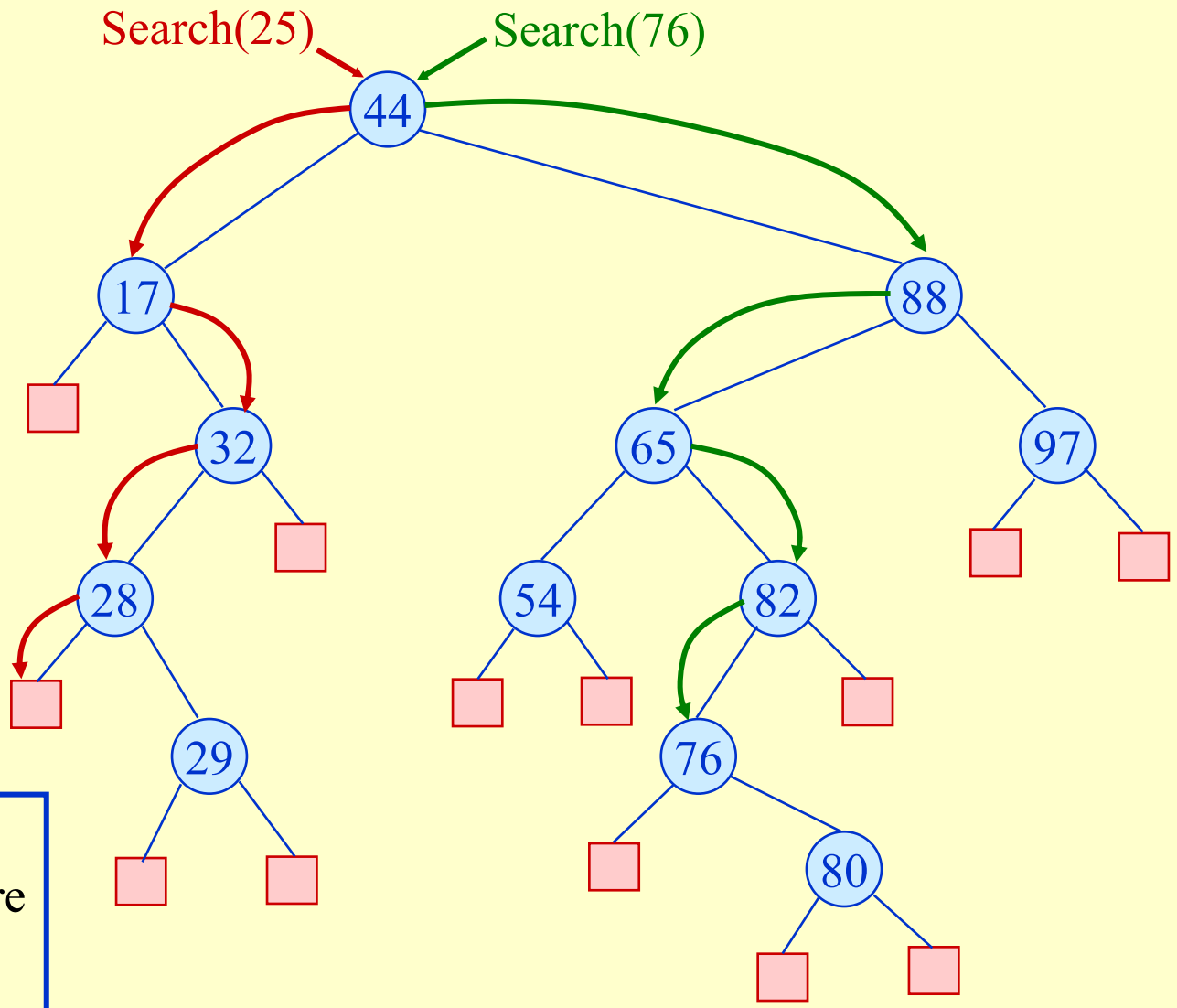
Splay Trees

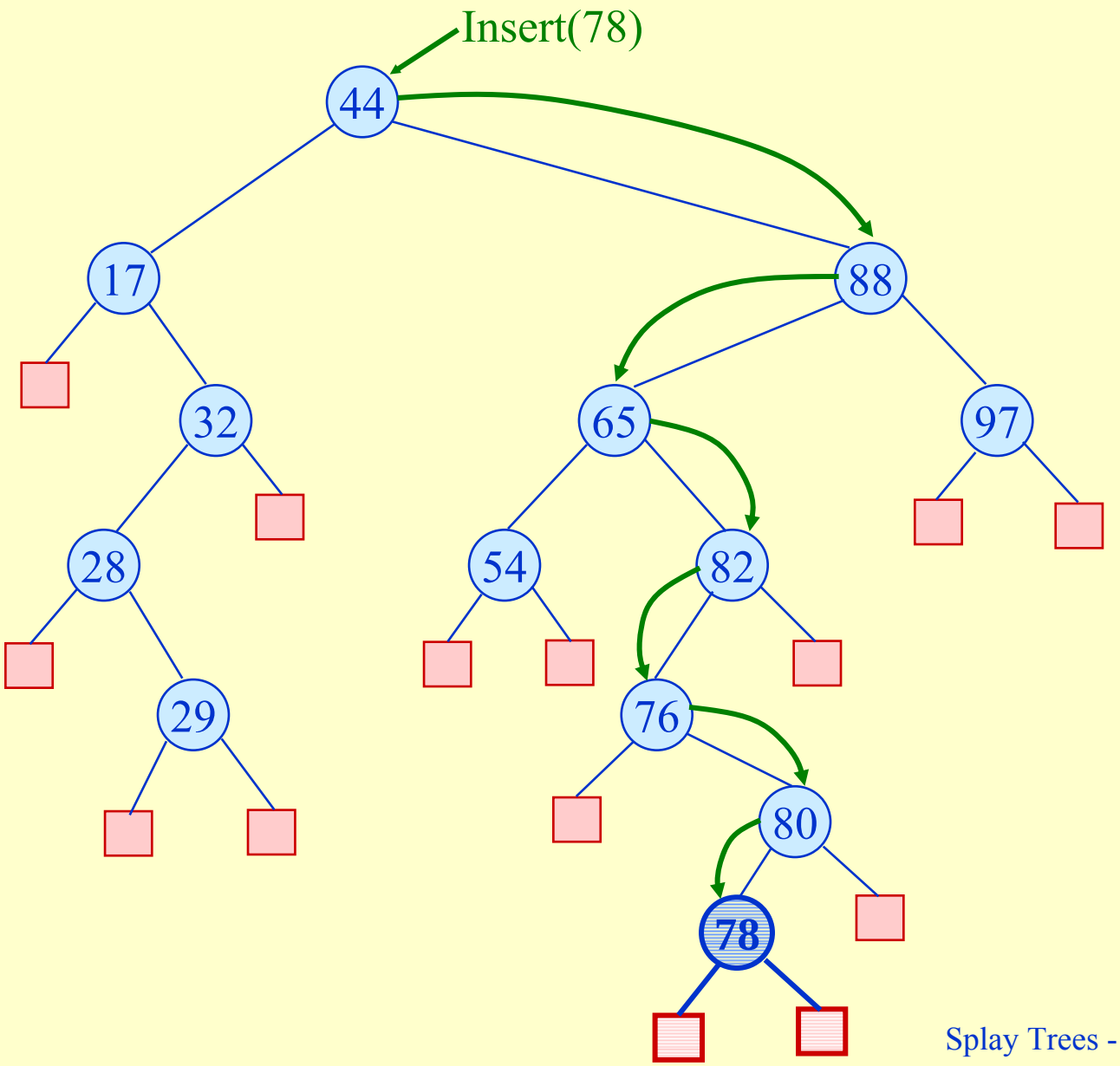
- In balanced tree schemes, explicit rules are followed to ensure balance.
- In splay trees, there are no such rules.
- Search, insert, and delete operations are like in binary search trees, except at the end of each operation a special step called **splaying** is done.
- Splaying ensures that all operations take O(lg n) amortized time.
- First, a quick review of BST operations…

BST: Search



Note:
Sentinel leaf nodes are assumed.
$\Rightarrow$ tree with n keys has 2n+1 nodes.

