Algorithms

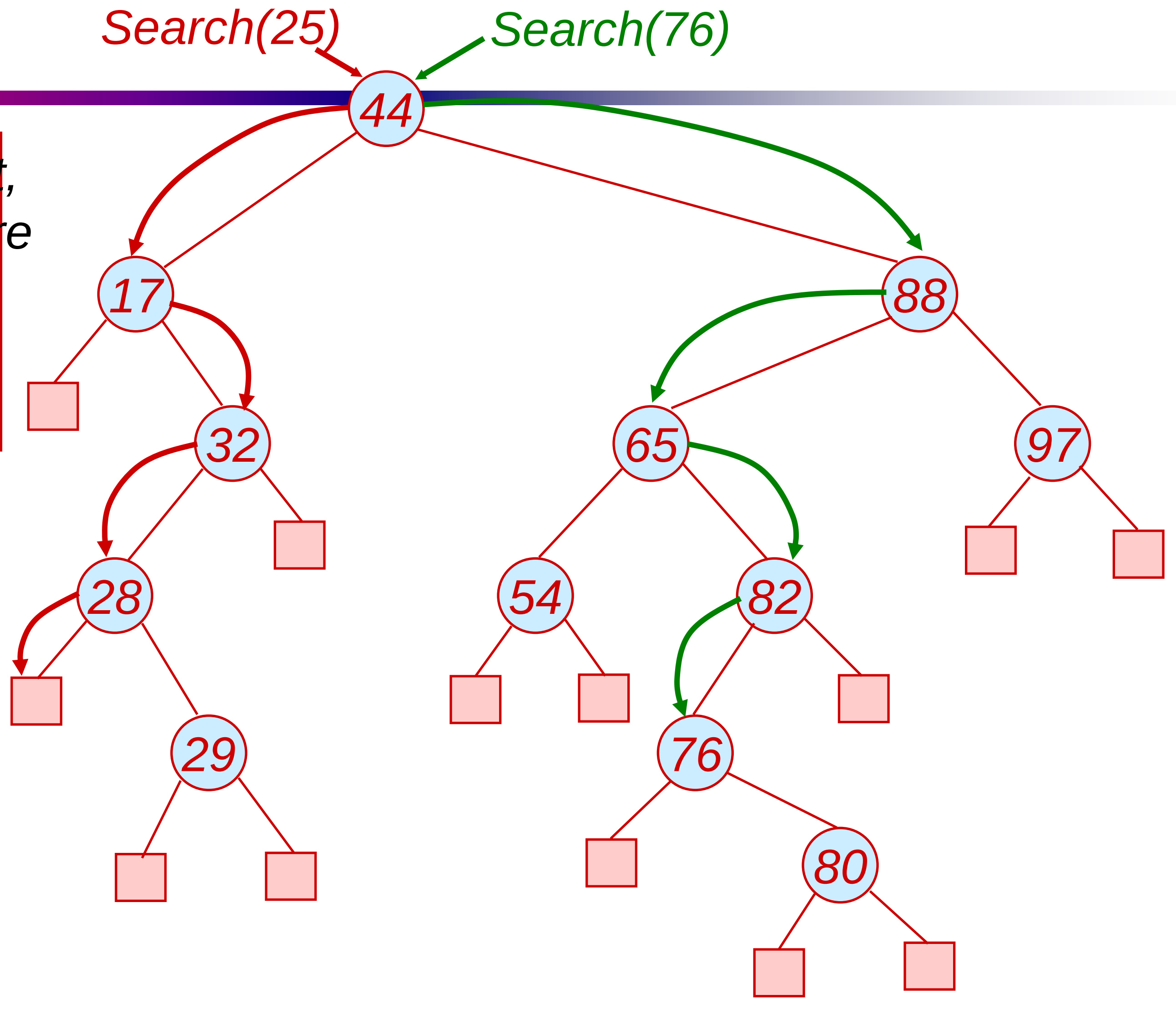
Splay Trees

Splay Trees

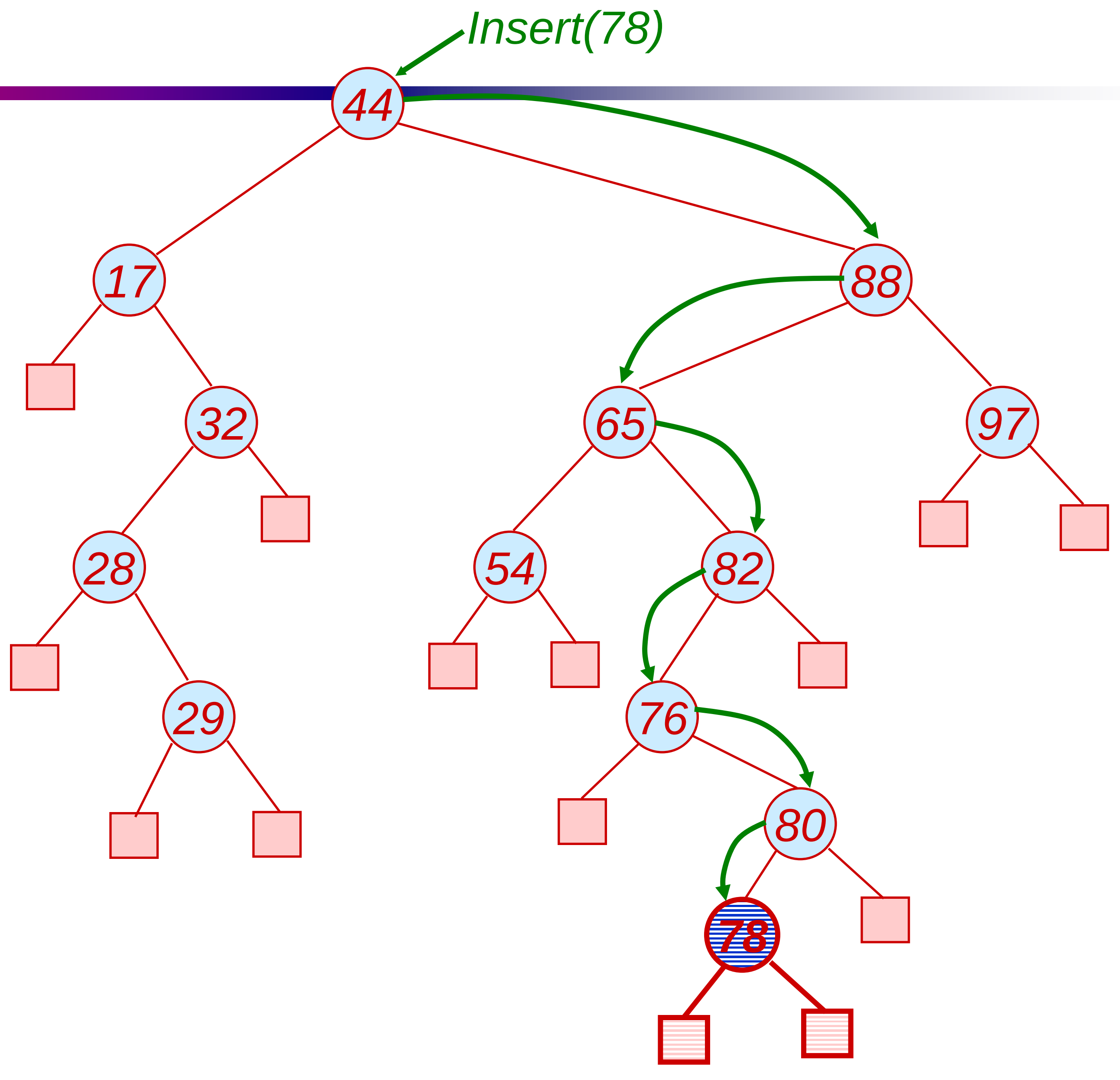
- In balanced tree schemes, explicit rules are followed to ensure balance.
- In splay trees, there are no such rules.
- Search, insert, and delete operations are like in binary search trees, except at the end of each operation a special step called **splaying** is done.
- Splaying ensures that all operations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

BST: Search

Note: *In the handout, sentinel leaf nodes are assumed.*
$\Rightarrow$ tree with n keys has 2n+1 nodes.

