ right[y] **do**
  $x \leftarrow y$; $y \leftarrow$ parent[x]
return y;

Binary Search Trees: Insert

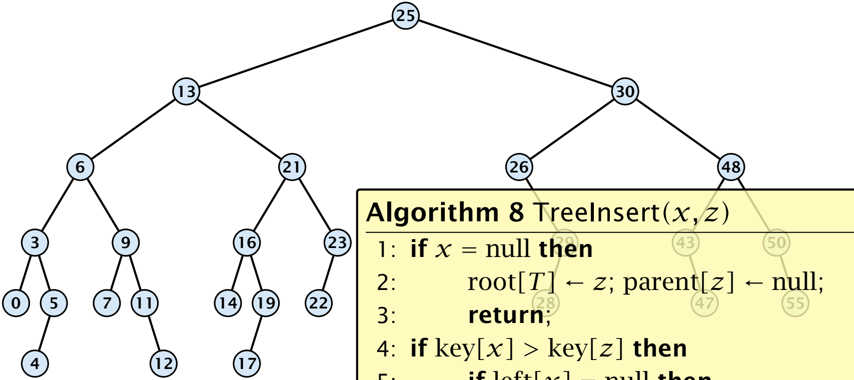


Algorithm 8 TreeInsert(x, z)

```
if x = null then
    root[T] ← z; parent[z] ← null;
    return;
if key[x] > key[z] then
    if left[x] = null then
        left[x] ← z; parent[z] ← x;
    else TreeInsert(left[x], z);
else
    if right[x] = null then
        right[x] ← z; parent[z] ← x;
    else TreeInsert(right[x], z);
```

Binary Search Trees: Insert

Insert element **not** in the tree.

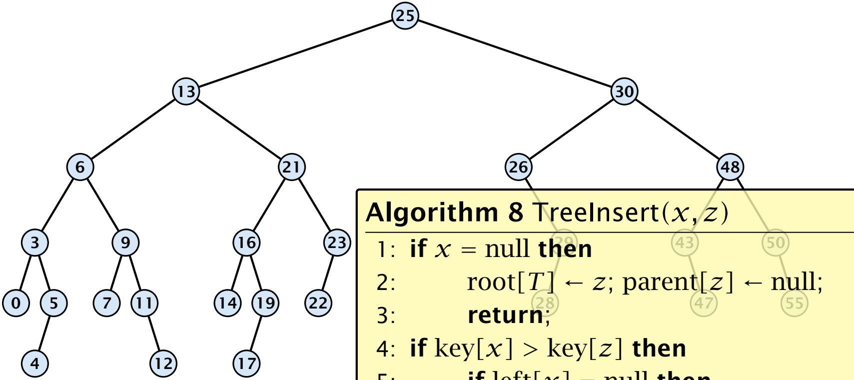


Algorithm 8 TreeInsert(x, z)

```
if x = null then
    root[T] ← z; parent[z] ← null;
    return;
if key[x] > key[z] then
    if left[x] = null then
        left[x] ← z; parent[z] ← x;
    else TreeInsert(left[x], z);
else
    if right[x] = null then
        right[x] ← z; parent[z] ← x;
    else TreeInsert(right[x], z);
```

Binary Search Trees: Insert

Insert element **not** in the tree.



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

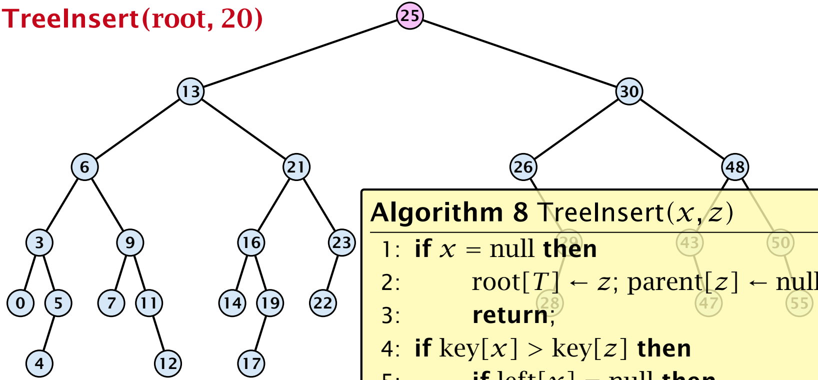
Algorithm 8 TreeInsert(x, z)

```
if x = null then
    root[T] ← z; parent[z] ← null;
    return;
if key[x] > key[z] then
    if left[x] = null then
        left[x] ← z; parent[z] ← x;
    else TreeInsert(left[x], z);
else
    if right[x] = null then
        right[x] ← z; parent[z] ← x;
    else TreeInsert(right[x], z);
```