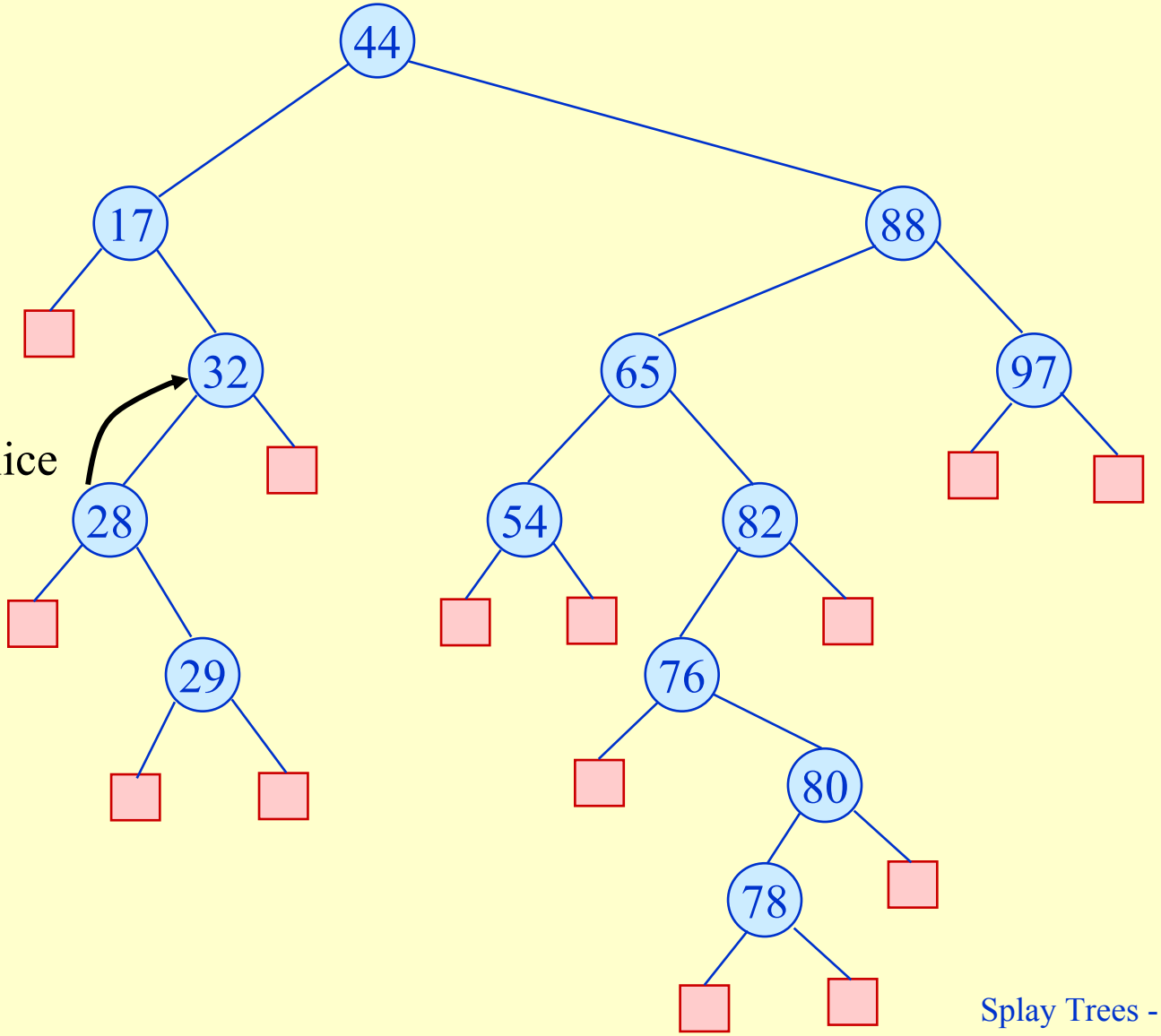
BST: Insert



BST: Delete

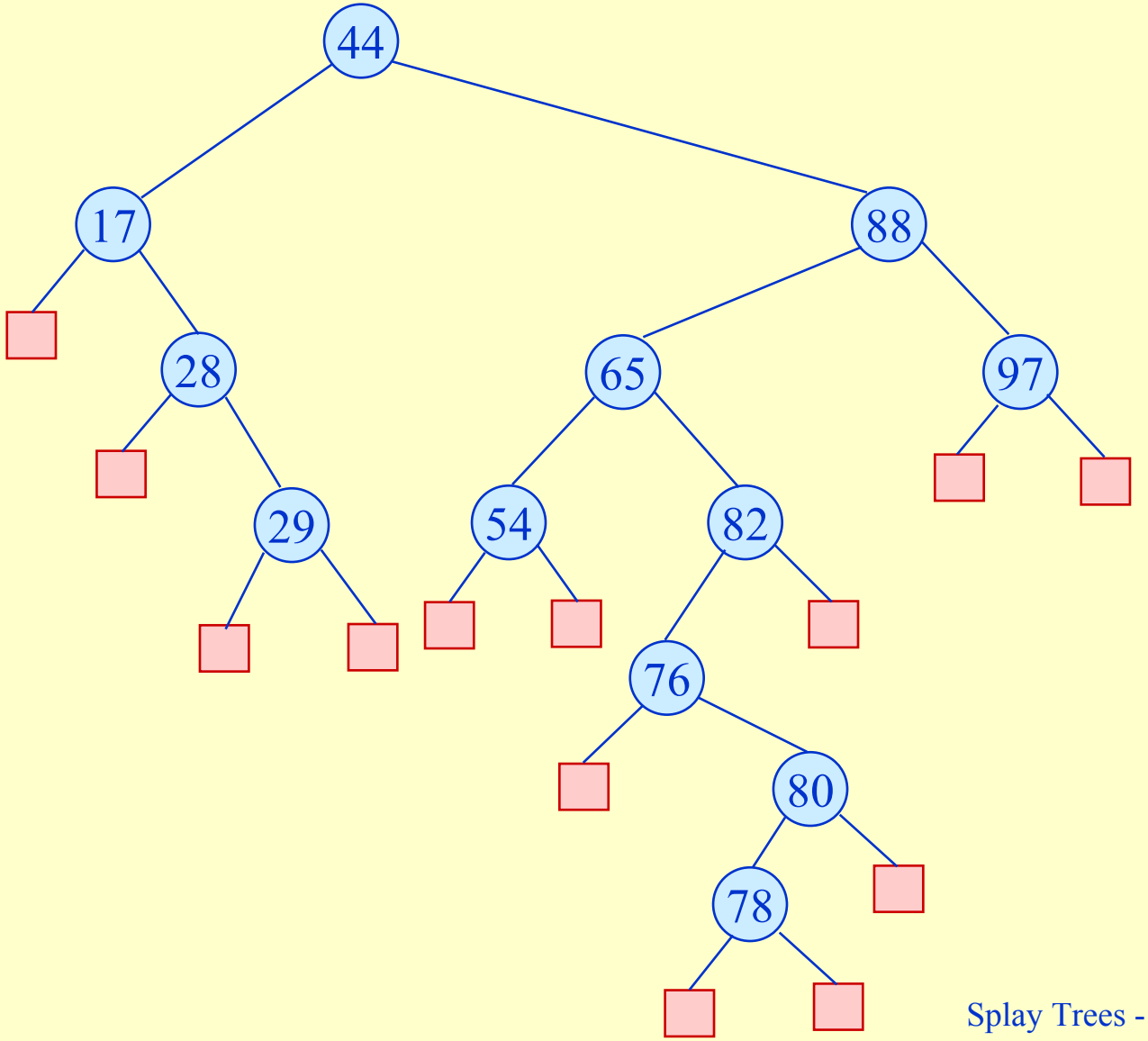
Delete(32)

Has only one child: just splice out 32.



BST: Delete

Delete(32)

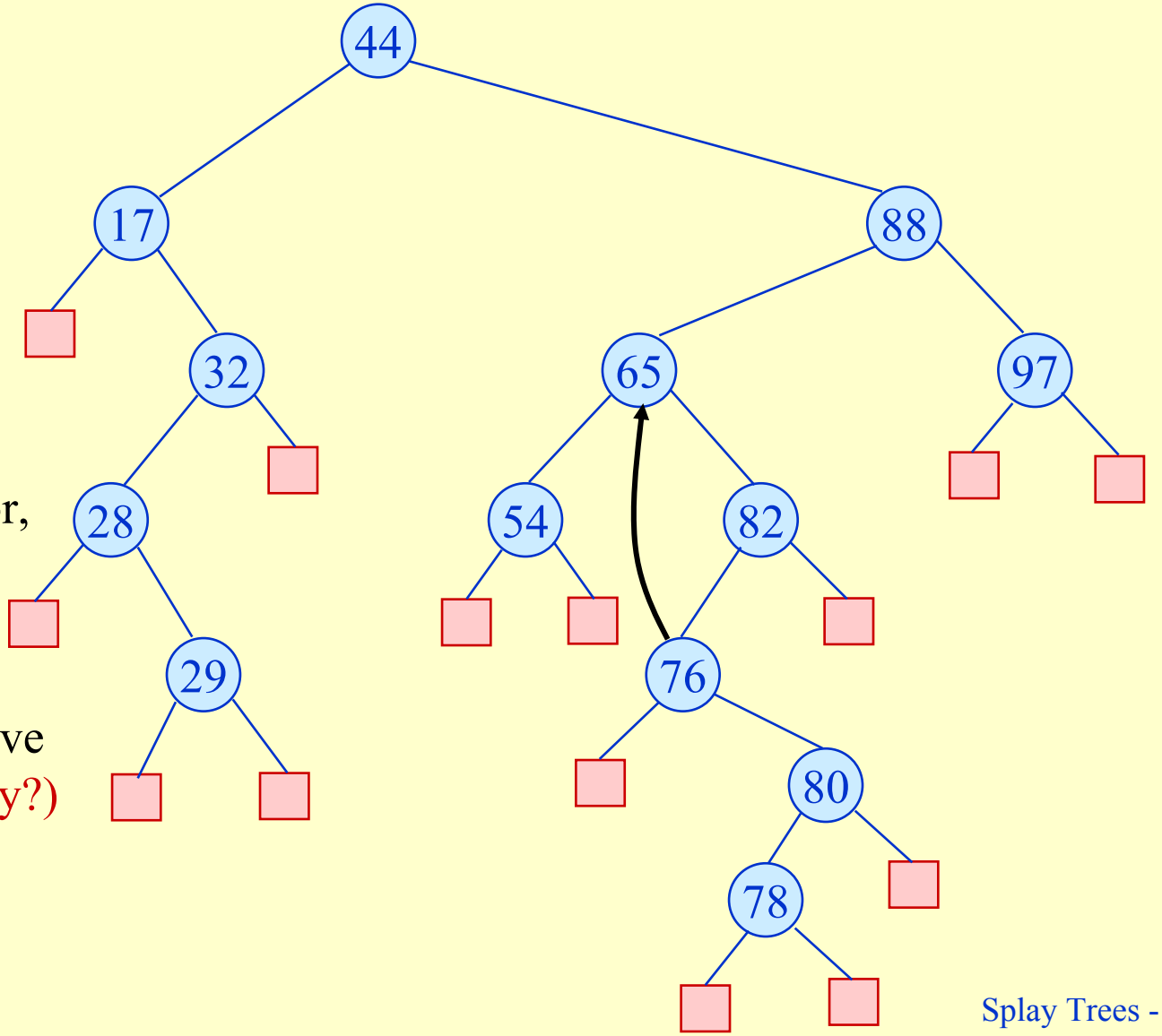


BST: Delete

Delete(65)

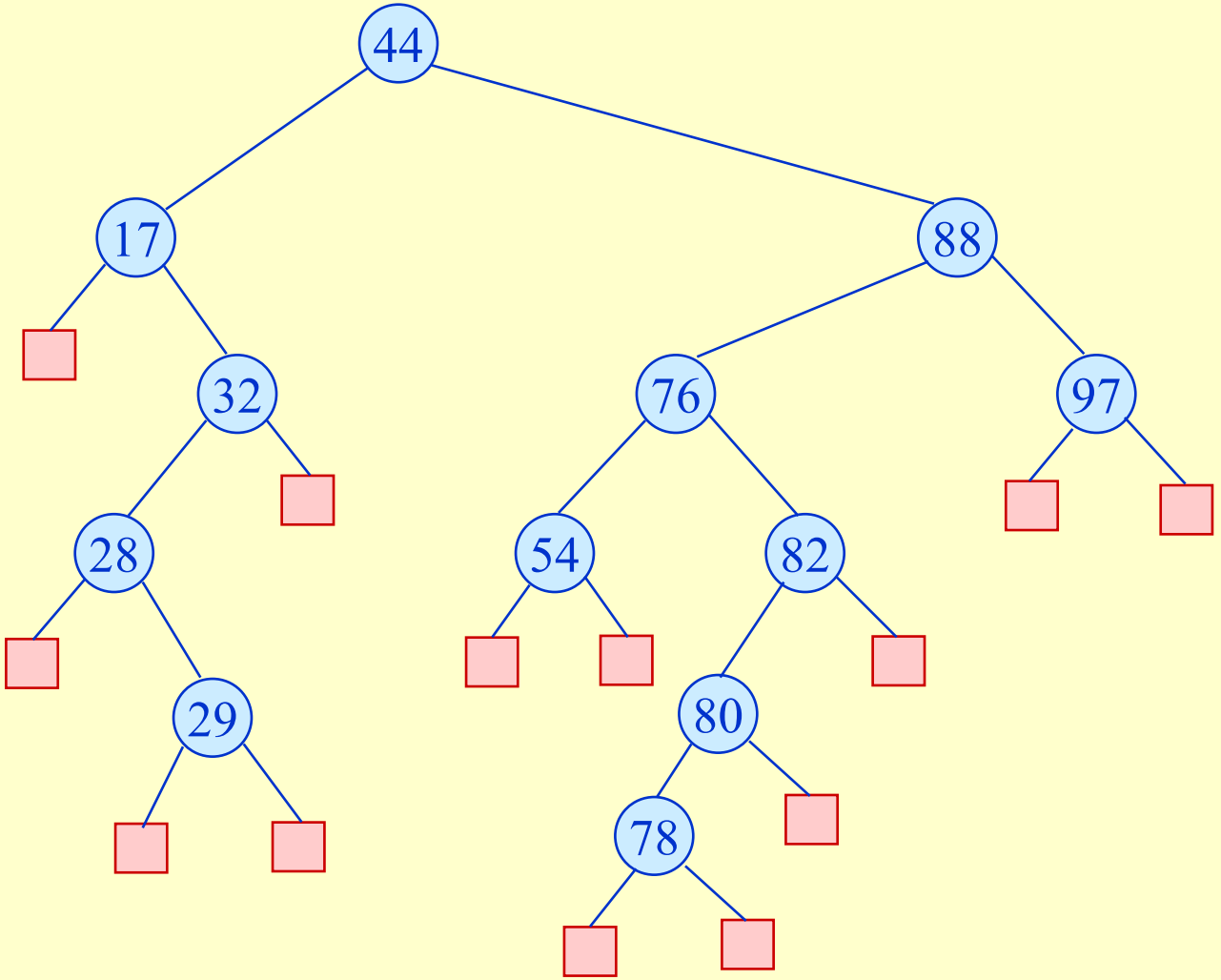
Has two children:
Replace 65 by successor, 76, and splice out successor.