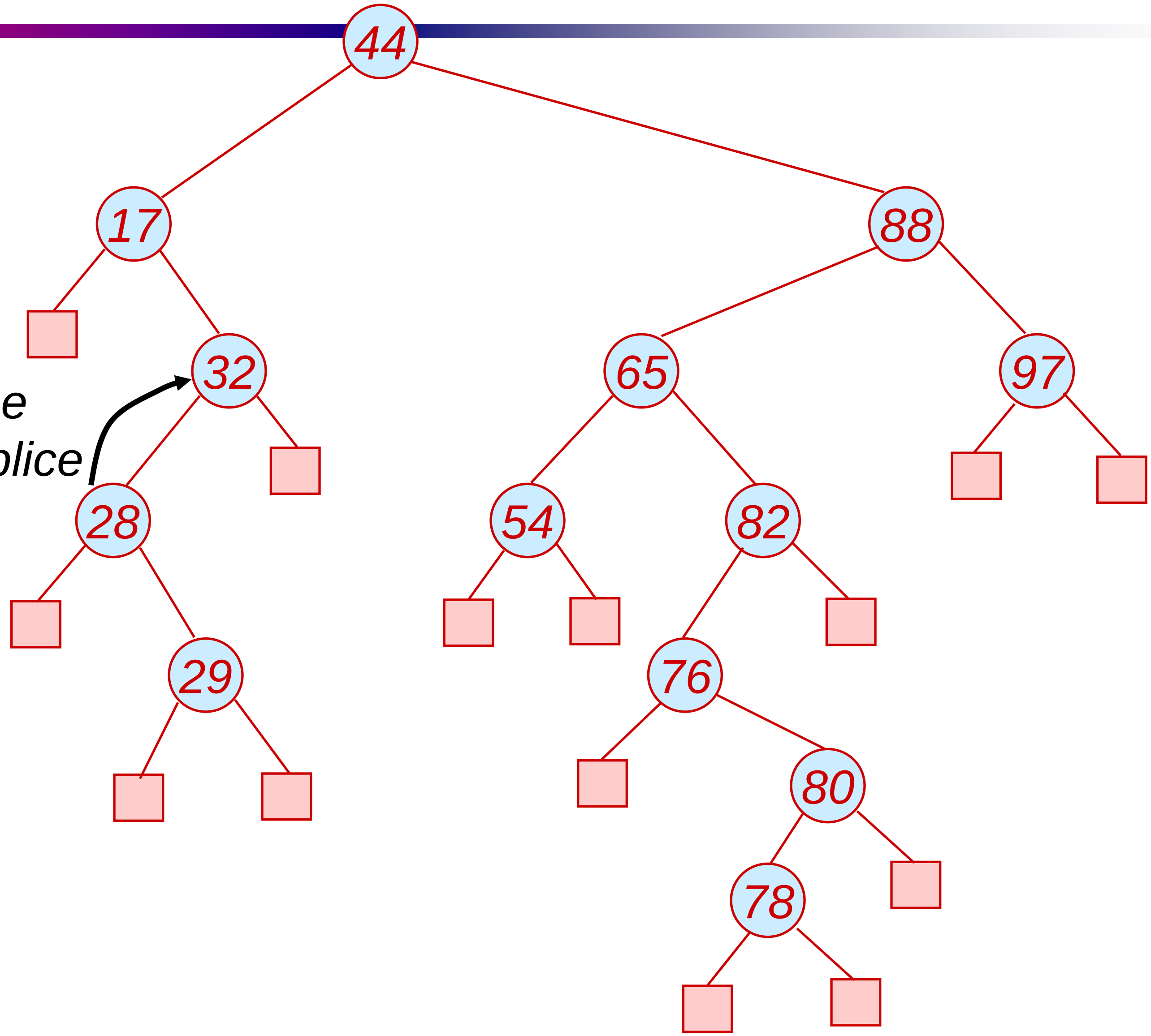


BST: Insert



BST: Delete

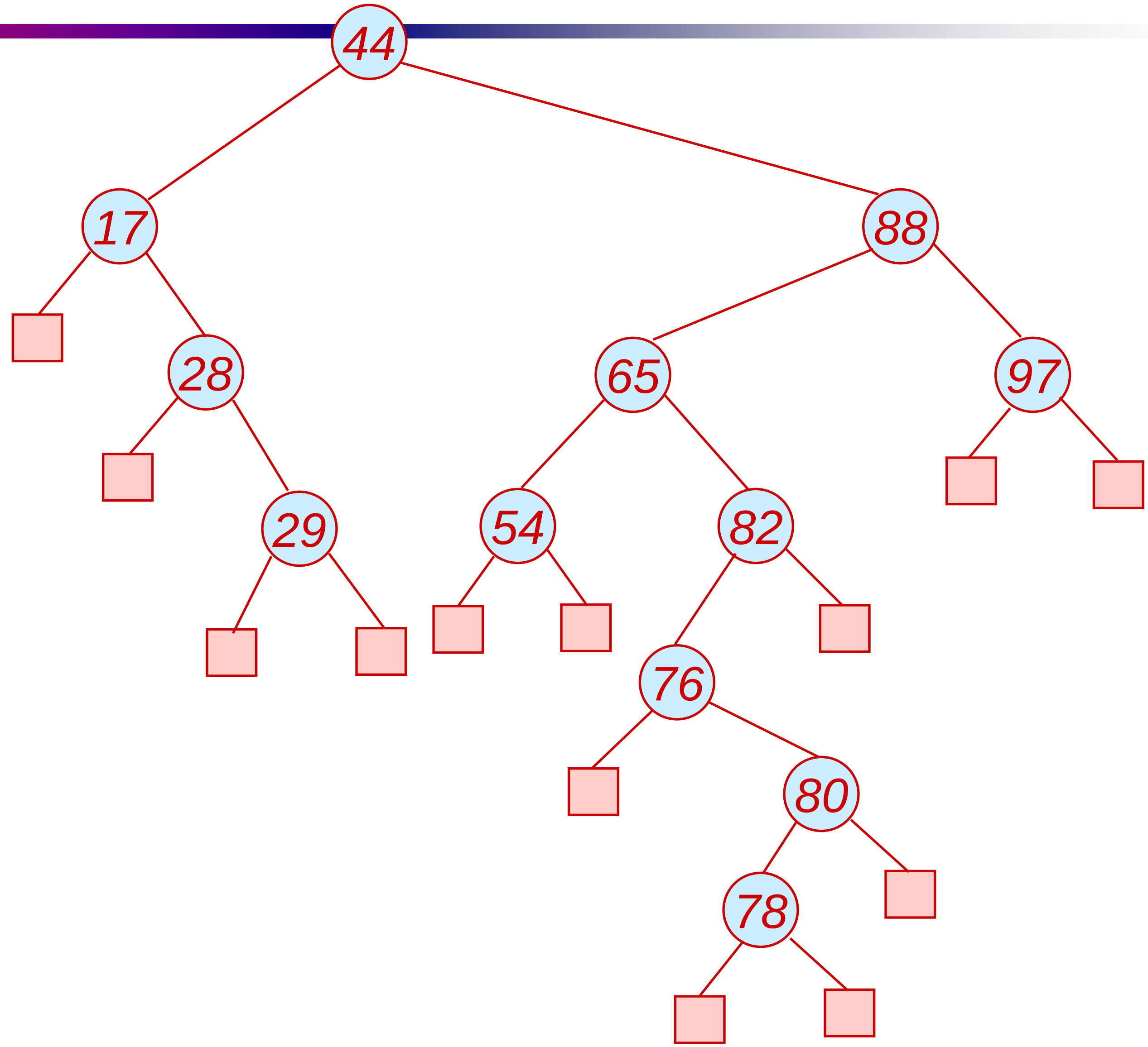
Delete(32)



Has only one child: just splice out 32.

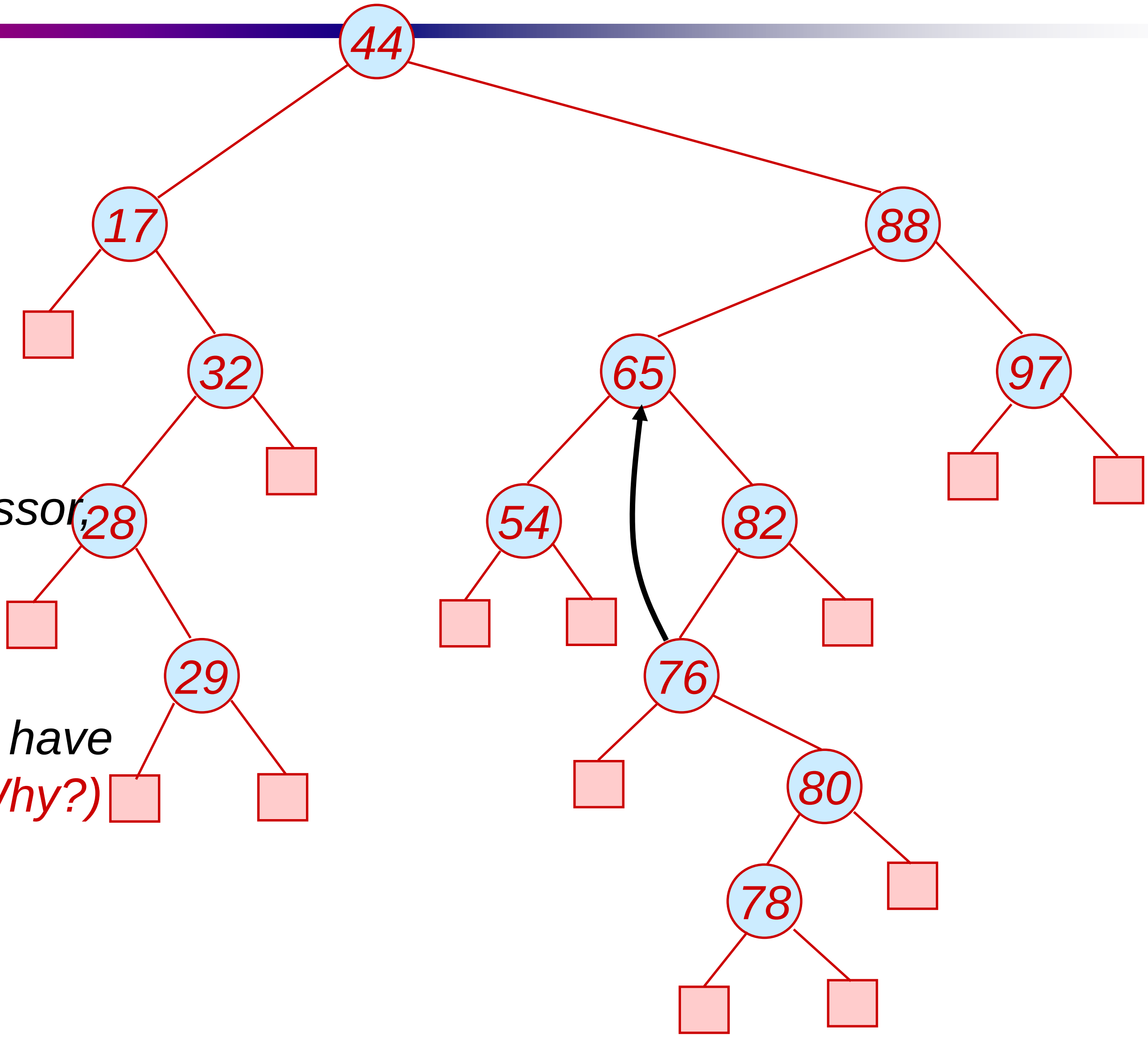
BST: Delete

Delete(32)



BST: Delete

Delete(65)

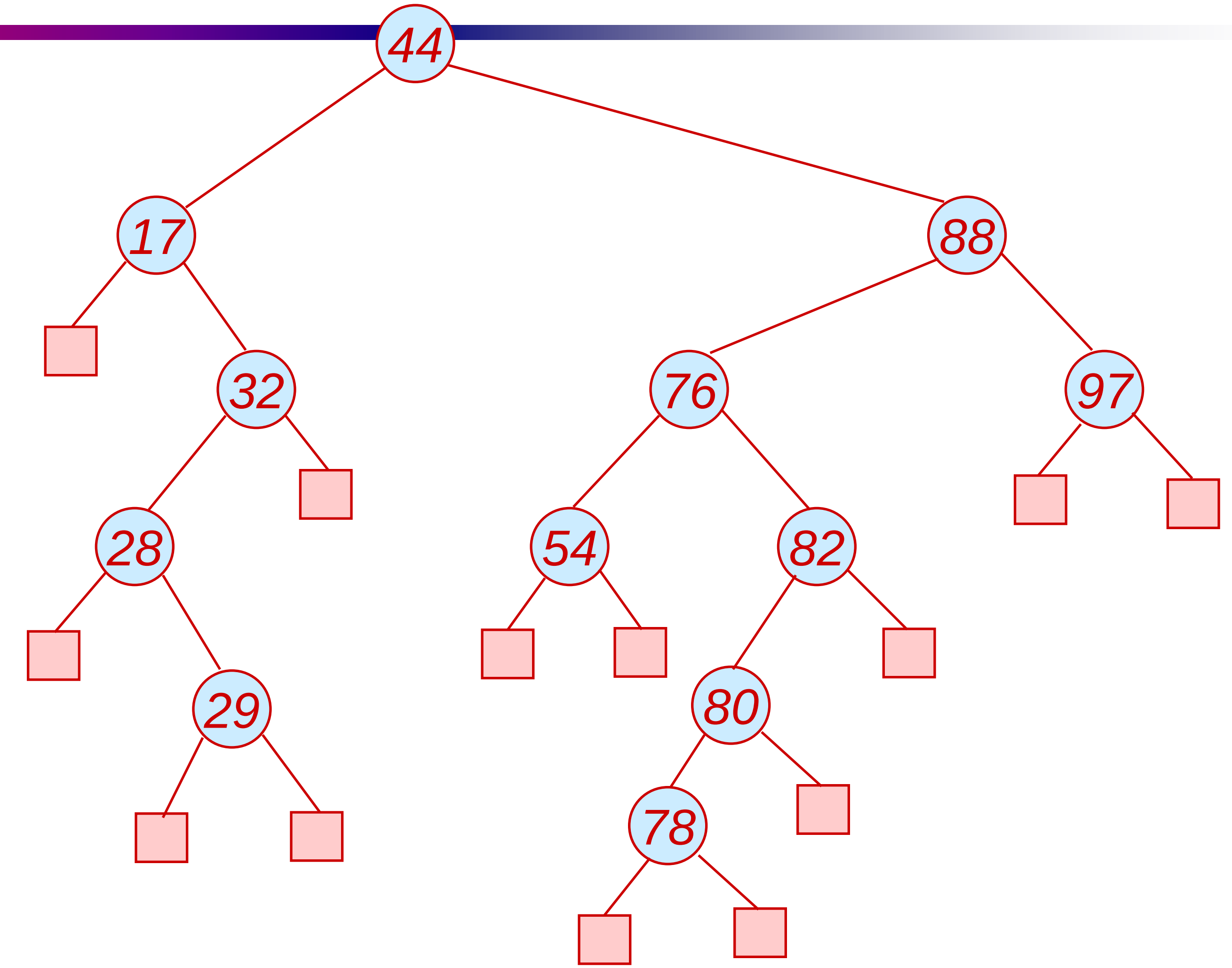


Has two children: Replace 65 by successor, 76, and splice out successor.