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.

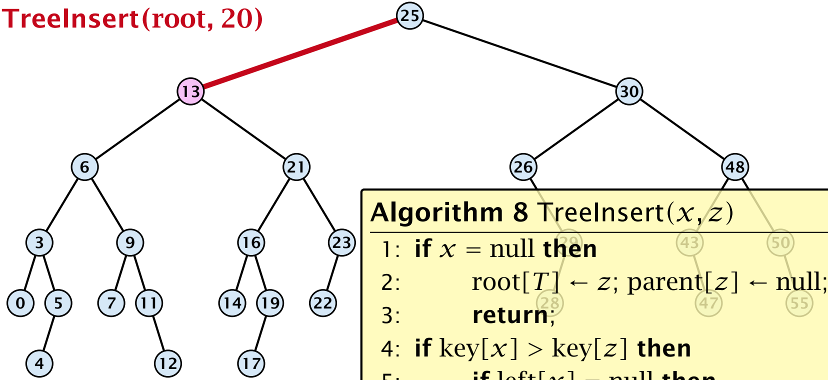
Algorithm 8 TreeInsert(x, z)

```
if x = null then
    root[T] ← z; parent[z] ← null;
    return;
if key[x] > key[z] then
    if left[x] = null then
        left[x] ← z; parent[z] ← x;
    else TreeInsert(left[x], z);
else
    if right[x] = null then
        right[x] ← z; parent[z] ← x;
    else TreeInsert(right[x], z);
```

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.

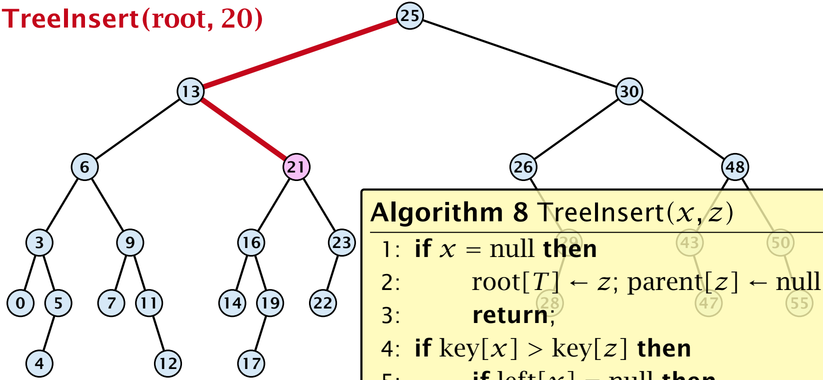
Algorithm 8 TreeInsert(x, z)

```
if x = null then
    root[T] ← z; parent[z] ← null;
    return;
if key[x] > key[z] then
    if left[x] = null then
        left[x] ← z; parent[z] ← x;
    else TreeInsert(left[x], z);
else
    if right[x] = null then
        right[x] ← z; parent[z] ← x;
    else TreeInsert(right[x], z);
```

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)