Note: Successor can have at most one child. (Why?)



BST: Delete

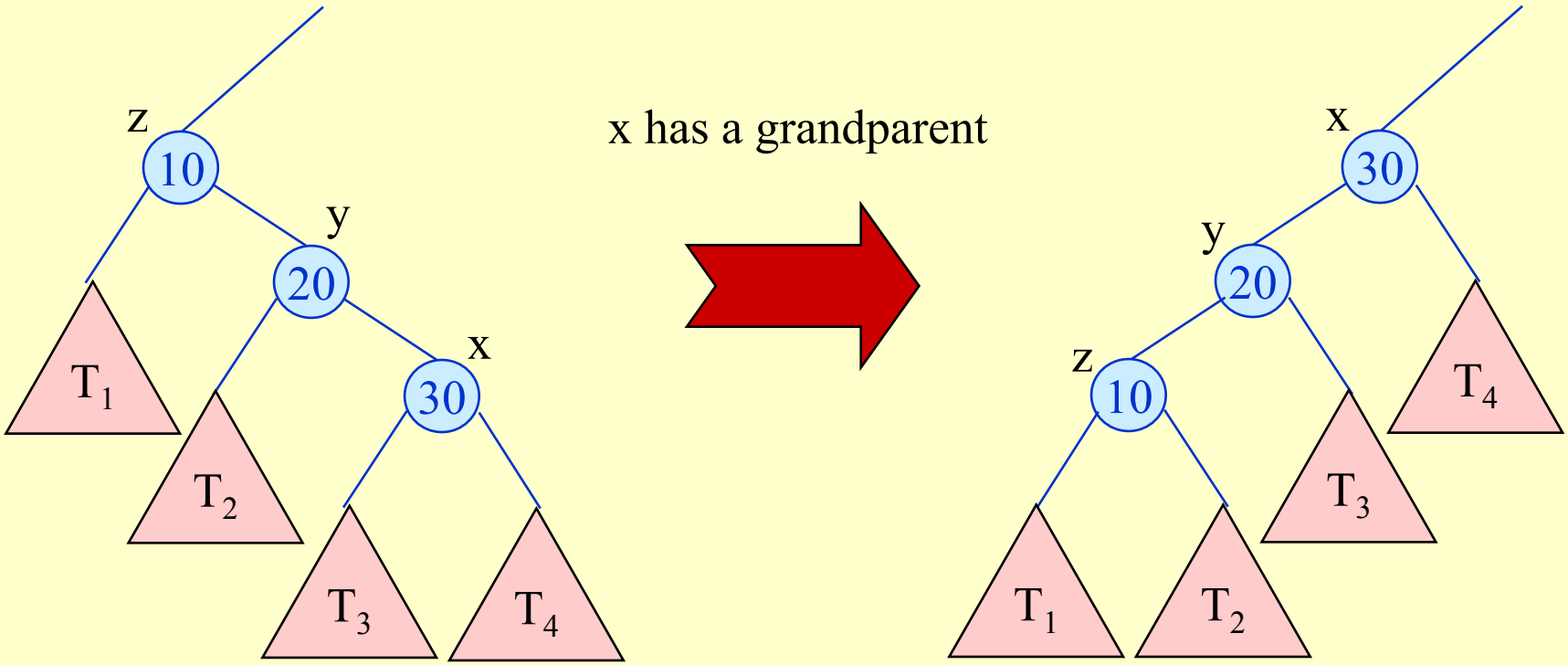
Delete(65)



Splaying

- In splay trees, after performing an ordinary BST Search, Insert, or Delete, a **splay operation** is performed on some node x (as described later).
- The splay operation moves x to the root of the tree.
- The splay operation consists of sub-operations called zig-zig, zig-zag, and zig.

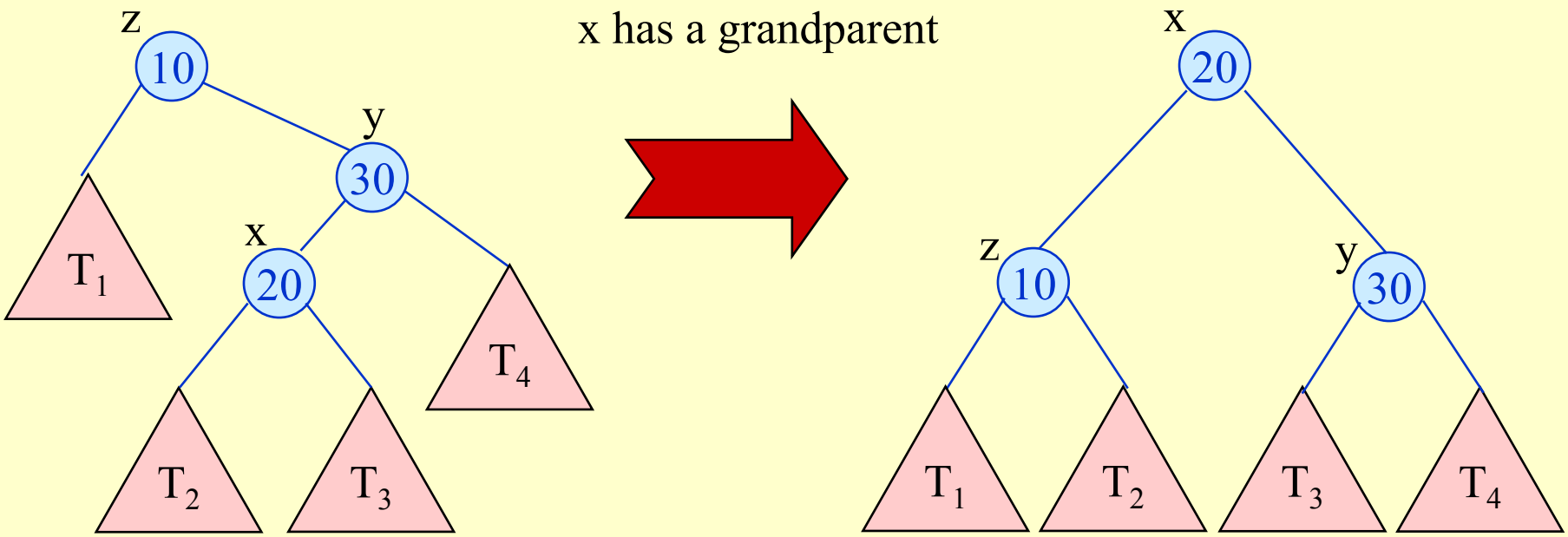
Zig-Zig



(Symmetric case too)

Note: x's depth decreases by two.

Zig-Zag



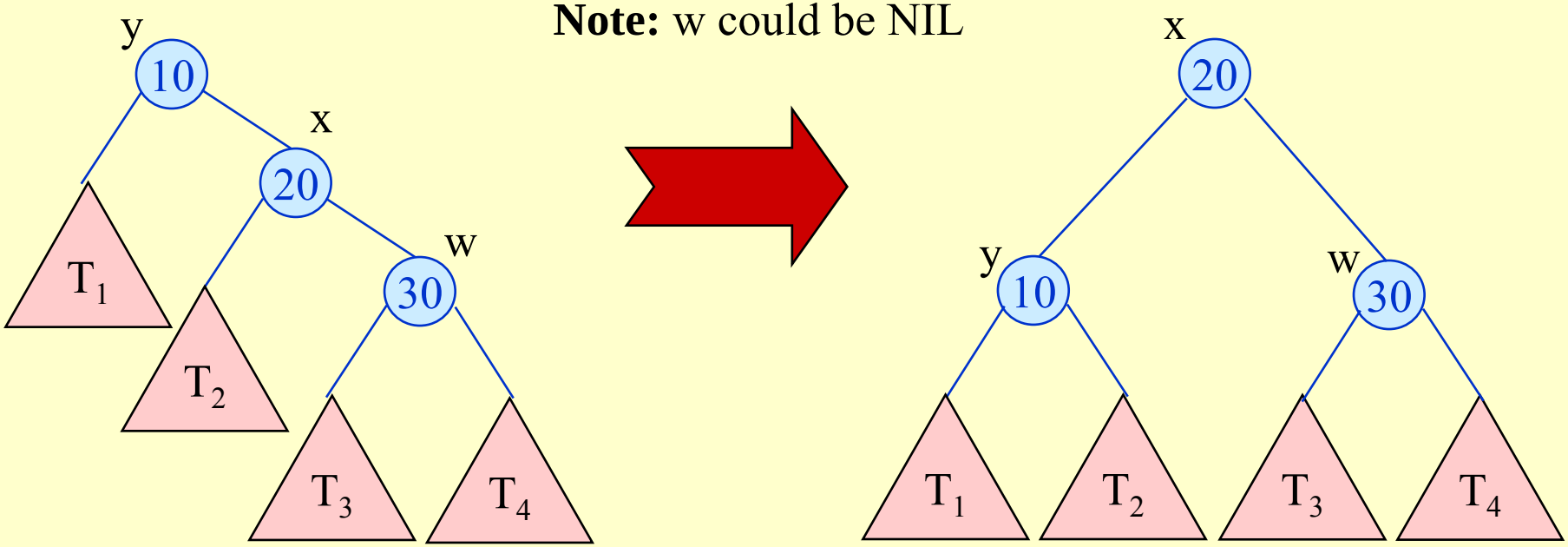
(Symmetric case too)

Note: x's depth decreases by two.