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

BST: Delete

Delete(65)

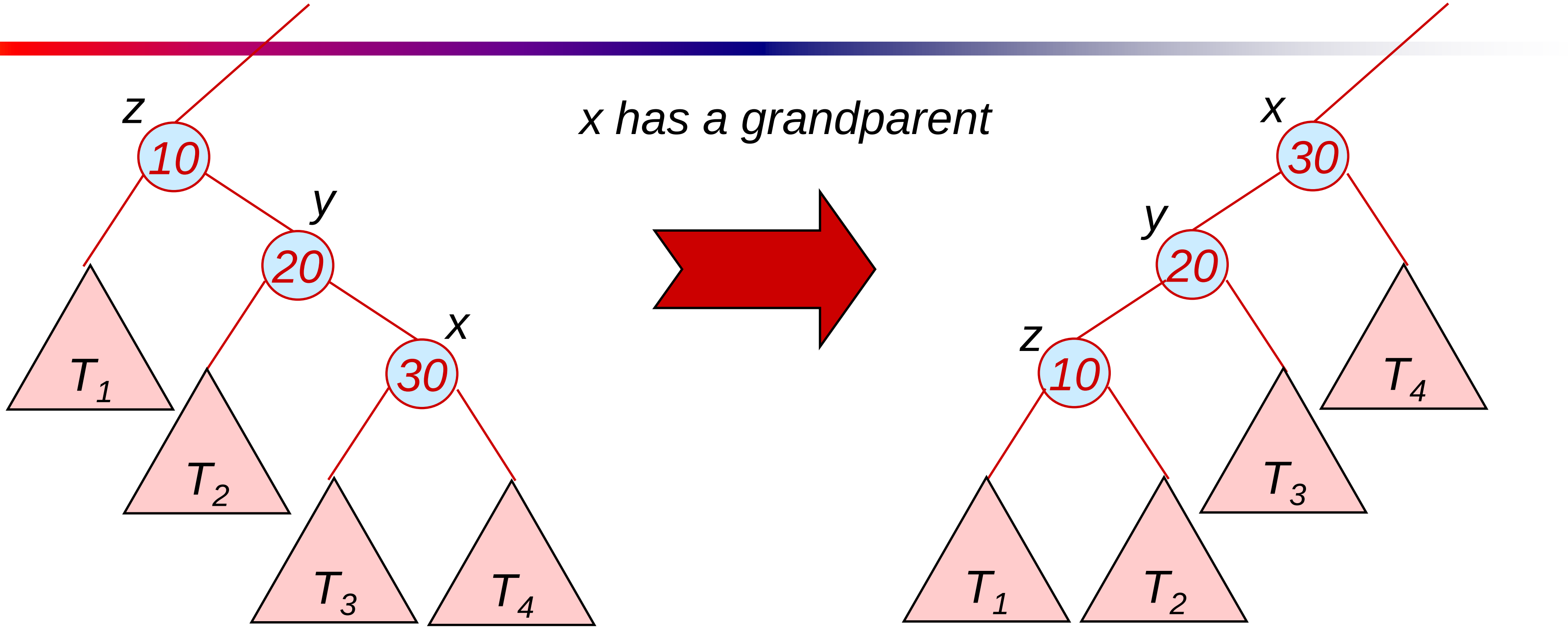


Splaying

- 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

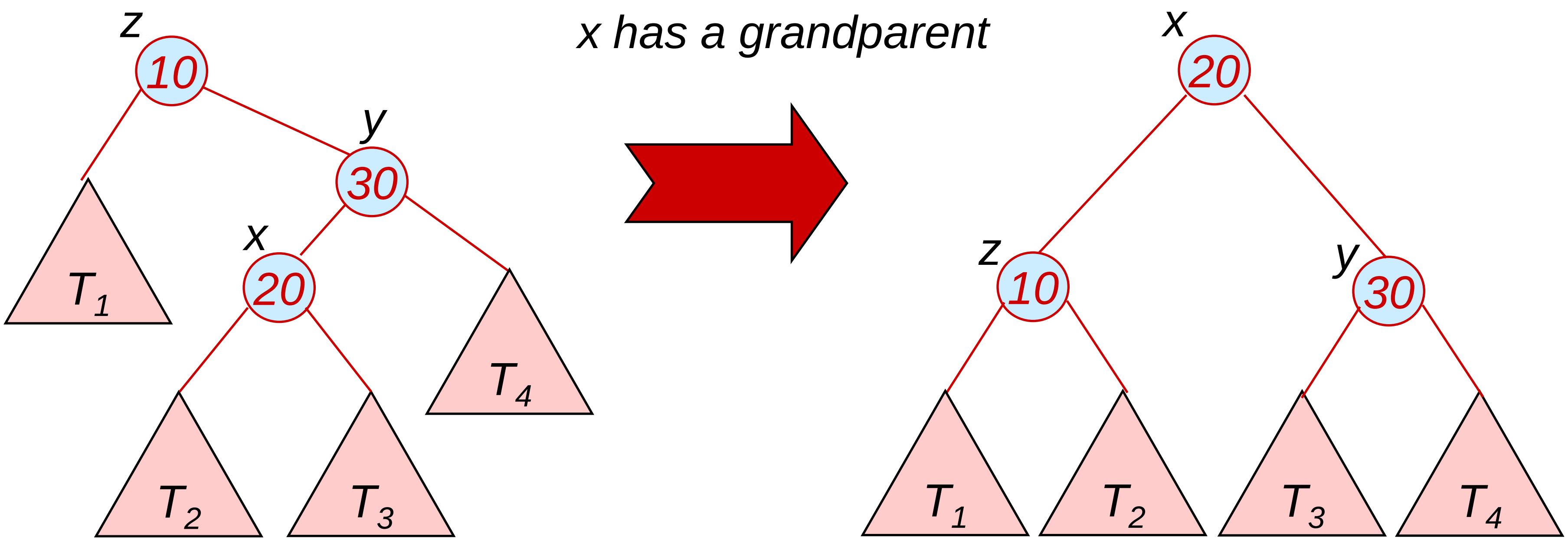
x has a grandparent



(Symmetric case too)

Note: *x's depth decreases by two.*

Zig-Zag



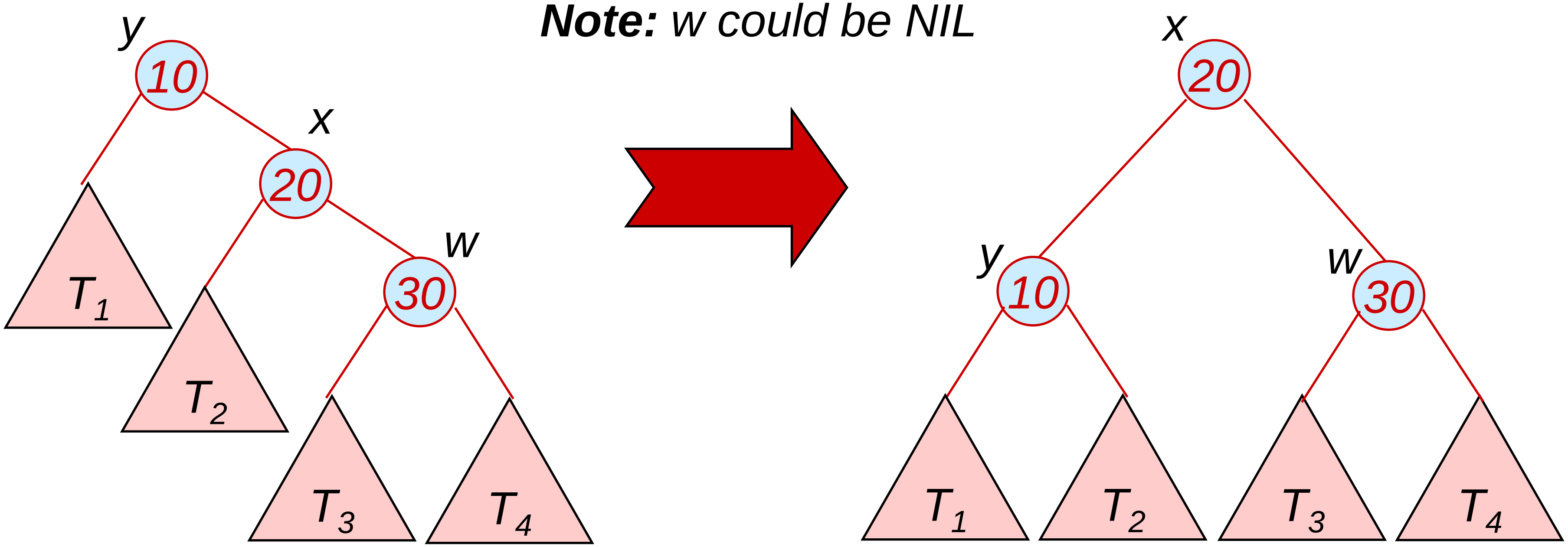
(Symmetric case too)

Note: *x's depth decreases by two.*

Zig

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

Note: *w could be NIL*



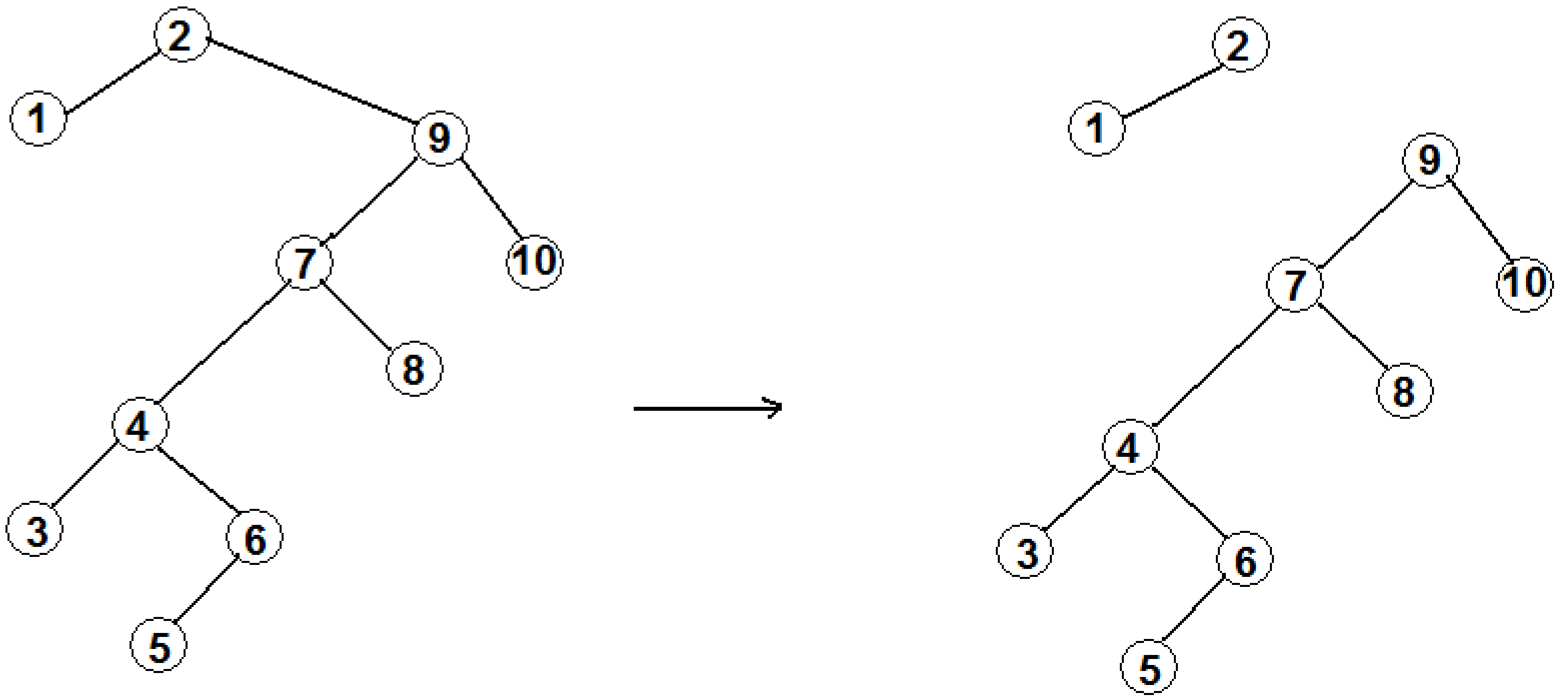
(Symmetric case too)