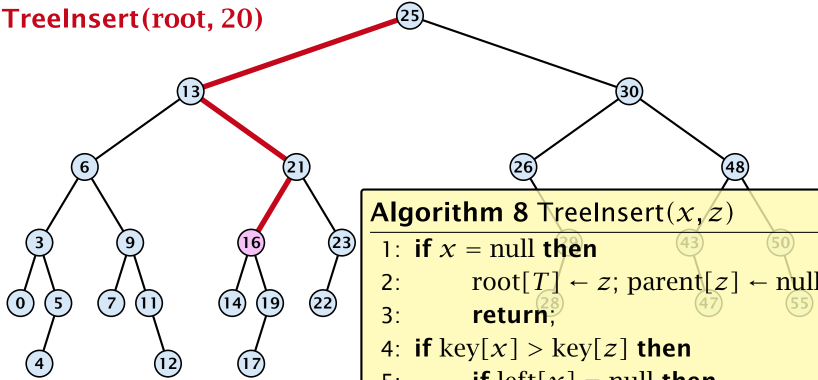
---

if x = null **then**
    root$[T] \leftarrow z$; parent$[z] \leftarrow$ null;
    return;
if key$[x] >$ key$[z]$ **then**
    if left$[x]$ = null **then**
        left$[x] \leftarrow z$; parent$[z] \leftarrow x$;
    else TreeInsert(left$[x], z$);
else
    if right$[x]$ = null **then**
        right$[x] \leftarrow z$; parent$[z] \leftarrow x$;
    else TreeInsert(right$[x], z$);

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.

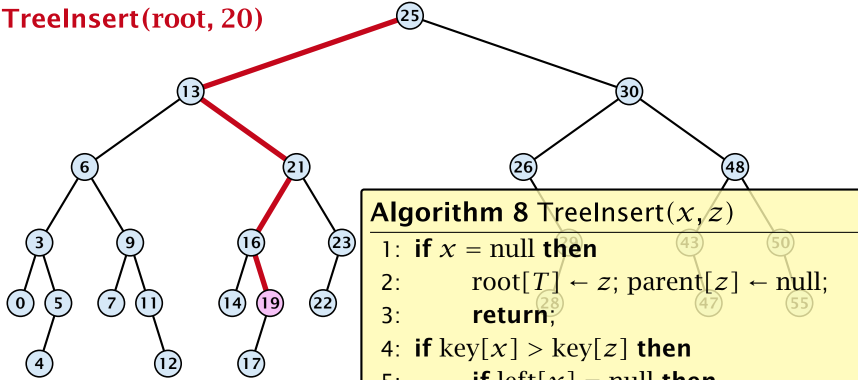
Algorithm 8 TreeInsert(x, z)

```
if x = null then
    root[T] ← z; parent[z] ← null;
    return;
if key[x] > key[z] then
    if left[x] = null then
        left[x] ← z; parent[z] ← x;
    else TreeInsert(left[x], z);
else
    if right[x] = null then
        right[x] ← z; parent[z] ← x;
    else TreeInsert(right[x], z);
```

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.

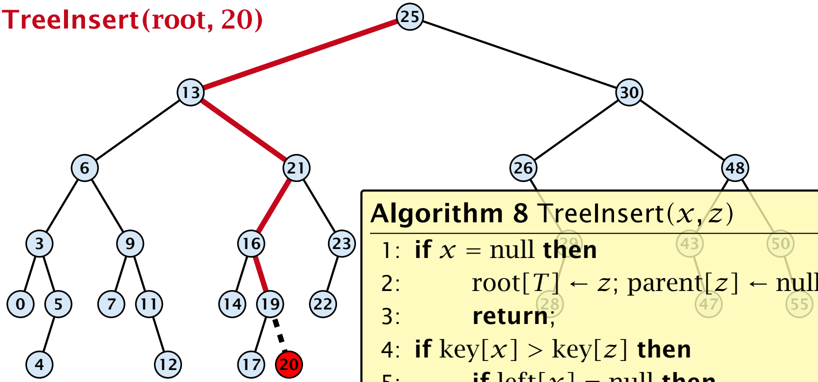
Algorithm 8 TreeInsert(x, z)

```
if x = null then
    root[T] ← z; parent[z] ← null;
    return;
if key[x] > key[z] then
    if left[x] = null then
        left[x] ← z; parent[z] ← x;
    else TreeInsert(left[x], z);
else
    if right[x] = null then
        right[x] ← z; parent[z] ← x;
    else TreeInsert(right[x], z);
```

Binary Search Trees: Insert

Insert element **not** in the tree.