Zig

x has no grandparent (so, y is the root)

Note: w could be NIL



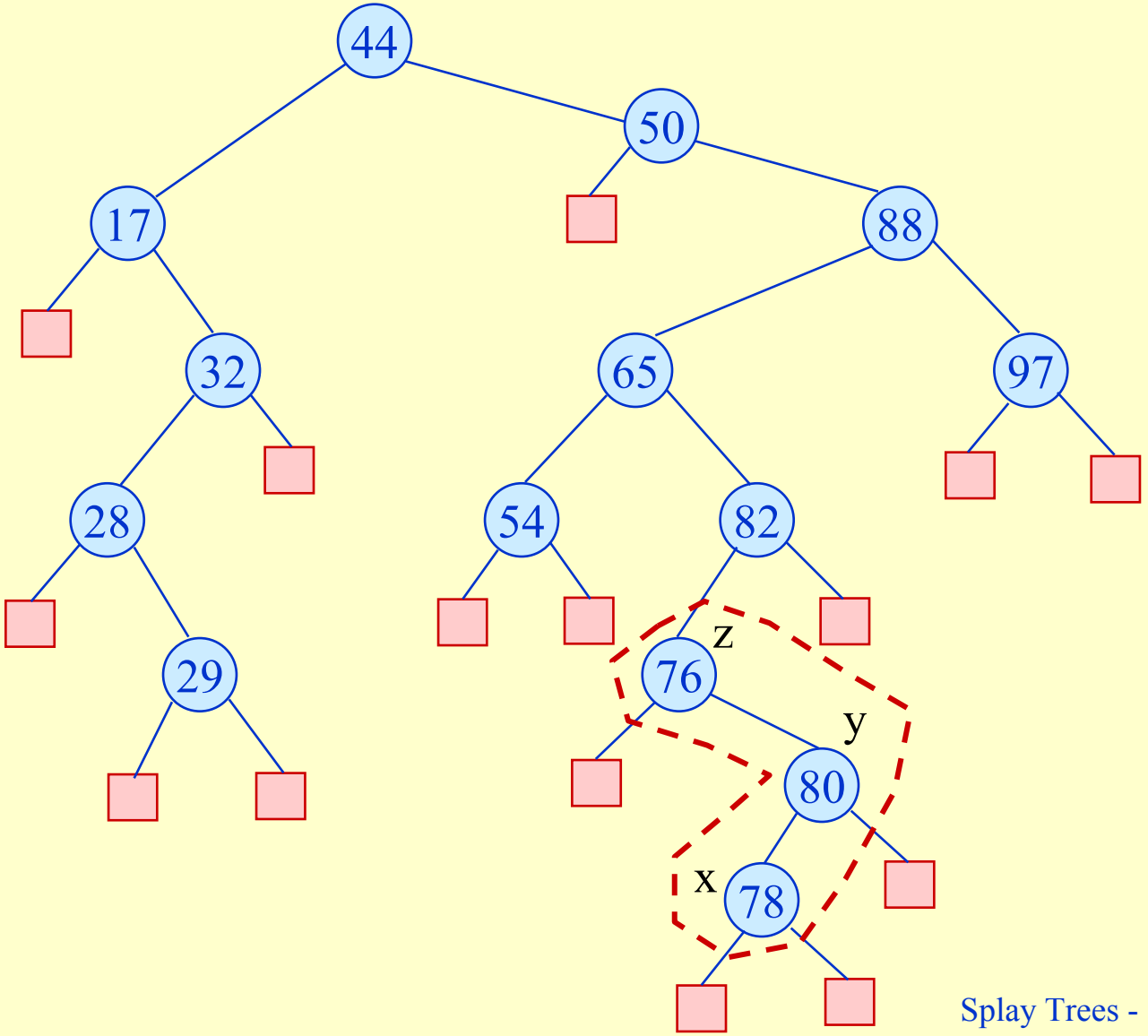
(Symmetric case too)

Note: x's depth decreases by one.

Complete Example

Splay(78)

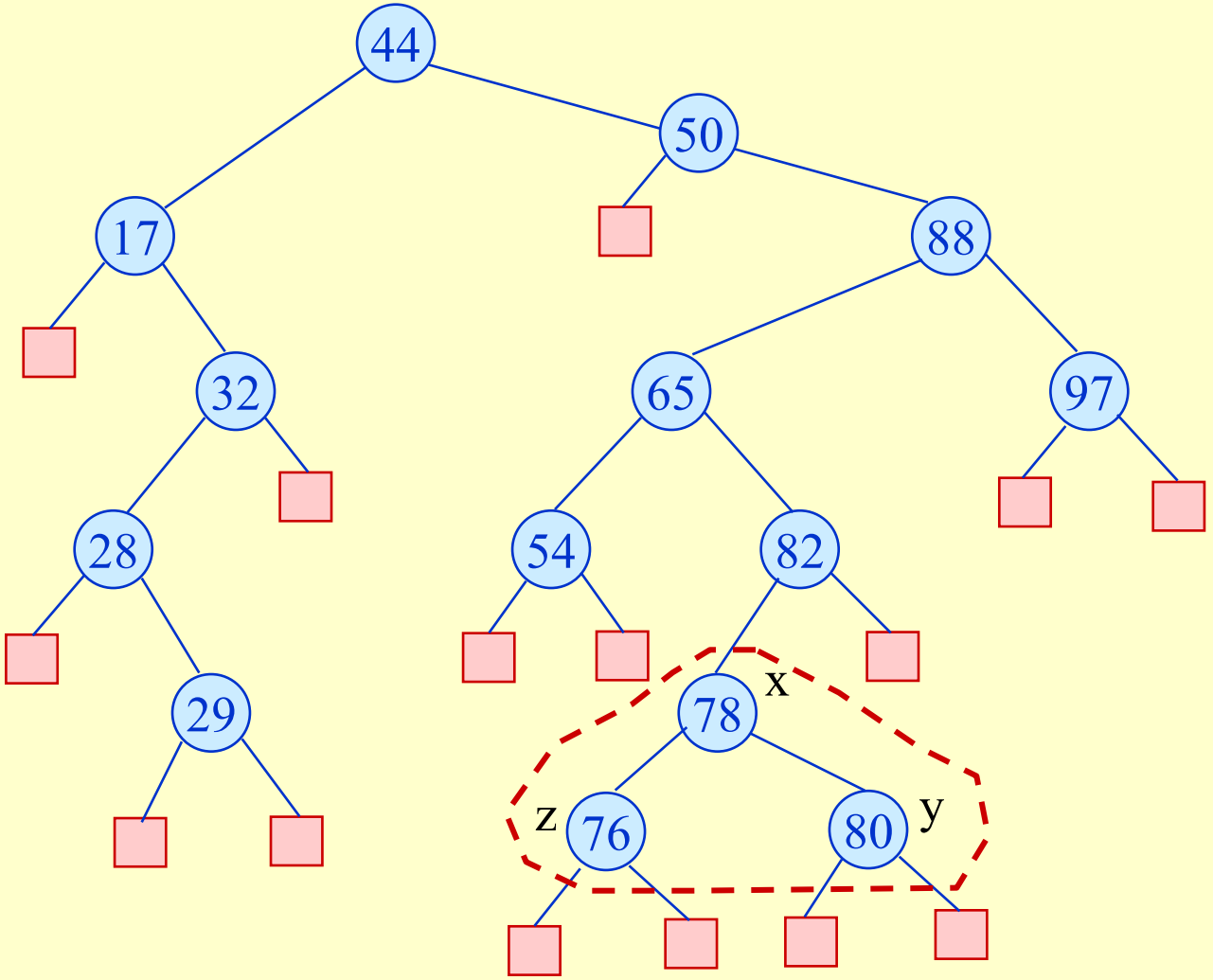
zig-zag



Complete Example

Splay(78)

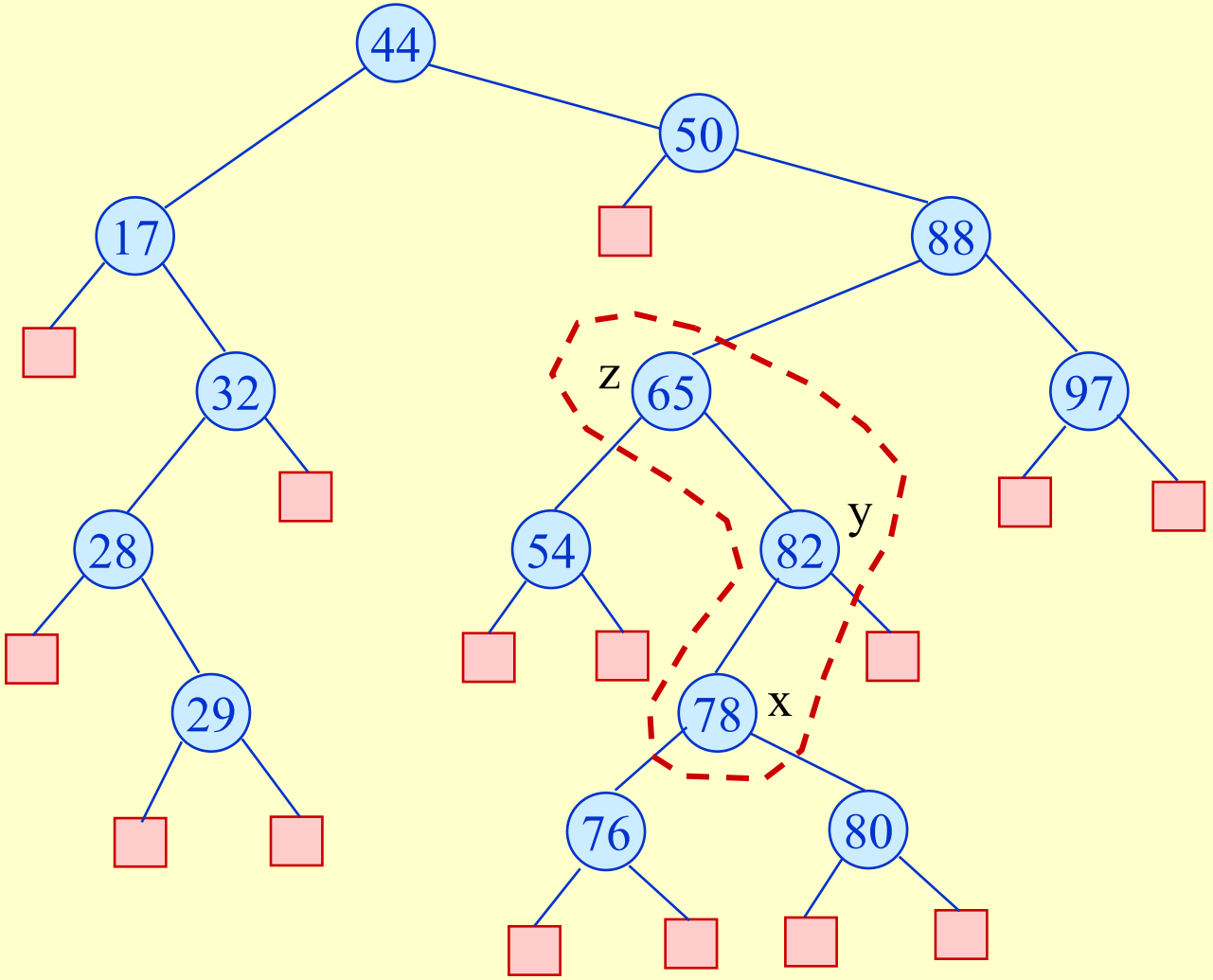
zig-zag



Complete Example

Splay(78)