Note: *x's depth decreases by one.*

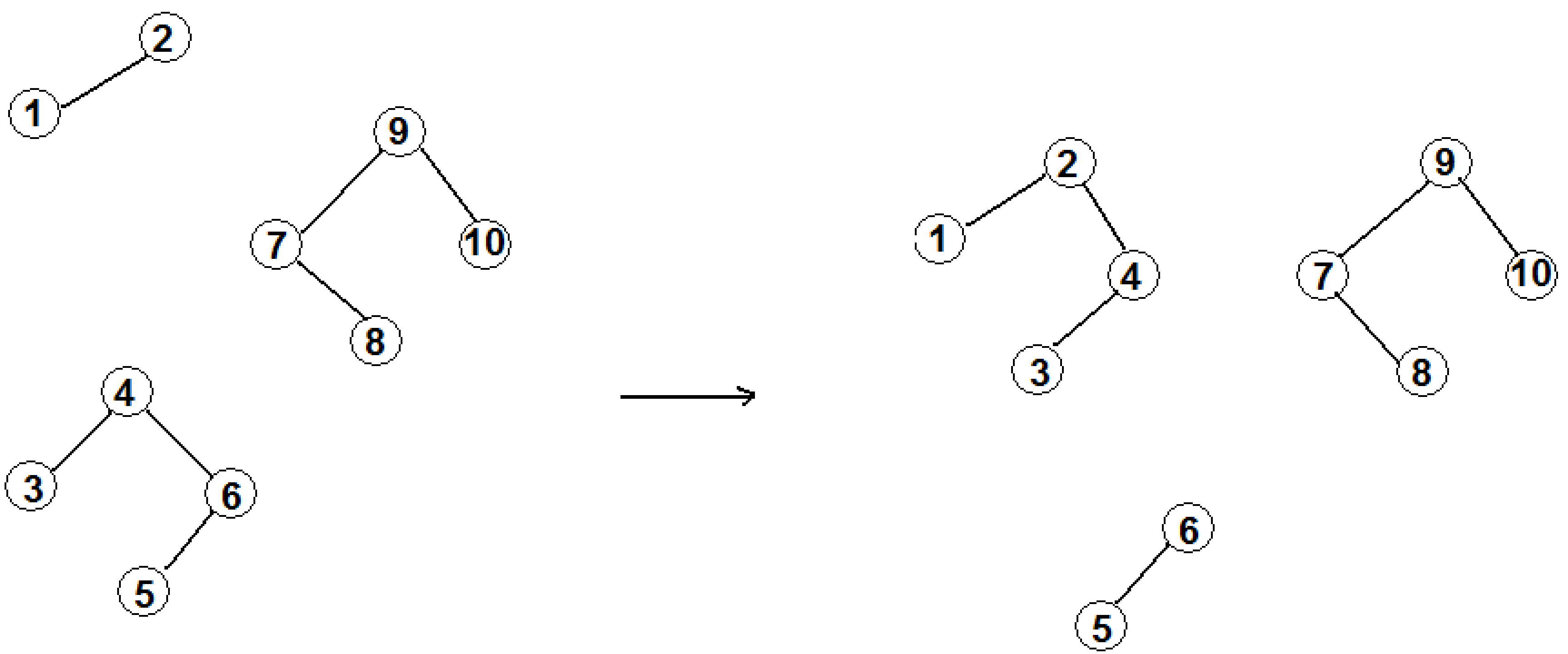
Top Down Splay Trees

- Works by starting at the top and working down to produce a similar result.
- All of the nodes lower than the target are put into one tree and all of the nodes greater than the target are put into another tree then it is recombined.

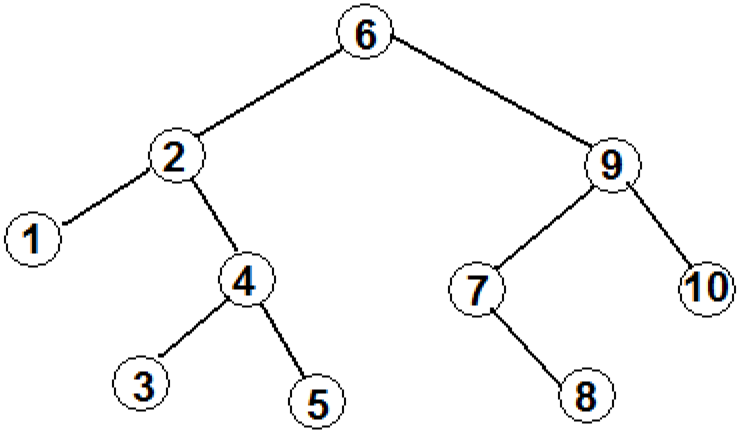
Top Down Splaying



Top Down Splaying



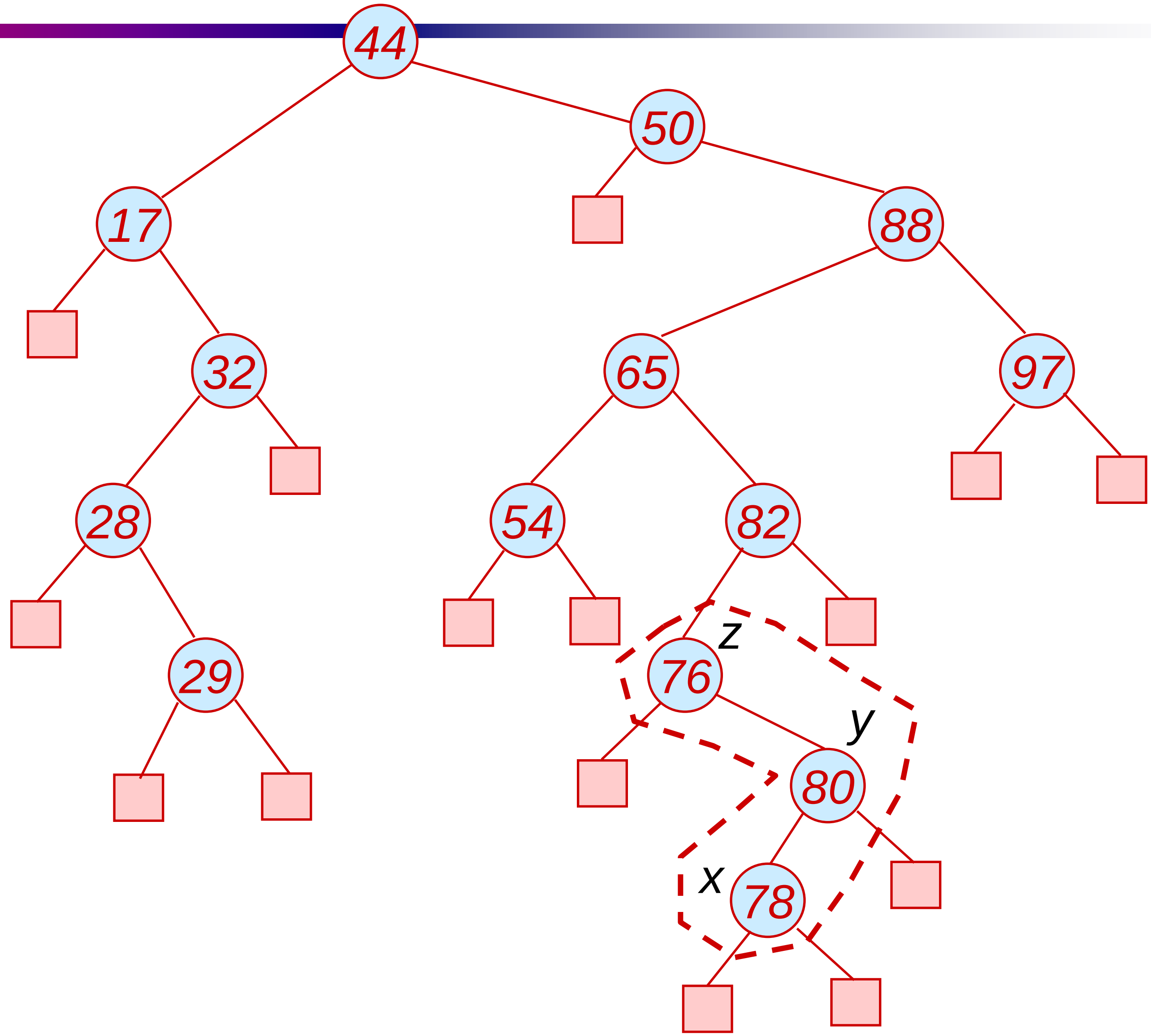
Top Down Splaying



Complete Example

Splay(78)