TreeInsert(root, 20)

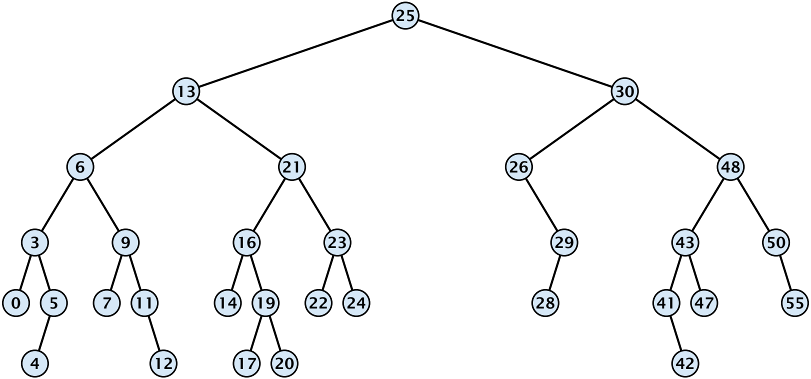


Search for z. At some point the search stops at a null-pointer. This is the place to insert z.

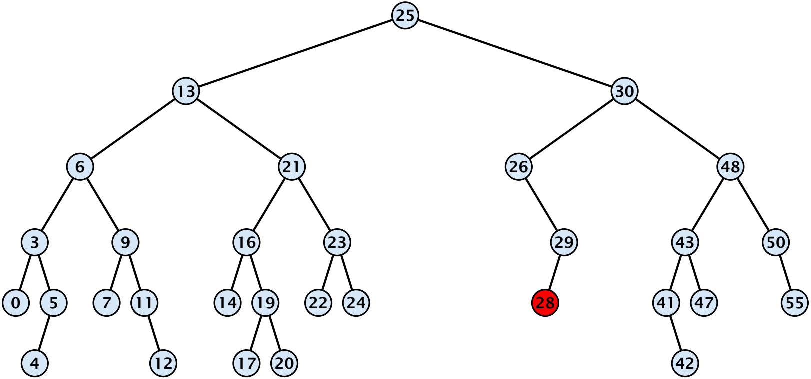
Algorithm 8 TreeInsert(x, z)

if x = null **then**
    root$[T] \leftarrow z$; parent$[z] \leftarrow$ null;
    return;
if key$[x]$ > key$[z]$ **then**
    if left$[x]$ = null **then**
        left$[x] \leftarrow z$; parent$[z] \leftarrow x$;
    else TreeInsert(left$[x]$, z);
else
    if right$[x]$ = null **then**
        right$[x] \leftarrow z$; parent$[z] \leftarrow x$;
    else TreeInsert(right$[x]$, z);

Binary Search Trees: Delete



Binary Search Trees: Delete

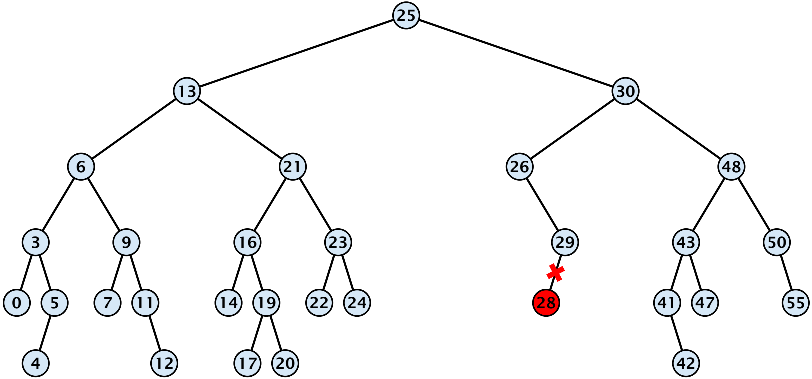


Case 1:

Element does not have any children

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

Binary Search Trees: Delete

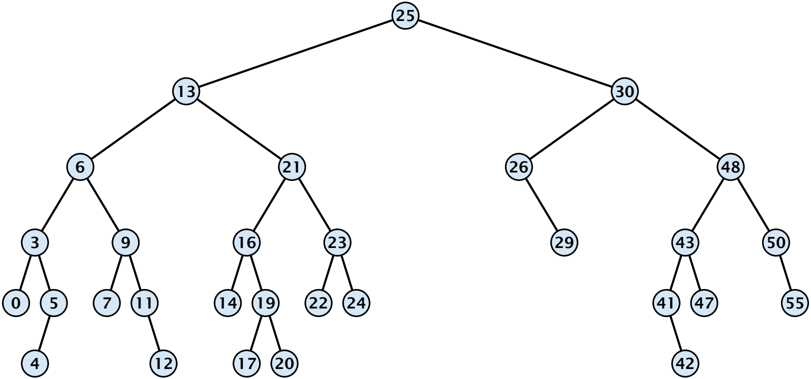


Case 1:

Element does not have any children

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

Binary Search Trees: Delete

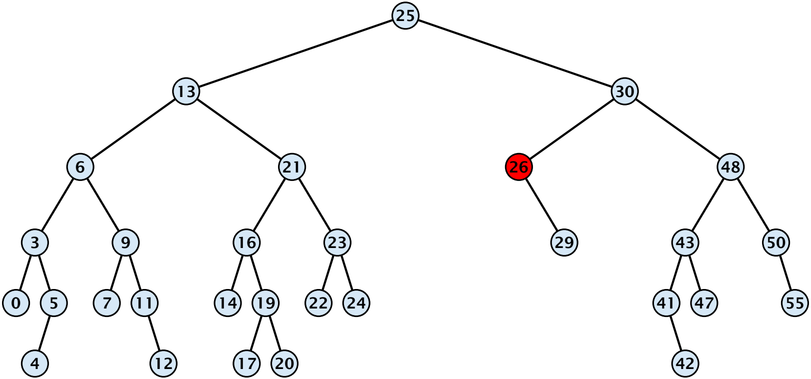


Case 1:

Element does not have any children

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

Binary Search Trees: Delete

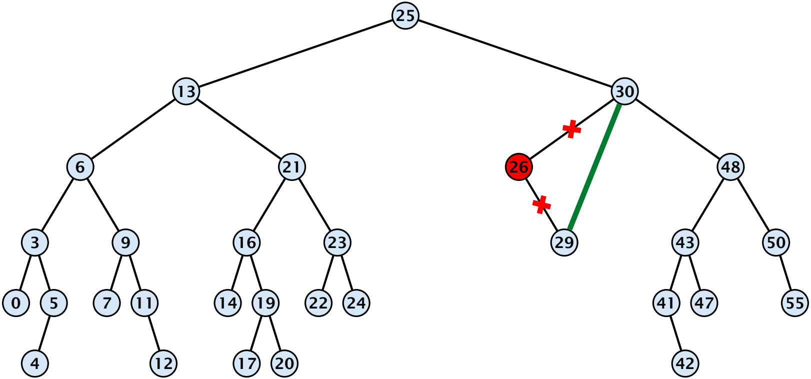


Case 2:

Element has exactly one child

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

Binary Search Trees: Delete

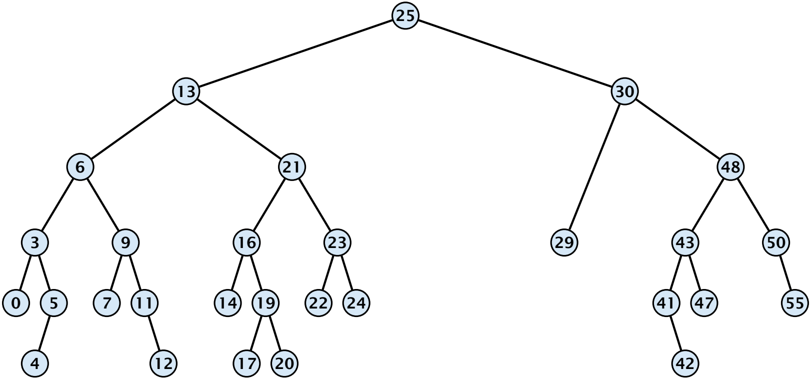


Case 2:

Element has exactly one child

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

Binary Search Trees: Delete

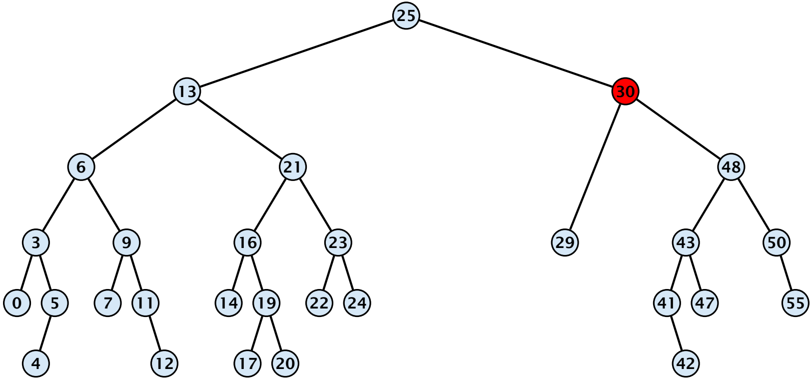


Case 2:

Element has exactly one child

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

Binary Search Trees: Delete

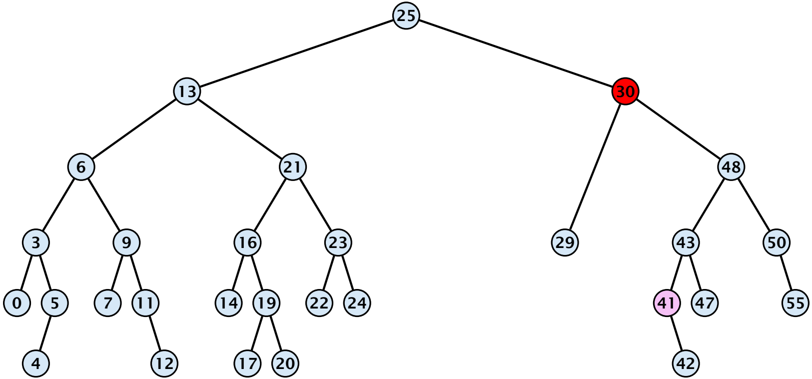


Case 3:

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

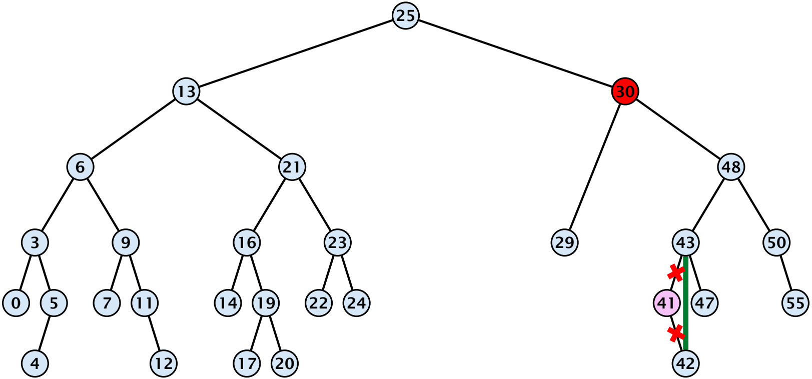


Case 3:

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

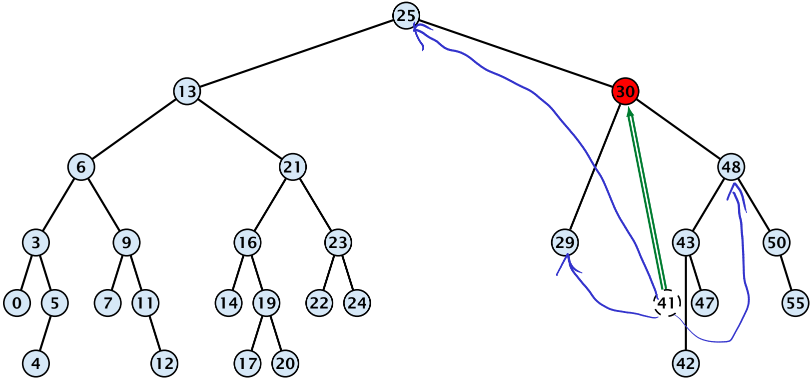


Case 3:

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

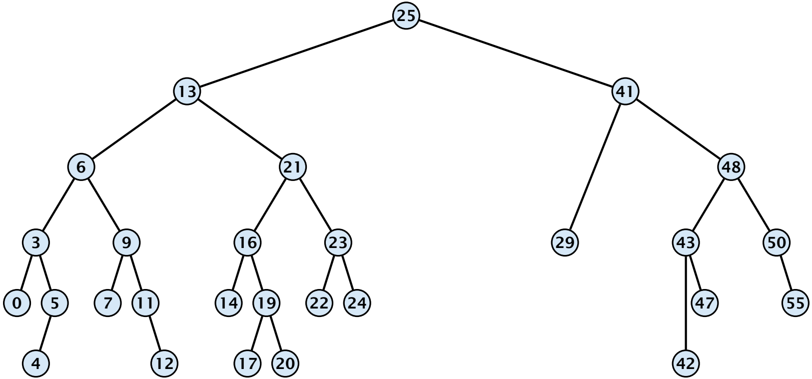


Case 3:

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

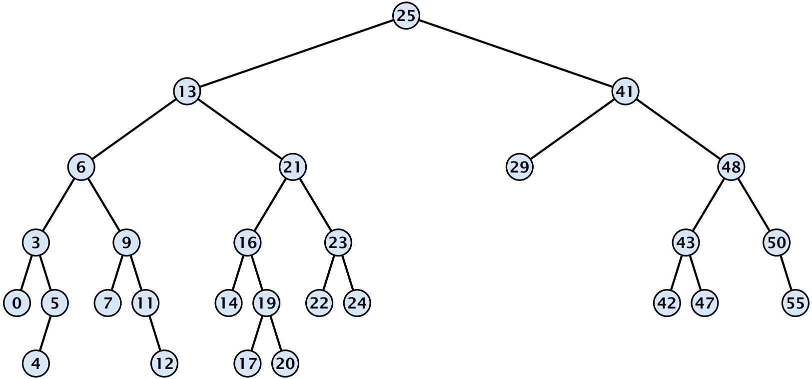


Case 3:

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



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)

if left$[z]$ = null **or** right$[z]$ = null
    then $y \leftarrow z$ **else** $y \leftarrow$ TreeSucc(z); — *select y to splice out*
if left$[y] \neq$ null
    then $x \leftarrow$ left$[y]$ **else** $x \leftarrow$ right$[y]$; — *x is child of y (or null)*
if $x \neq$ null **then** parent$[x] \leftarrow$ parent$[y]$; — *parent$[x]$ is correct*
if parent$[y]$ = null **then**
    root$[T] \leftarrow x$
else
    if y = left[parent$[y]$] **then**
        left[parent$[y]$] $\leftarrow x$
    else
        right[parent$[y]$] $\leftarrow x$
if $y \neq z$ **then** copy y-data to z

fix pointer to x (applies to the if/else block on parent$[y]$)

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.

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.

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.

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.

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.

7.2 Red Black Trees

Definition 12

A red black tree is a balanced binary search tree in which each internal node has two children. Each internal node has a color, such that

1. The root is black.
2. All leaf nodes are black.
3. For each node, all paths to descendant leaves contain the same number of black nodes.
4. If a node is red then both its children are black.

The null-pointers in a binary search tree are replaced by pointers to special null-vertices, that do not carry any object-data

7.2 Red Black Trees

Definition 12

A red black tree is a balanced binary search tree in which each internal node has two children. Each internal node has a color, such that

1. The root is black.
2. All leaf nodes are black.
3. For each node, all paths to descendant leaves contain the same number of black nodes.
4. If a node is red then both its children are black.

The null-pointers in a binary search tree are replaced by pointers to special null-vertices, that do not carry any object-data

7.2 Red Black Trees

Definition 12

A red black tree is a balanced binary search tree in which each internal node has two children. Each internal node has a color, such that

1. The root is black.
2. All leaf nodes are black.
3. For each node, all paths to descendant leaves contain the same number of black nodes.
4. If a node is red then both its children are black.