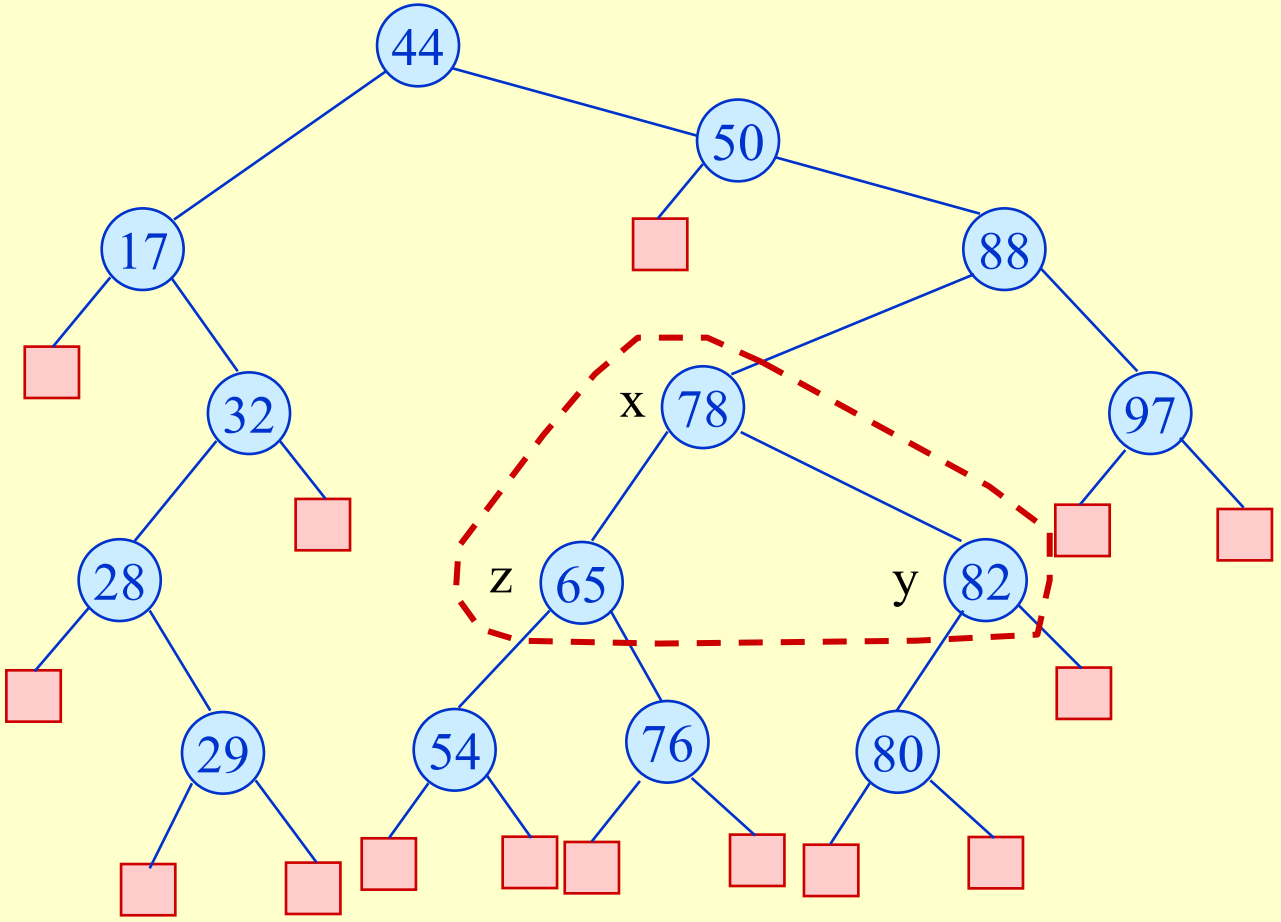
zig-zag



Complete Example

Splay(78)

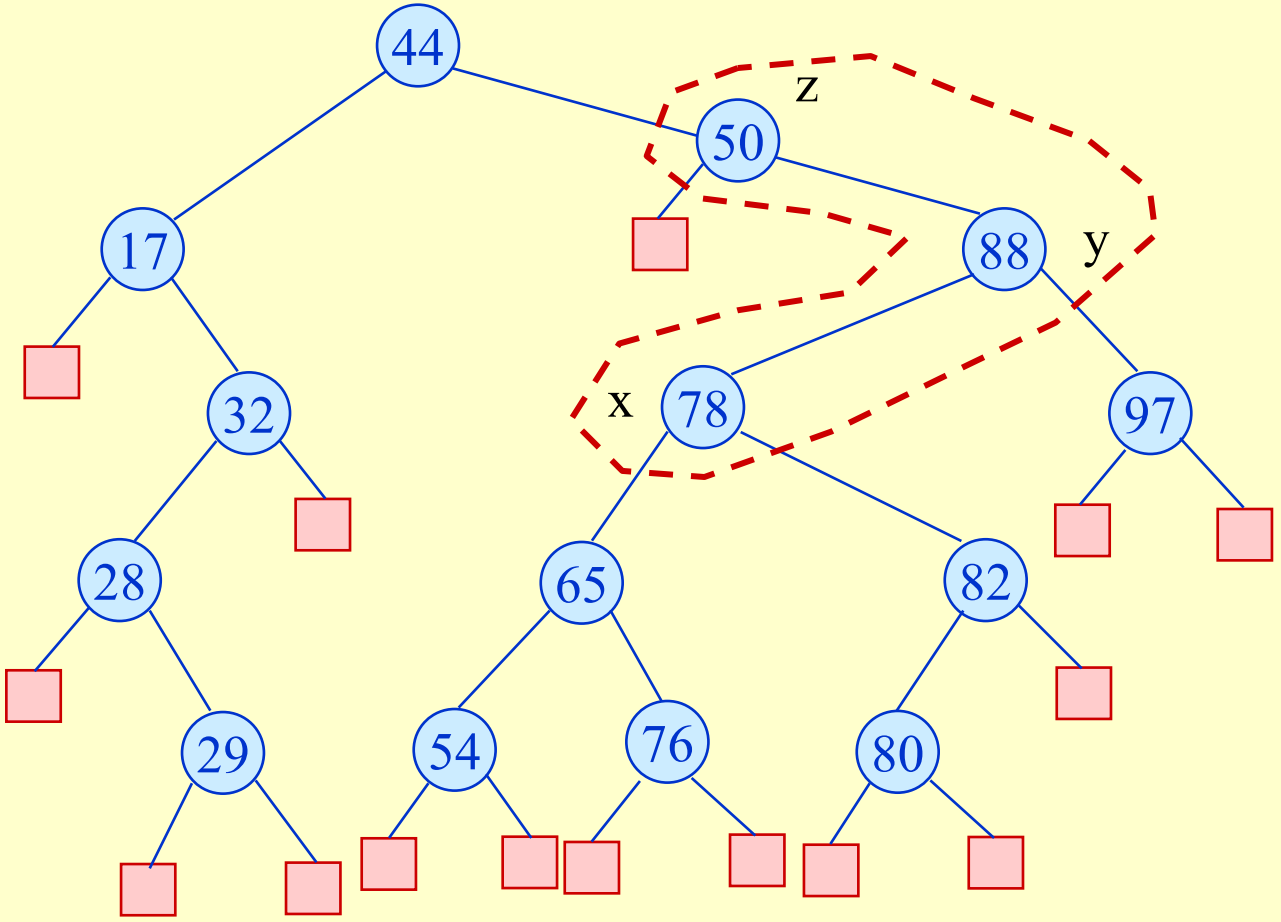
zig-zag



Complete Example

Splay(78)

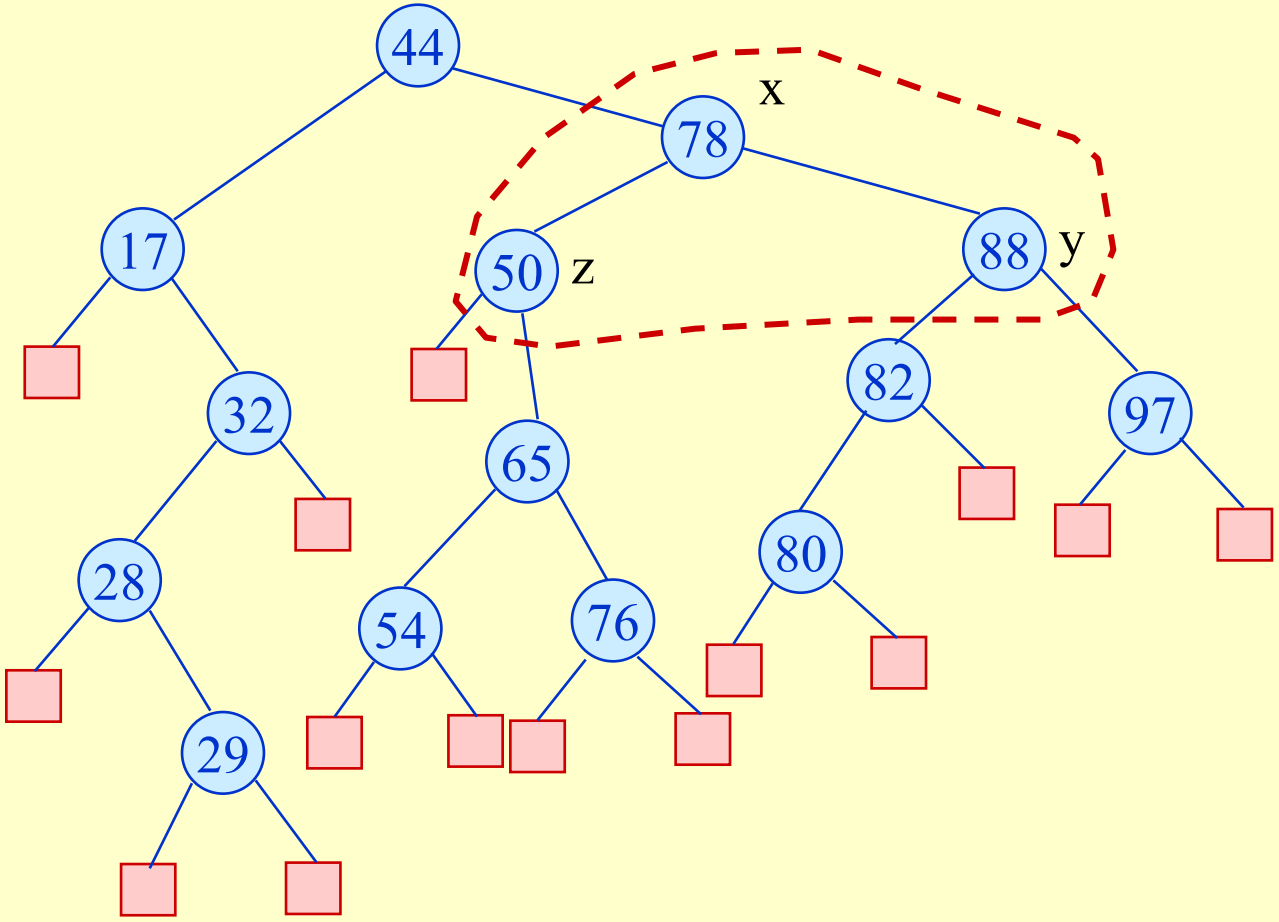
zig-zag



Complete Example

Splay(78)