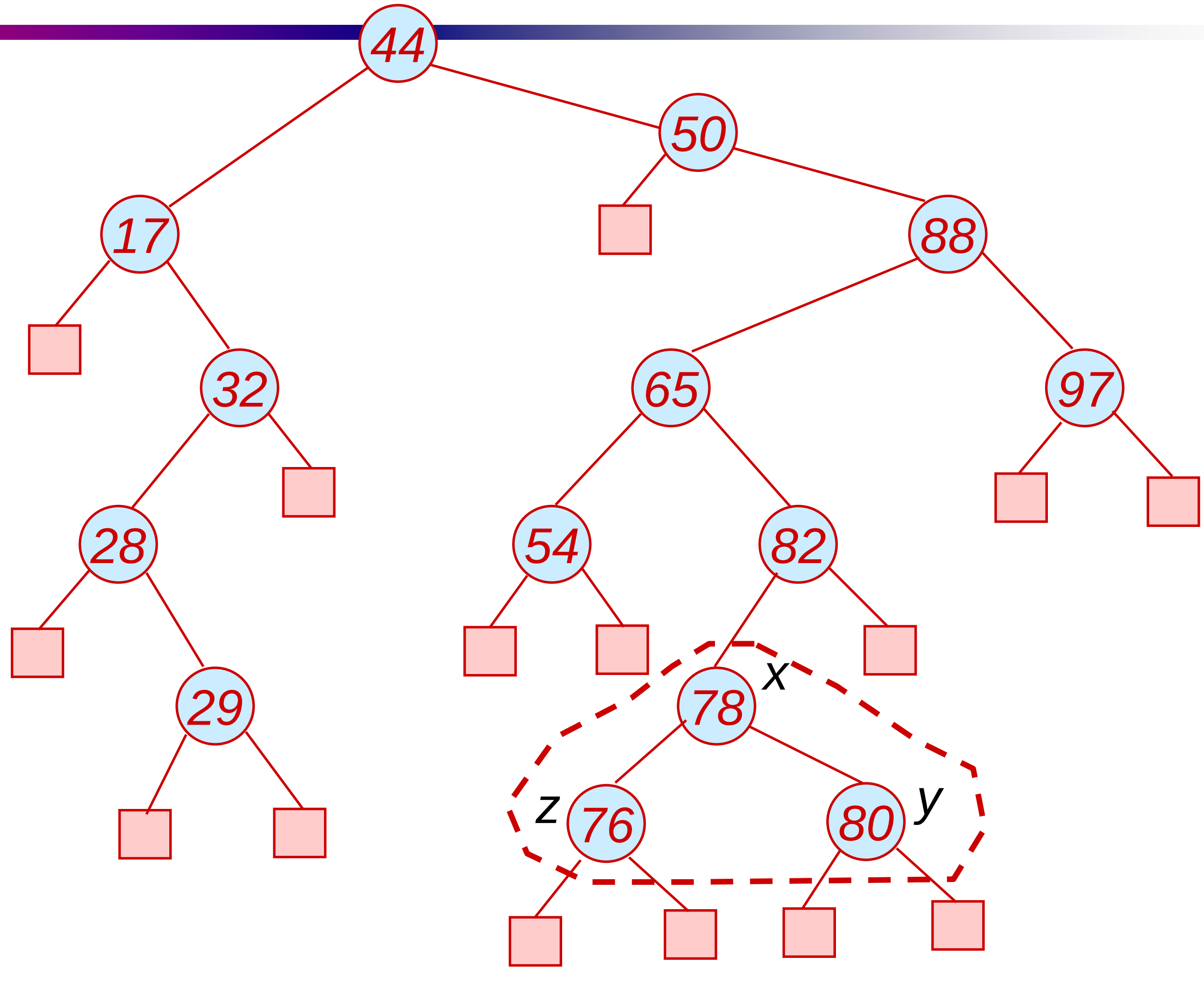
zig-zag



Complete Example

Splay(78)

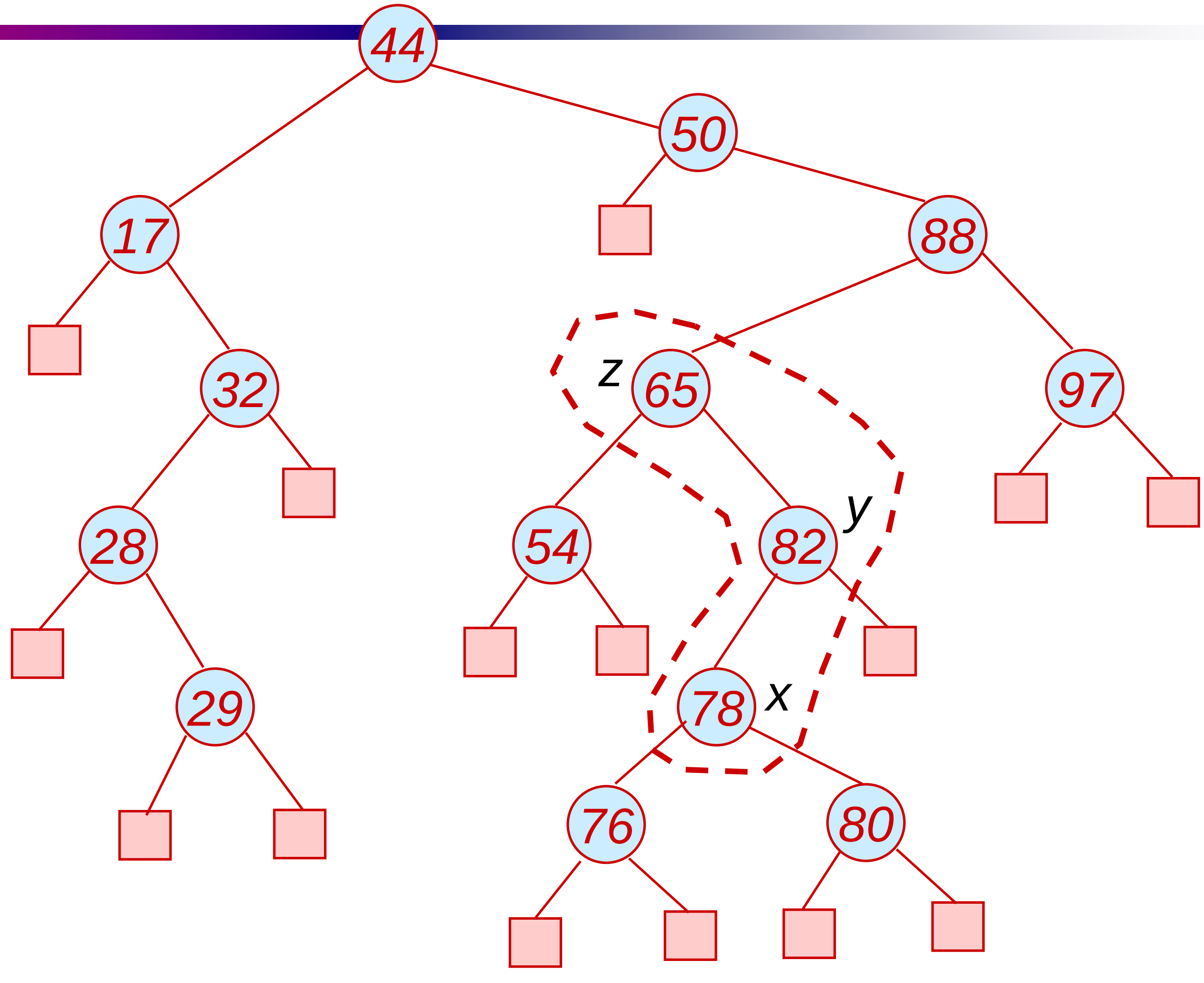
zig-zag



Complete Example

Splay(78)

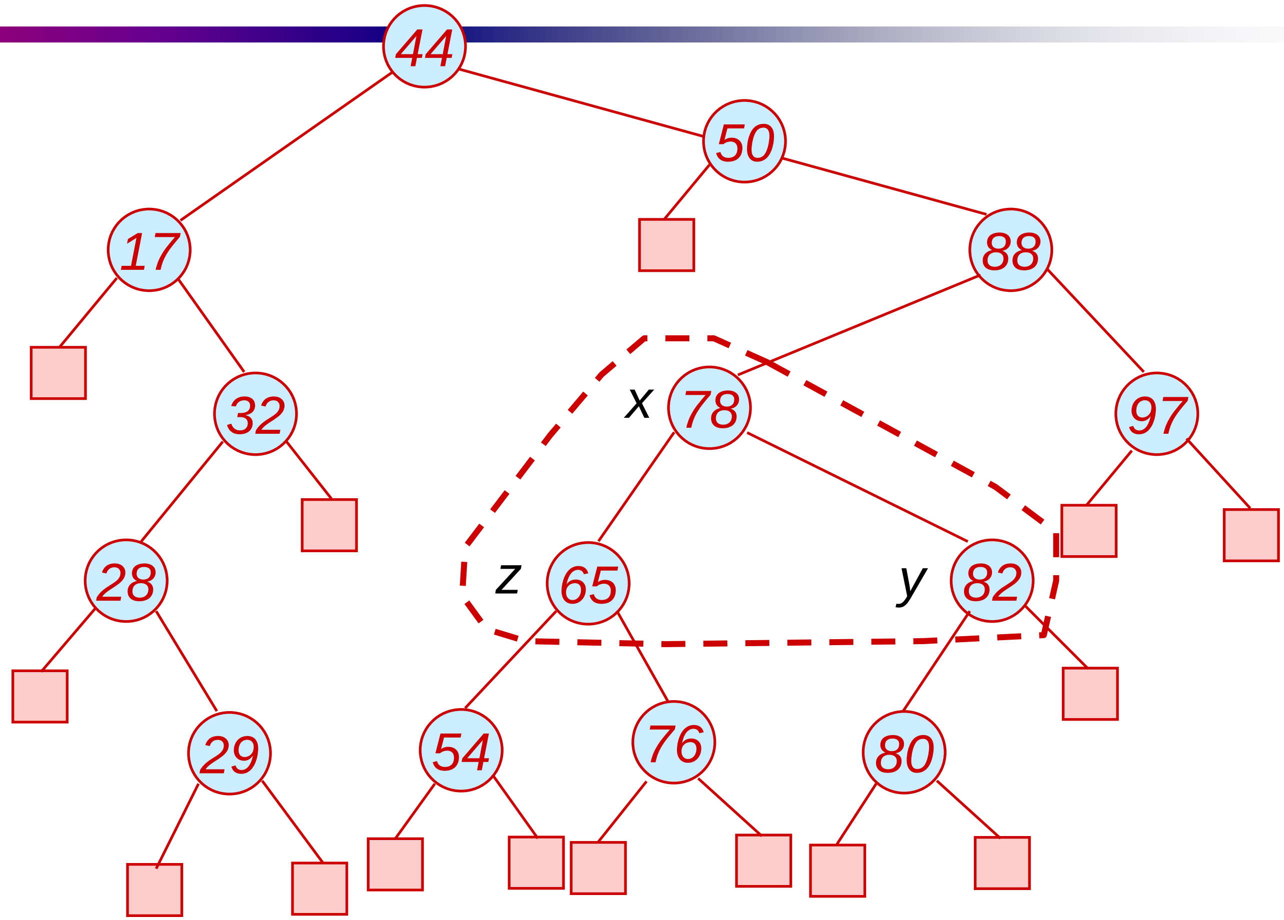
zig-zag



Complete Example

Splay(78)