The null-pointers in a binary search tree are replaced by pointers to special null-vertices, that do not carry any object-data

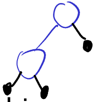
7.2 Red Black Trees

Definition 12

A red black tree is a balanced binary search tree in which each internal node has two children. Each internal node has a color, such that

1. The root is black.
2. All leaf nodes are black.
3. For each node, all paths to descendant leaves contain the same number of black nodes.
4. If a node is red then both its children are black.

The null-pointers in a binary search tree are replaced by pointers to special null-vertices, that do not carry any object-data



Definition 12

A red black tree is a balanced binary search tree in which each internal node has two children. Each internal node has a color, such that

1. The root is black.
2. All leaf nodes are black.
3. For each node, all paths to descendant leaves contain the same number of black nodes.
4. If a node is red then both its children are black.

The null-pointers in a binary search tree are replaced by pointers to special null-vertices, that do not carry any object-data

7.2 Red Black Trees

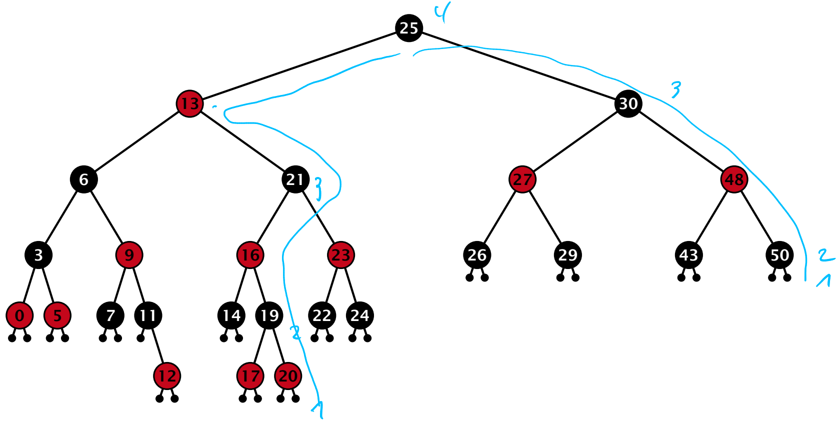
Definition 12

A red black tree is a balanced binary search tree in which each internal node has two children. Each internal node has a color, such that

1. The root is black.
2. All leaf nodes are black.
3. For each node, all paths to descendant leaves contain the same number of black nodes.
4. If a node is red then both its children are black.

The null-pointers in a binary search tree are replaced by pointers to special null-vertices, that do not carry any object-data

Red Black Trees: Example



7.2 Red Black Trees

Lemma 13

A red-black tree with n internal nodes has height at most $\mathcal{O}(\log n)$.

Definition 14

The black height $\text{bh}(v)$ of a node v in a red black tree is the number of black nodes on a path from v to a leaf vertex (not counting v).

We first show:

Lemma 15

A sub-tree of black height $\text{bh}(v)$ in a red black tree contains at least $2^{\text{bh}(v)} - 1$ internal vertices.

7.2 Red Black Trees

Lemma 13

A red-black tree with n internal nodes has height at most $\mathcal{O}(\log n)$.

Definition 14

The black height $\mathrm{bh}(v)$ of a node v in a red black tree is the number of black nodes on a path from v to a leaf vertex (not counting v).

We first show:

Lemma 15

A sub-tree of black height $\mathrm{bh}(v)$ in a red black tree contains at least $2^{\mathrm{bh}(v)} - 1$ internal vertices.

7.2 Red Black Trees

Lemma 13

A red-black tree with n internal nodes has height at most $\mathcal{O}(\log n)$.

Definition 14

The black height $\mathrm{bh}(v)$ of a node v in a red black tree is the number of black nodes on a path from v to a leaf vertex (not counting v).

We first show:

Lemma 15

A sub-tree of black height $\mathrm{bh}(v)$ in a red black tree contains at least $2^{\mathrm{bh}(v)} - 1$ internal vertices.