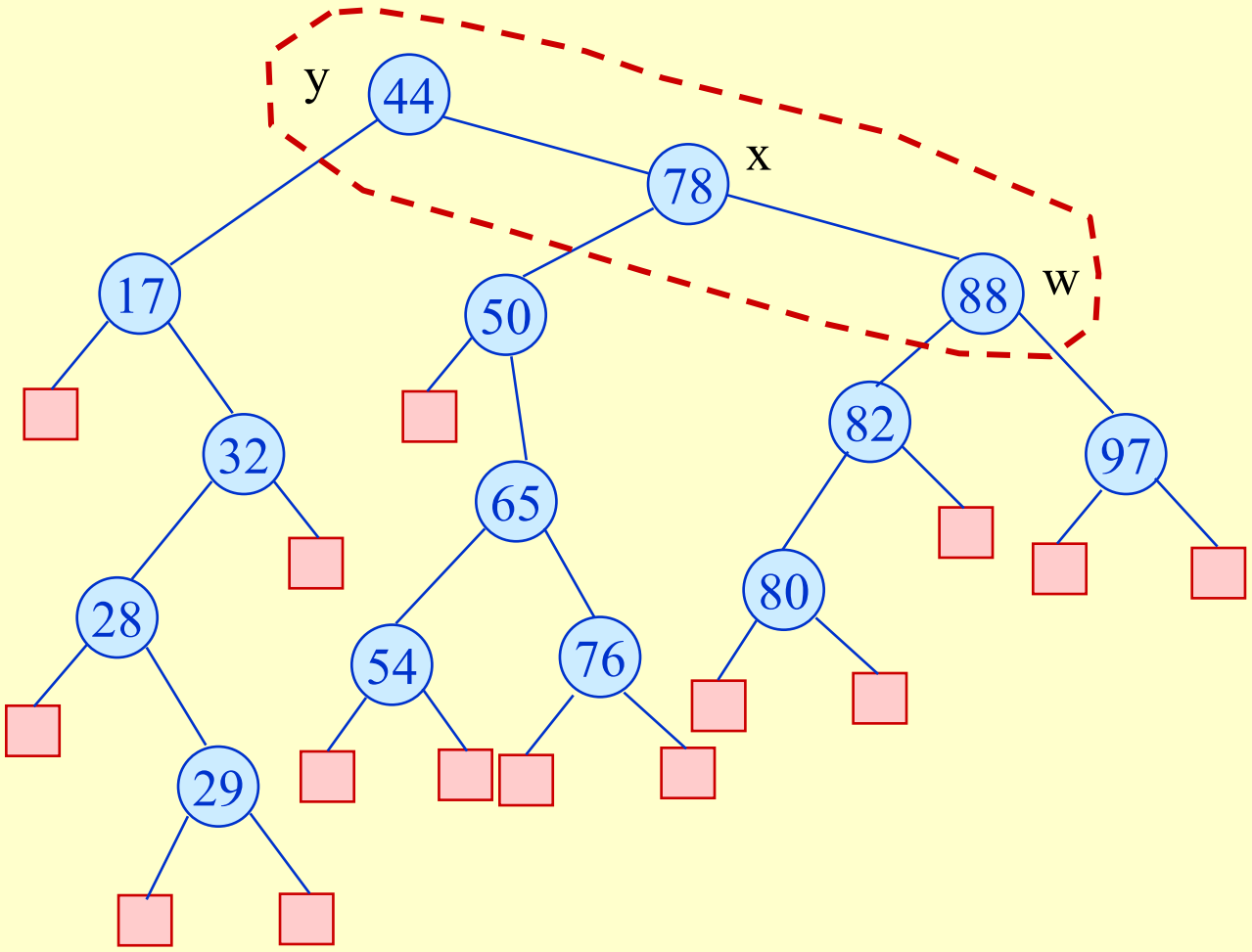
zig-zag



Complete Example

Splay(78)

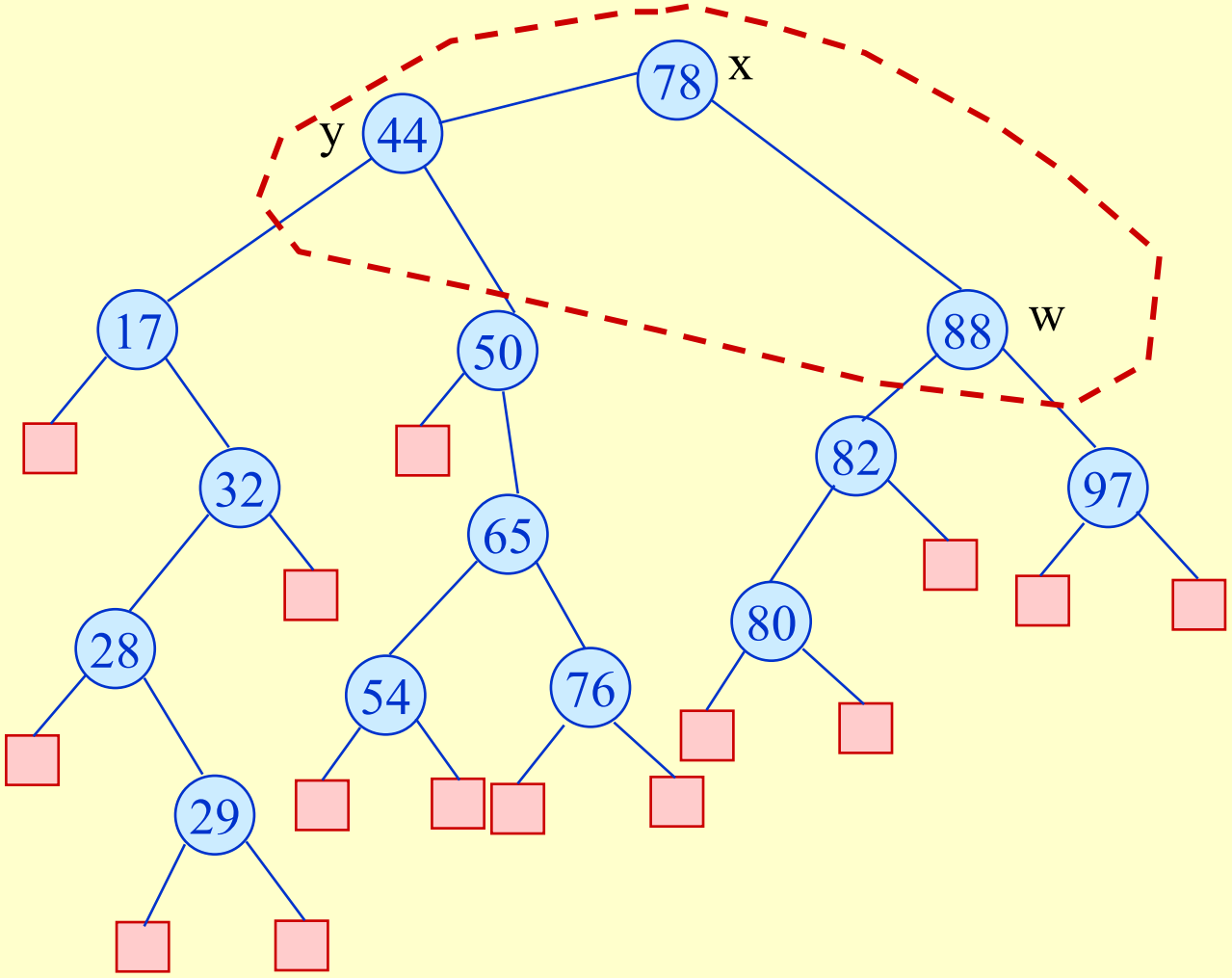
zig



Complete Example

Splay(78)

zig



Result of splaying

- The result is a binary tree, with the left subtree having all keys less than the root, and the right subtree having keys greater than the root.
- Also, the final tree is “more balanced” than the original.
- However, if an operation near the root is done, the tree can become less balanced.

When to Splay

- **Search:**
  - Successful: Splay node where key was found.
  - Unsuccessful: Splay last-visited internal node (i.e., last node with a key).
- **Insert:**
  - Splay newly added node.
- **Delete:**
  - Splay parent of removed node (which is either the node with the deleted key or its successor).
- **Note:** All operations run in O(h) time, for a tree of height h.

Amortized Analysis Review

- **Accounting Method**
  - **Idea:** When an operation's amortized cost exceeds it actual cost, the difference is assigned to certain tree nodes as **credit**.
  - Credit is used to pay for subsequent operations whose amortized cost is less than their actual cost.
- Most of our analysis will focus on splaying.
  - The BST operations will be easily dealt with at the end.

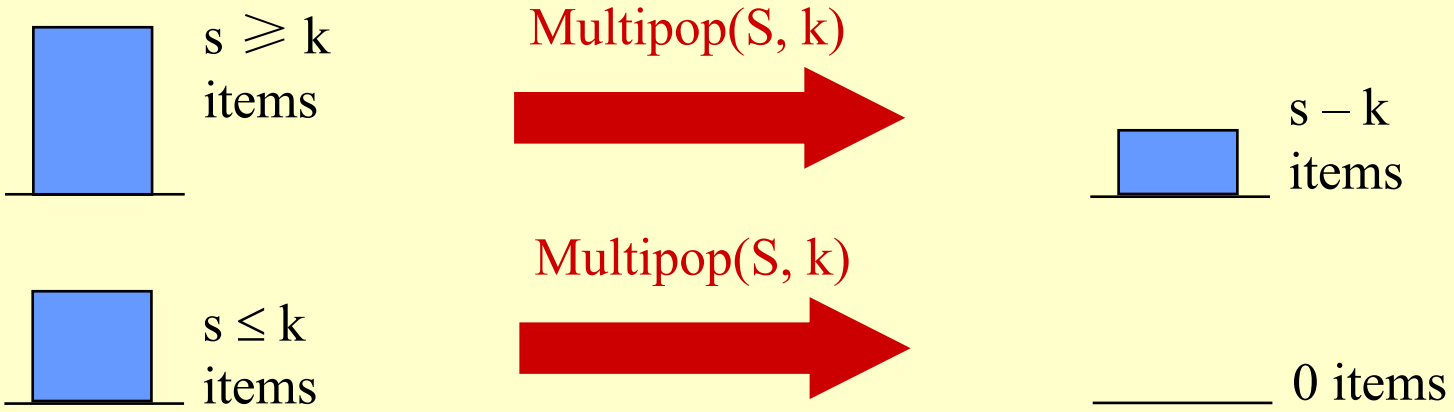
Review: Accounting Method

Stack Example:

- Operations:
  - **Push(S, x)**.
  - **Pop(S)**.

  } Can implement in O(1) time.

  - **Multipop(S, k):** if stack has s items, pop off min(s, k) items.



Accounting Method (Continued)

- We charge each operation an amortized cost.
- Charge may be more or less than actual cost.
- If more, then we have **credit**.
- This credit can be used to pay for future operations whose amortized cost is less than their actual cost.
- **Require:** For any sequence of operations, amortized cost upper bounds worst-case cost.
  - That is, we always have nonnegative credit.

Accounting Method (Continued)

Stack Example:

Actual Costs:

Push:	1
Pop:	1
Multipop:	min(s, k)

Amortized Costs:

Push:	2
Pop:	0
Multipop:	0

All O(1).

Pays for the push and a future pop.

For a sequence of n operations, does total amortized cost upper bound total worst-case cost, as required?

What is the total worst-case cost of the sequence?

Review: Potential method

IDEA: View the bank account as the potential energy (*à la* physics) of the dynamic set.

Framework:

- Start with an initial data structure D_0.
- Operation i transforms D_{i-1} to D_i.
- The cost of operation i is c_i.
- Define a ***potential function*** $\Phi : \{D_i\} \to \mathsf{R}$, such that $\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i.
- The ***amortized cost*** $\hat{c}_i$ with respect to Φ is defined to be $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$.

Potential method II

- Like the accounting method, but think of the credit as ***potential*** stored with the ***entire data structure***.
  - Accounting method stores credit with specific objects while potential method stores potential in the data structure as a whole.
  - Can release potential to pay for future operations
- Most flexible of the amortized analysis methods.