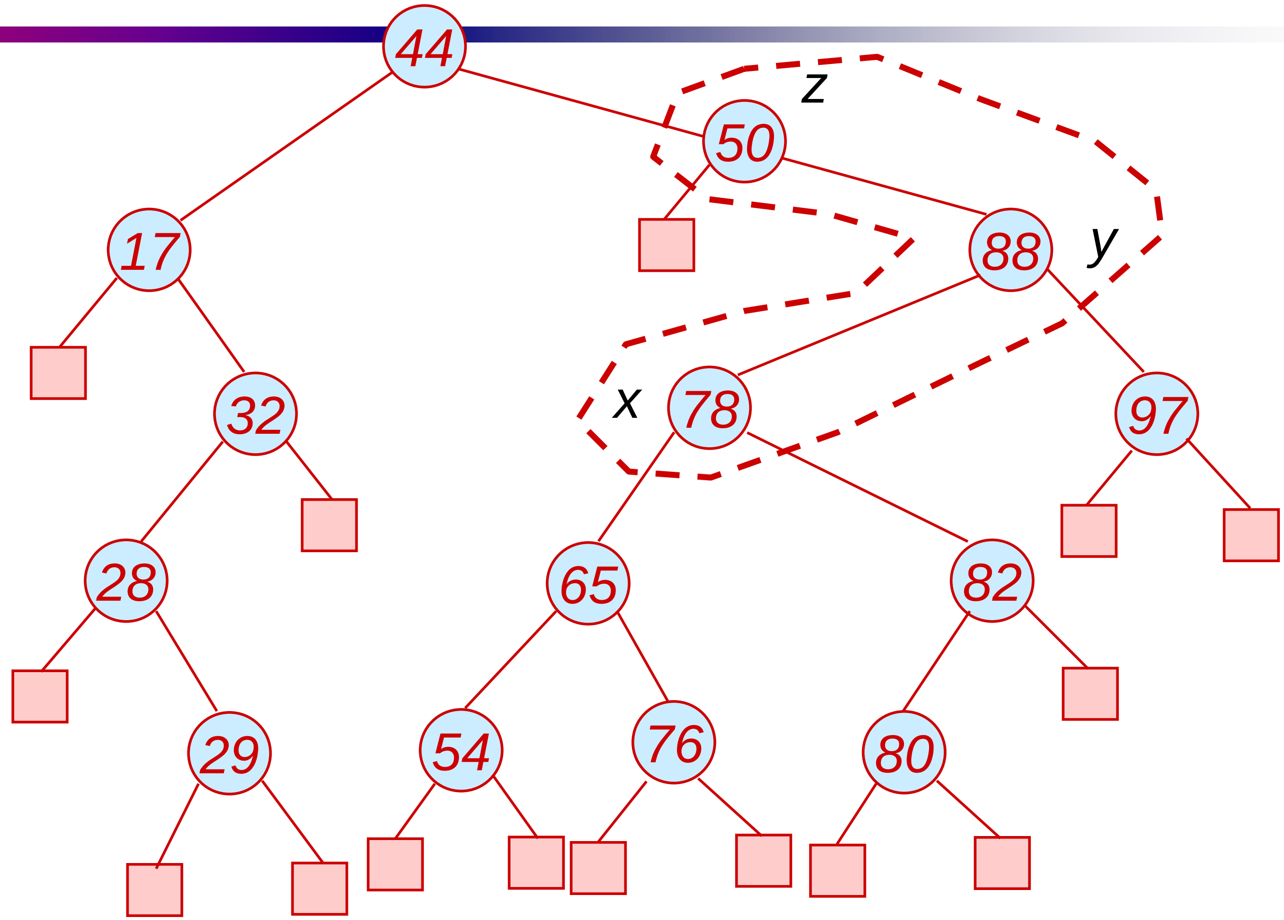
zig-zag



Complete Example

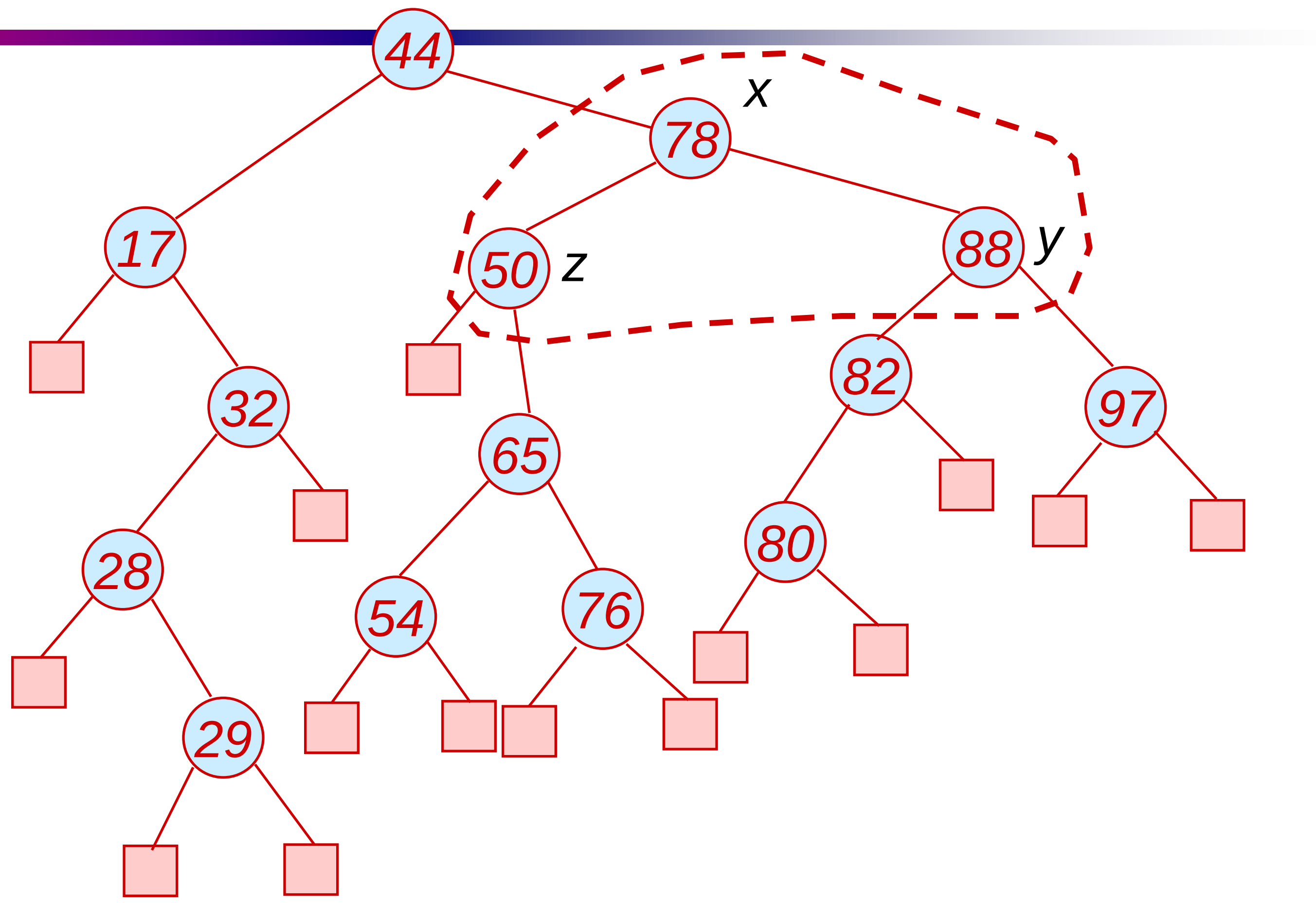
Splay(78)

zig-zag



Complete Example

Splay(78)

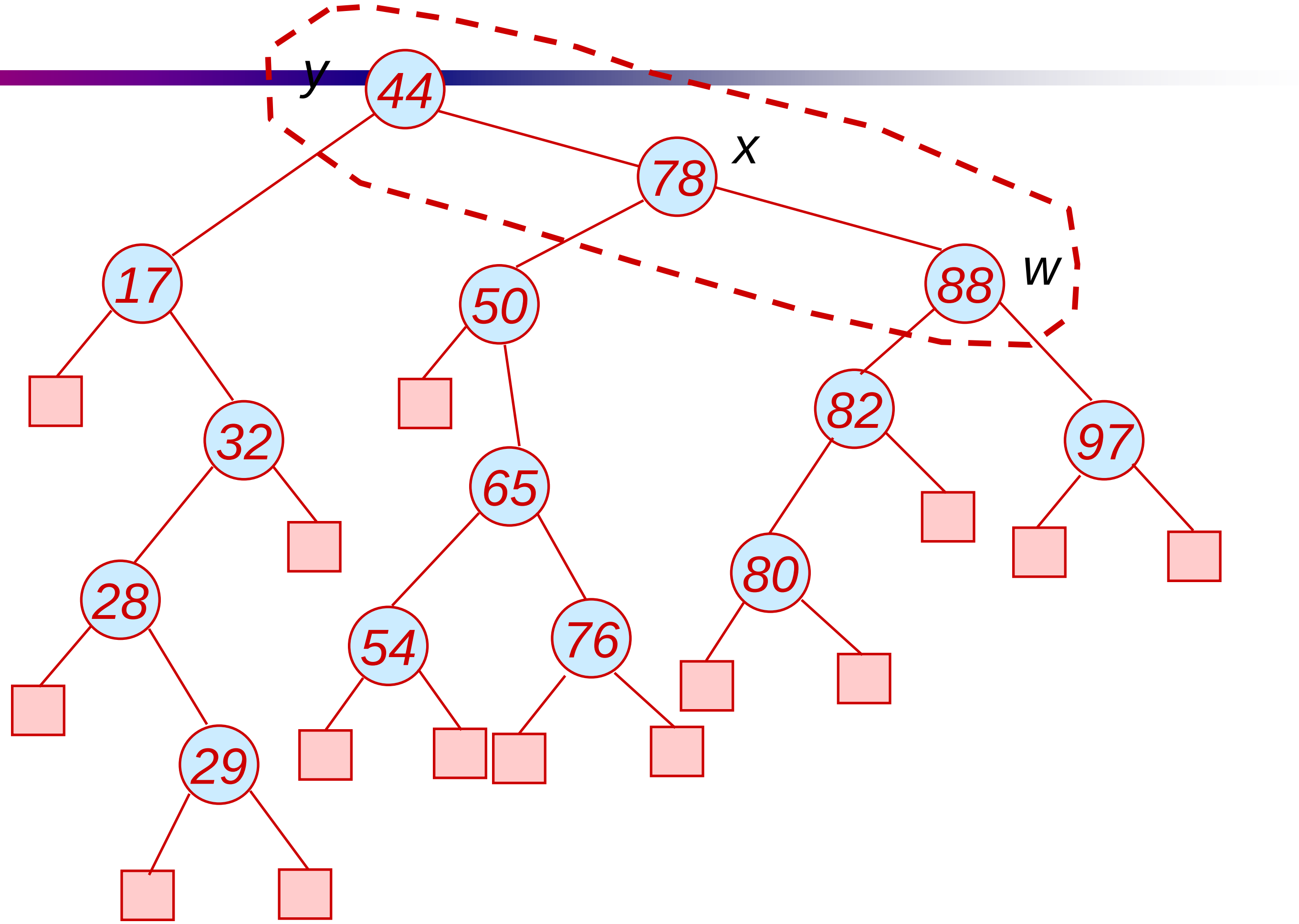


zig-zag

Complete Example

Splay(78)

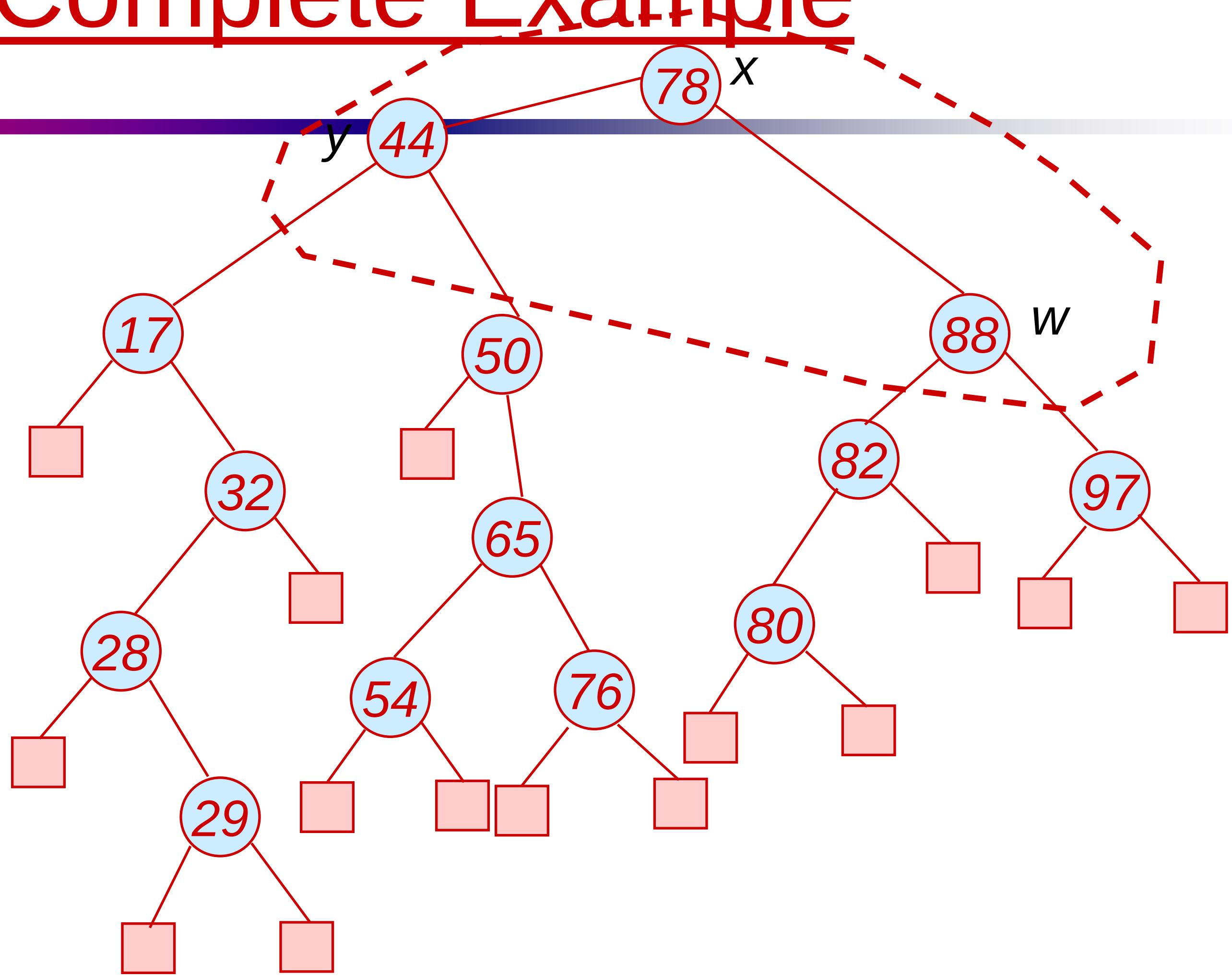
zig



Complete Example

Splay(78)

zig



Result of splaying

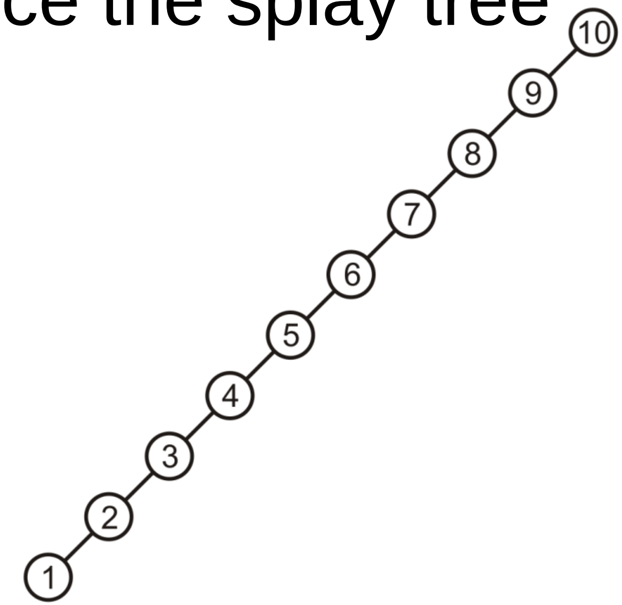
- The result is a binary tree, with the left subtree having all keys less than the root, and the right subtree having keys greater than the root.
- Also, the final tree is “more balanced” than the original.
- However, if an operation near the root is done, the tree can become less balanced.

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.

Examples

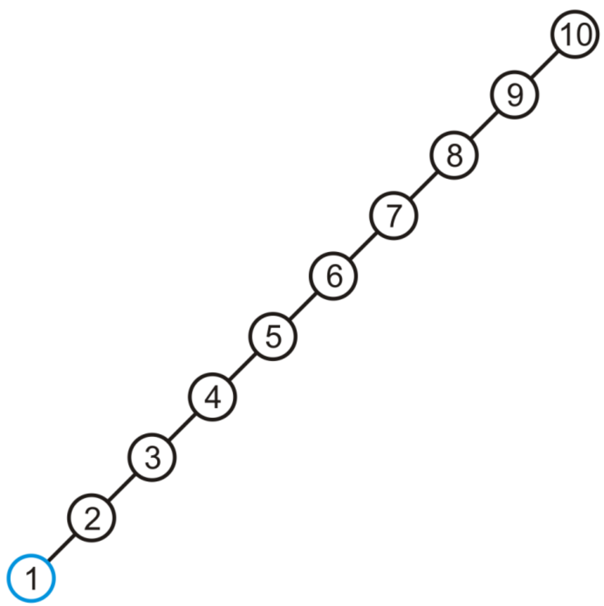
With a little consideration, it becomes obvious that inserting 1 through 10, in that order, will produce the splay tree



Examples

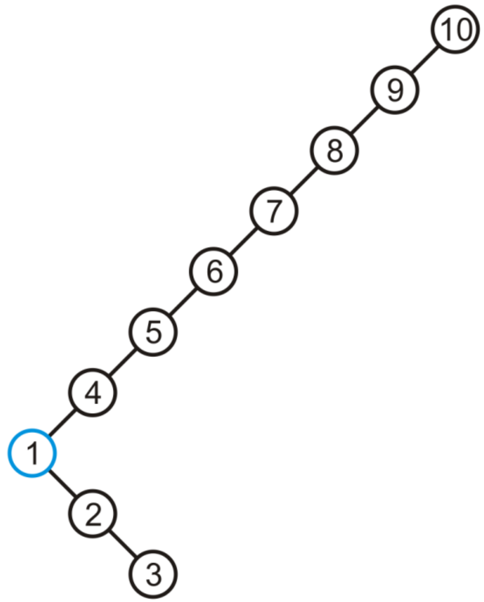
We will repeatedly access the deepest node in the tree

- With each operation, this node will be splayed to the root
- We begin with a zig-zig rotation



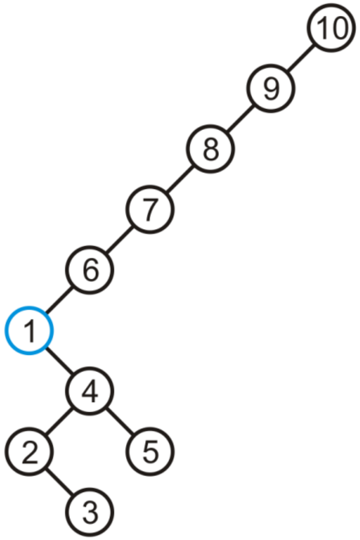
Examples

This is followed by another zig-zig operation...



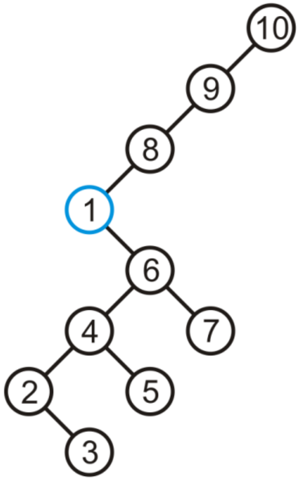
Examples

...and another



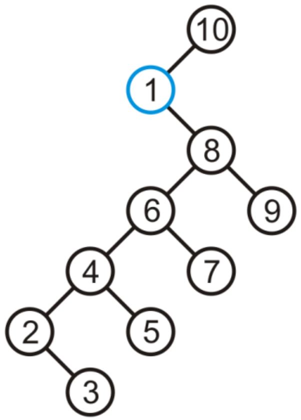
Examples

...and another



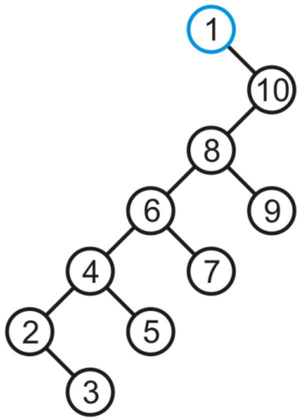
Examples

At this point, this requires a single zig operation to bring 1 to the root



Examples

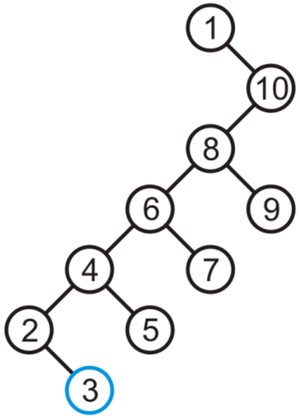
The height of this tree is now 6 and no longer 9



Examples

The deepest node is now 3:

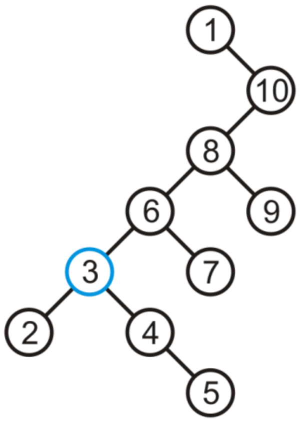
- This node must be splayed to the root beginning with a zig-zag operation



Examples

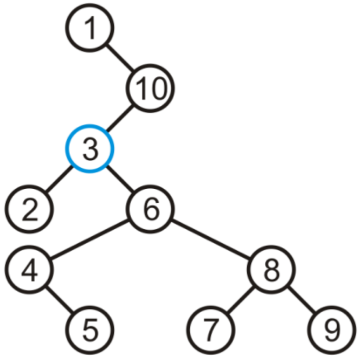
The node 3 is rotated up

- Next we require a zig-zig operation



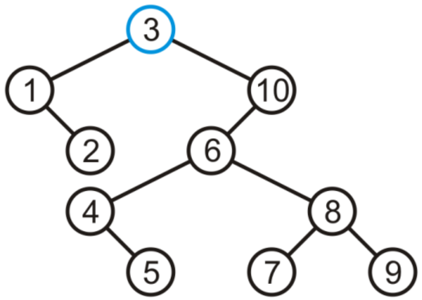
Examples

Finally, to bring 3 to the root, we need a zig-zag operation



Examples

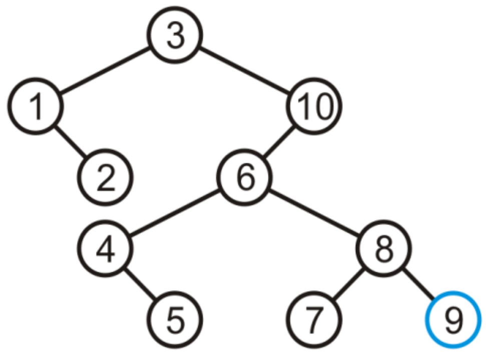
The height of this tree is only 4



Examples

Of the three deepest nodes, 9 requires a zig-zig operation, so will access it next