Understanding potentials

$$\hat{c}_i = c_i + \underbrace{\Phi(D_i) - \Phi(D_{i-1})}_{\textbf{\textit{potential difference}}\ \Delta\Phi_i}$$

- If $\Delta\Phi_i > 0$, then $\hat{c}_i > c_i$. Operation i stores work in the data structure for later use.
- If $\Delta\Phi_i < 0$, then $\hat{c}_i < c_i$. The data structure delivers up stored work to help pay for operation i.

Amortized costs bound the true costs

The total amortized cost of n operations is

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} \left(c_i + \Phi(D_i) - \Phi(D_{i-1})\right)$$

Summing both sides.

Amortized costs bound the true costs

The total amortized cost of n operations is

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} \left(c_i + \Phi(D_i) - \Phi(D_{i-1})\right)$$

$$= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

The series telescopes.

Amortized costs bound the true costs

The total amortized cost of n operations is

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} \left(c_i + \Phi(D_i) - \Phi(D_{i-1})\right)$$

$$= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

$$\geq \sum_{i=1}^{n} c_i$$

since $\Phi(D_n) \geq 0$ and $\Phi(D_0) = 0$.

Stack Example: Potential

Define: $\phi(D_i)$ = #items in stack Thus, $\phi(D_0)=0$.

Plug in for operations:

Push: $\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1})$

$= 1 + j - (j-1)$

$= 2$

Pop: $\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1})$

$= 1 + (j-1) - j$

$= 0$

Multi-pop: $\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1})$

$= k' + (j-k') - j$ $k'=\min(|S|,k)$

$= 0$

Thus O(1) amortized time per op.

Ranks

- T is a splay tree with n keys.
- **Definition:** The **size** of node v in T, denoted n(v), is the number of nodes in the subtree rooted at v. (In Sleator & Tarjan Paper, there is a weight w(i) attached to each node.)
  - **Note:** The root is of size 2n+1.
- **Definition:** The **rank** of v, denoted r(v), is lg(n(v)).
  - **Note:** The root has rank lg(2n+1).
- **Definition:** $r(T) = \sum_{v \in T} r(v)$.

Meaning of Ranks

- The rank of a tree is a measure of how well balanced it is.
- A well balanced tree has a low rank.
- A badly balanced tree has a high rank.
- The splaying operations tend to make the rank smaller, which balances the tree and makes other operations faster.
- Some operations near the root may make the rank larger and slightly unbalance the tree.
- Amortized analysis is used on splay trees, with the rank of the tree being the potential .($\Phi(T) = r(T)$)

Credit Invariant

- We will define amortized costs so that the following invariant is maintained.

> Each node v of T has r(v) credits in its account.

  - So, each operation's **amortized cost** = its real cost + the total change in r(T) it causes (positive or negative).

- Let R_i = op. i's real cost and Δ_i = change in r(T) it causes. Total am. cost = $\Sigma_{i=1,\ldots,n} (R_i + \Delta_i)$. Initial tree has rank 0 & final tree has non-neg. rank. So, $\Sigma_{i=1,\ldots n} \Delta_i \geq 0$, which implies total am. cost ≥ total real cost.

What's Left?

- We want to show that the per-operation amortized cost is *logarithmic*.
- To do this, we need to look at how BST operations and splay operations affect r(T).
  - We spend most of our time on splaying, and consider the specific BST operations later.
- To analyze splaying, we first look at how r(T) changes as a result of a single substep, i.e., zig, zig-zig, or zig-zag.
  - **Notation:** Ranks before and after a substep are denoted r(v) and r′(v), respectively.

Proposition 13.6

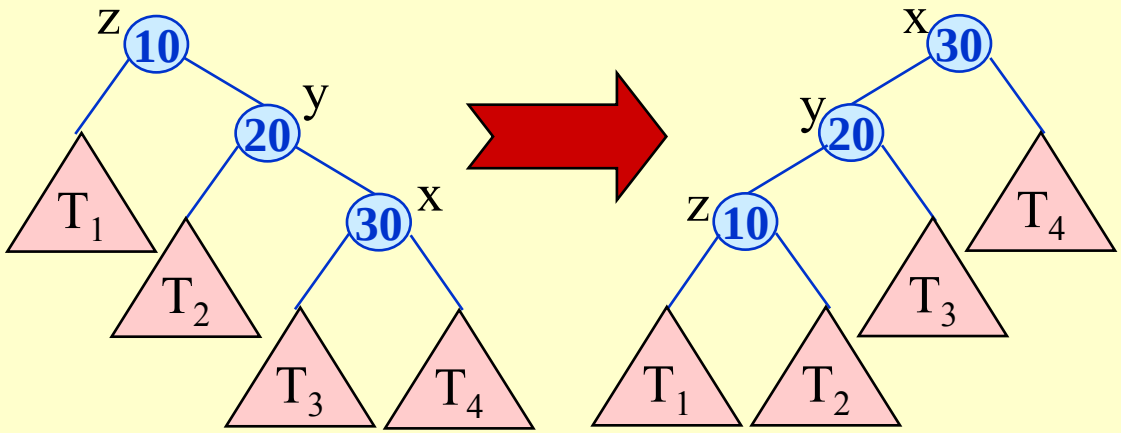
Proposition 13.6: Let δ be the change in r(T) caused by a single substep. Let x be the "x" in our descriptions of these substeps. Then,

- $\delta \leq 3(r'(x) - r(x)) - 2$ if the substep is a zig-zig or a zig-zag;
- $\delta \leq 3(r'(x) - r(x))$ if the substep is a zig.

Proof:

Three cases, one for each kind of substep…

Case 1: zig-zig



Only the ranks of x, y, and z change. Also, $r'(x) = r(z)$, $r'(y) \le r'(x)$, and $r(y) \ge r(x)$. Thus,

$$\begin{aligned}\delta &= r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z)\\ &= r'(y) + r'(z) - r(x) - r(y)\\ &\le r'(x) + r'(z) - 2r(x). \quad \textbf{(*)}\end{aligned}$$

Also, $n(x) + n'(z) \le n'(x)$, which (by property of lg), implies $r(x) + r'(z) \le 2r'(x) - 2$, i.e.,

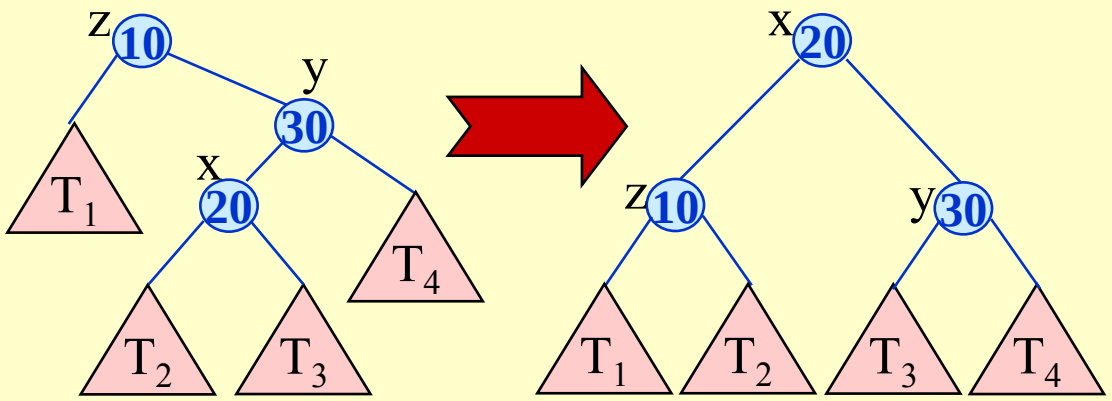
$$r'(z) \le 2r'(x) - r(x) - 2. \quad \textbf{(**)}$$

If $a > 0$, $b > 0$, and $c \ge a + b$, then $\lg a + \lg b \le 2 \lg c - 2$.

By (*) and (**), $\delta \le r'(x) + (2r'(x) - r(x) - 2) - 2r(x)$

$$= 3(r'(x) - r(x)) - 2.$$

Case 2: zig-zag



Only the ranks of x, y, and z change. Also, $r'(x) = r(z)$ and $r(x) \le r(y)$. Thus,

$$\begin{aligned}\delta &= r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z)\\ &= r'(y) + r'(z) - r(x) - r(y)\\ &\le r'(y) + r'(z) - 2r(x). \quad (*)\end{aligned}$$

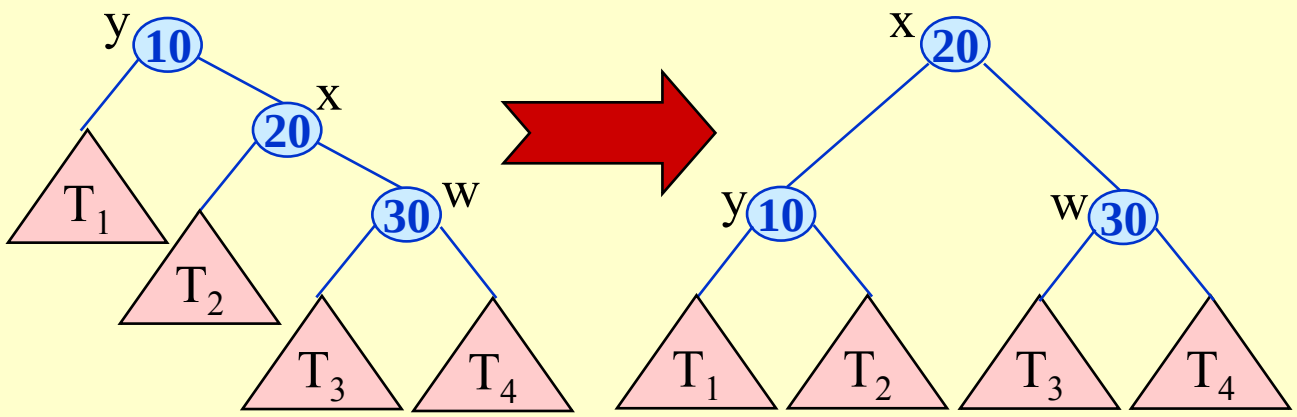
Also, $n'(y) + n'(z) \le n'(x)$, which (by property of lg), implies

$$r'(y) + r'(z) \le 2r'(x) - 2. \quad (**)$$

By (*) and (**), $\delta \le 2r'(x) - 2 - 2r(x)$

$$\le 3(r'(x) - r(x)) - 2.$$

Case 3: zig



Only the ranks of x and y change.
Also, $r'(y) \leq r(y)$ and $r'(x) \geq r(x)$. Thus,

$$\begin{aligned}\delta &= r'(x) + r'(y) - r(x) - r(y) \\ &\leq r'(x) - r(x) \\ &\leq 3(r'(x) - r(x)).\end{aligned}$$

Proposition 13.7

Proposition 13.7: Let T be a splay tree with root t, and let Δ be the total variation of r(T) caused by splaying a node x at depth d. Then,

$$\Delta \leq 3(r(t) - r(x)) - d + 2.$$

Proof:

Splay(x) consists of $p = \lceil d/2 \rceil$ substeps, each of which is a zig-zig or zig-zag, except possibly the last one, which is a zig if d is odd.

Let $r_0(x)$ = x's initial rank, $r_i(x)$ = x's rank after the i^{th} substep, and δ_i = the variation of r(T) caused by the i^{th} substep, where $1 \leq i \leq p$.

By Proposition 13.6,

$$\Delta = \sum_{i=1}^{p} \delta_i \leq \sum_{i=1}^{p} \left(3(r_i(x) - r_{i-1}(x)) - 2\right) + 2$$

$$= 3(r_p(x) - r_0(x)) - 2p + 2$$

$$\leq 3(r(t) - r(x)) - d + 2$$

Meaning of Proposition

- If d is small (less than $3(r(t) - r(x)) + 2$) then the splay operation can increase r(t) and thus make the tree less balanced.
- If d is larger than this, then the splay operation decreases r(t) and thus makes the tree better balanced.
- Note that $r(t) \leq \lg(2n + 1)$

Amortized Costs

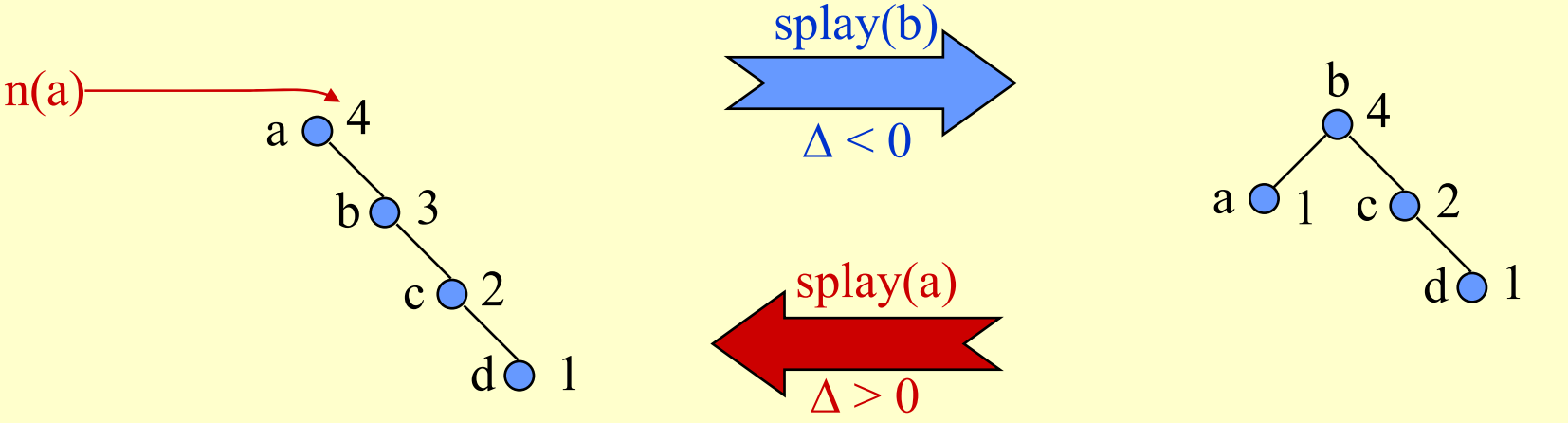
- As stated before, each operation's amortized cost = its real cost + the total change in r(T) it causes, i.e., Δ.
  - This ensures the Credit Invariant isn't violated.
- Real cost is d, so amortized cost is $d + \Delta$.
- The real cost of d even includes the cost of binary tree operations such as searching.
- **Note:** Δ can be positive or negative (or zero).
  - If it's positive, we're overcharging.
  - If it's negative, we're undercharging.

Another Look at Δ

Δ = the total change in r(T). $r(T) = \sum_{v \in T} r(v) = \sum_{v \in T} \lg(n(v))$

$$= \lg\left(\prod_{v \in T} n(v)\right)$$

Consider this example:



$r(T) = \lg(4 \cdot 3 \cdot 2 \cdot 1)$
$= \lg(24)$

$r(T) = \lg(4 \cdot 2 \cdot 1 \cdot 1)$
$= \lg(8)$

Unbalancing the Tree

- In fact, a sequence of zig operations can result in a completely unbalanced linear tree. Then a search operation can take O(n) time, but this is OK because at least n operations have been performed up to this point.

A Bound on Amortized Cost

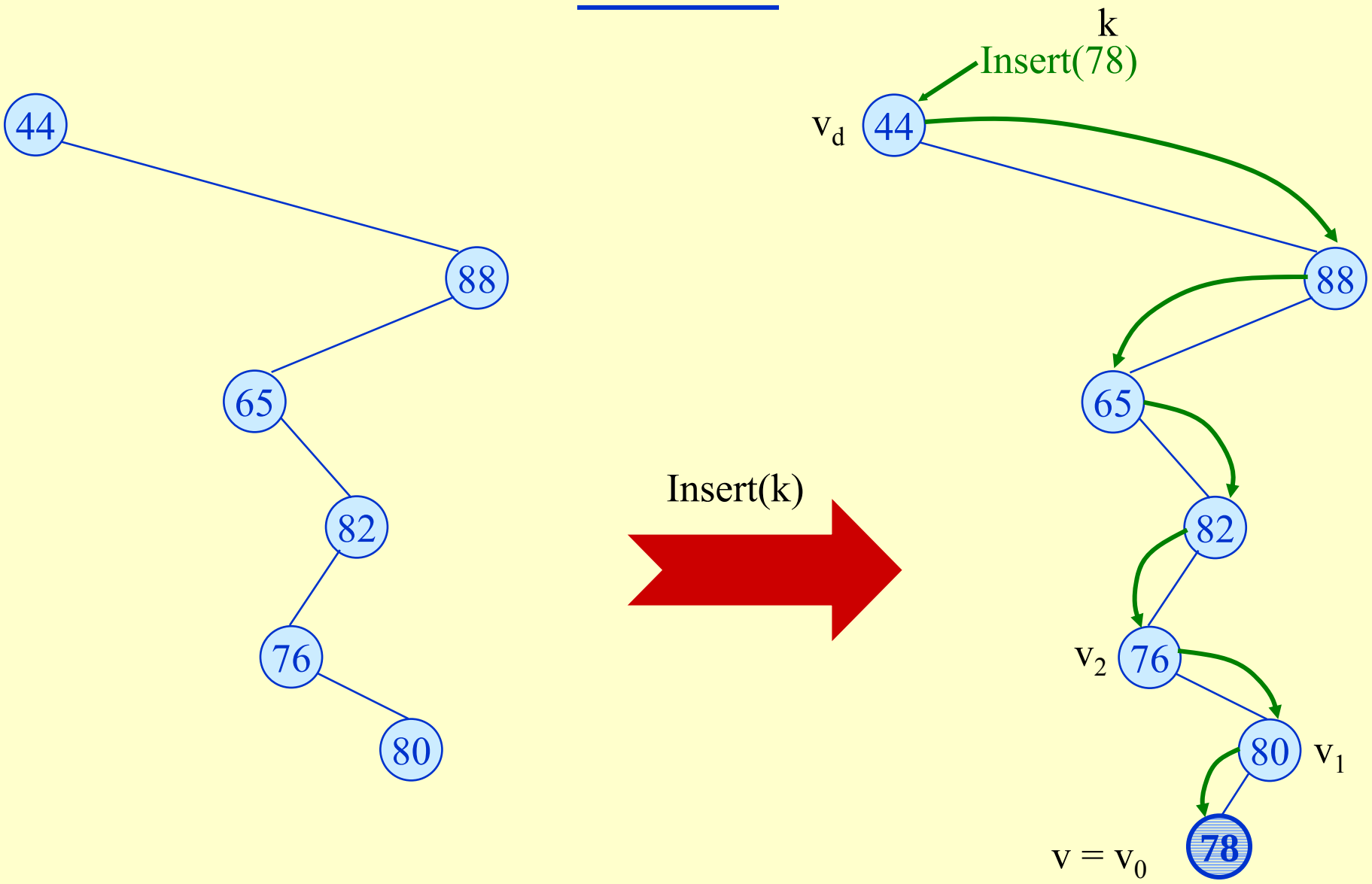
We have:

Amortized Cost of Splaying
$= d + \Delta$
$\leq d + (3(r(t) - r(x)) - d + 2)$ {Prop. 13.7}
$= 3(r(t) - r(x)) + 2$
$< 3r(t) + 2$
$= 3\lg(2n + 1) + 2$ {Recall t is the root}
$= O(\lg n)$

Finishing Up

- Until now, we've just focused on splaying costs.
- We also need to ensure that BST operations can be charged in a way that maintains the Credit Invariant.
- Three Cases:
  - **Search:** Not a problem — doesn't change the tree.
  - **Delete:** Not a problem — removing a node can only decrease ranks, so existing credits are still fine.
  - **Insert:** As shown next, an Insert can cause r(T) to increase by up to lg(2n+3) + lg 3. Thus, the Credit Invariant can be maintained if Insert is assessed an O(lg n) charge.

Insert


44
88
65
82
76
80
Insert(k)
k
Insert(78)
v_d
44
88
65
82
v_2
76
80
v_1
78
$v = v_0$

Insert

Lots of typos in book!

For i = 1, …, d, let $n(v_i)$ and $n'(v_i)$ be sizes before and after insertion, and $r(v_i)$ and $r'(v_i)$ be ranks before and after insertion.

We have: $n'(v_i) = n(v_i) + 2$.

Leaf gets replaced by "real" node and two leaves.

For i = 1, …, d – 1, $n(v_i) + 2 \le n(v_{i+1})$, and

$r'(v_i) = \lg(n'(v_i)) = \lg(n(v_i) + 2) \le \lg(n(v_{i+1})) = r(v_{i+1})$.

Subtree at v_i doesn't include v_{i+1} and its "other" child.

Thus, $\sum_{i=1..d} (r'(v_i) - r(v_i)) \le r'(v_d) - r(v_d) + \sum_{i=1..d-1}(r(v_{i+1}) - r(v_i))$

$= r'(v_d) - r(v_d) + r(v_d) - r(v_1)$

$\le \lg(2n + 3)$.

Note: v_0 is excluded here – it doesn't have an old rank! It's new rank is lg 3.

Thus, the Credit Invariant can be maintained if Insert is assessed a charge of at most $\lg(2n + 3) + \lg 3$.

For the insert operation, we perform a normal BST insert followed by a splay operation on the node inserted. Assume node x is inserted at depth k. Denote the parent of x as y_1, y_1's parent as y_2, and so on (the root of the tree is y_k). Then the change in potential due to the insertion of x is (r is rank before the insertion and r' is rank after the insertion, s is weight sum before the insertion):

$$
\begin{aligned}
\Delta\phi &= \sum_{j=1}^{k} (r'(y_j) - r(y_j)) \\
&= \sum_{j=1}^{k} (\log(s(y_j)+1) - \log(s(y_j)) \\
&= \sum_{j=1}^{k} \log\left(\frac{s(y_j)+1}{s(y_j)}\right) \\
&= \log\left(\prod_{j=1}^{k} \frac{s(y_j)+1}{s(y_j)}\right) \text{ (note that } s(y_j)+1 \le s(y_{j+1})) \\
&\le \log\left(\frac{s(y_2)}{s(y_1)} \cdot \frac{s(y_3)}{s(y_2)} \cdots \frac{s(y_k)}{s(y_{k-1})} \cdot \frac{s(y_k)+1}{s(y_k)}\right) \\
&= \log\left(\frac{s(y_k)+1}{s(y_k)}\right) \\
&\le \log n
\end{aligned}
$$

2012.12.21 not The End.