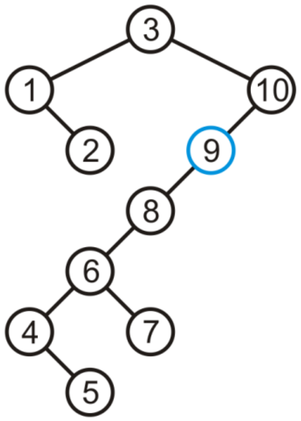
- The zig-zig operation will push 6 and its left sub-tree down



Examples

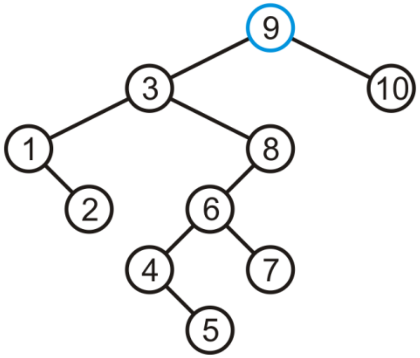
This is closer to a linked list; however, we're not finished

- A zig-zag operation will move 9 to the root



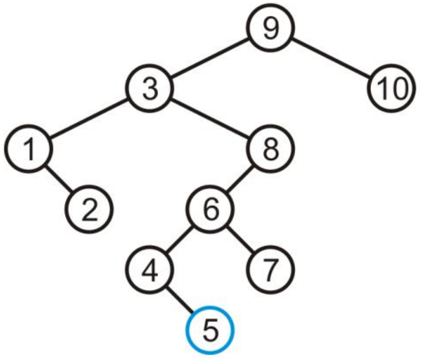
Examples

In this case, the height of the tree is now greater: 5



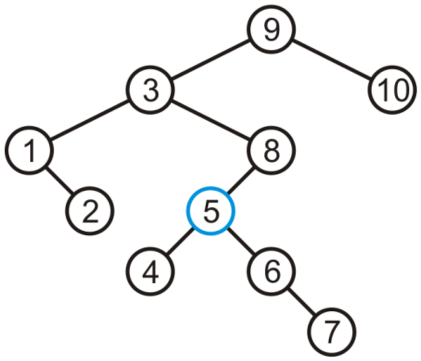
Examples

Accessing the deepest node, 5, we must begin with a zig-zag operation



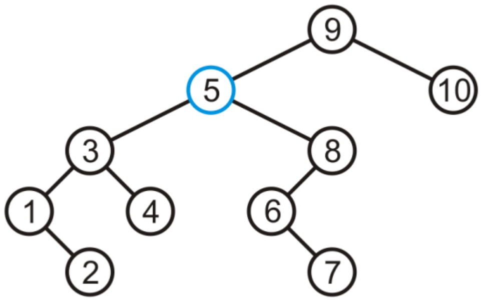
Examples

Next, we require a zig-zag operation to move 5 to the location of 3



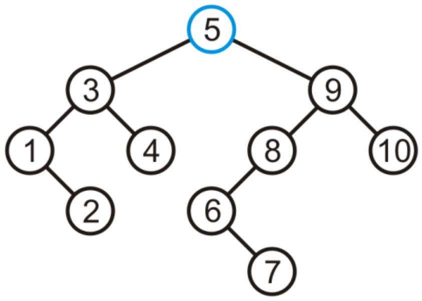
Examples

Finally, we require a single zig operation to move 5 to the root



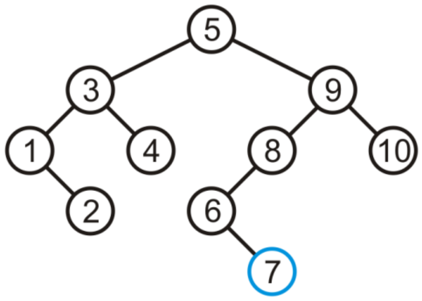
Examples

The height of the tree is 4; however, 7 of the nodes form a perfect tree at the root



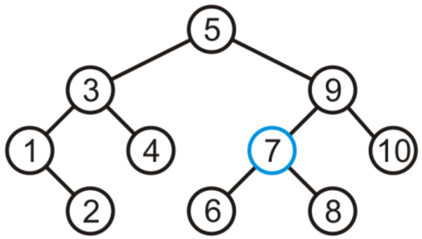
Examples

Accessing 7 will require two zig-zag operations



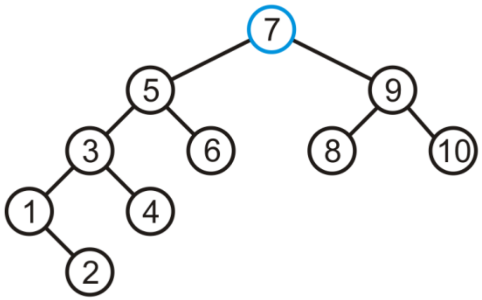
Examples

The first zig-zag moves it to depth 2



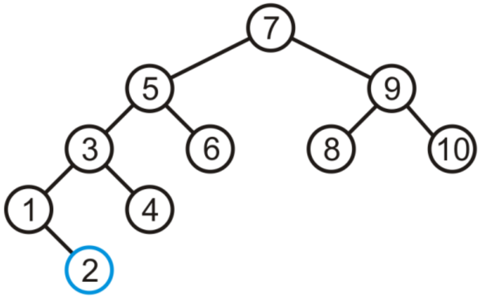
Examples

7 is promoted to the root through a zig-zag operation



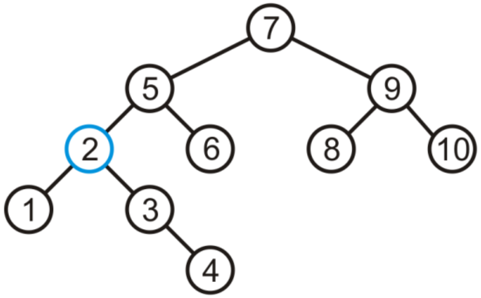
Examples

Finally, accessing 2, we first require a zig-zag operation



Examples

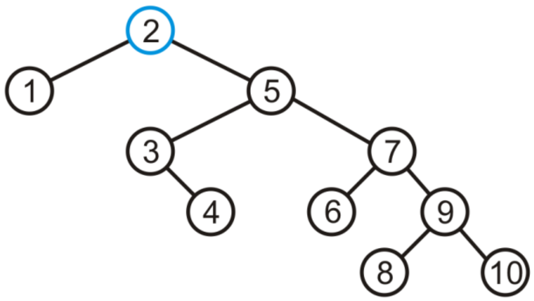
This now requires a zig-zig operation to promote 2 to the root



Examples

In this case, with 2 at the root, 3-10 must be in the right sub-tree

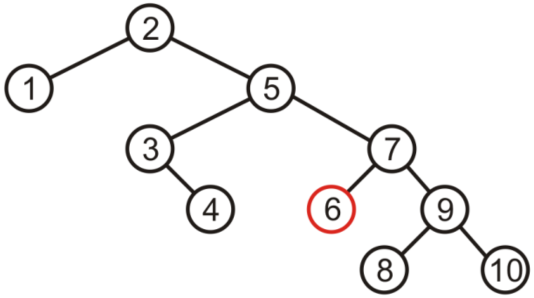
- The right sub-tree happens to be AVL balanced



Examples

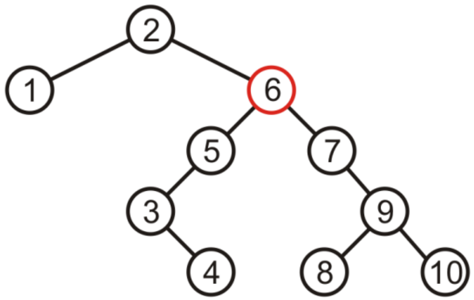
To remove a node, for example, 6, splay it to the root

- First we require a zig-zag operation



Examples

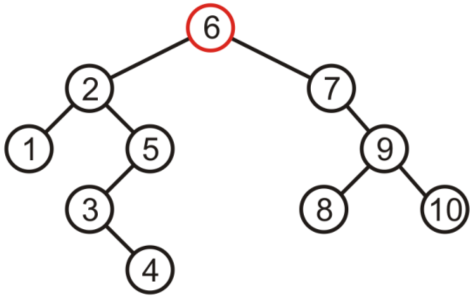
At this point, we need a zig operation to move 6 to the root



Examples

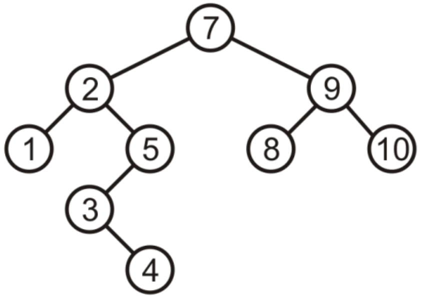
We will now copy the minimum element from the right sub-tree

- In this case, the node with 7 has a single sub-tree, we will simply move it up



Examples

Thus, we have removed 6 and the resulting tree is, again, reasonably balanced



Amortized Analysis

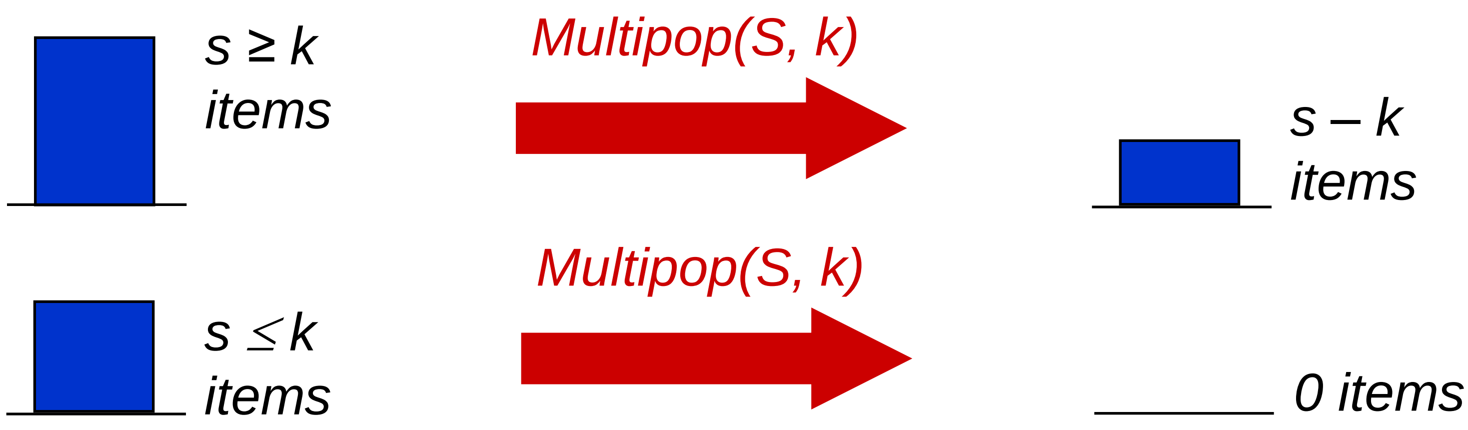
- Based on the **Accounting Method**
  - **Idea:** When an operation's amortized cost exceeds it actual cost, the difference is assigned to certain tree nodes as **credit**.
  - Credit is used to pay for subsequent operations whose amortized cost is less than their actual cost.
- Most of our analysis 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?

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.
  - **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

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 $\geq$ 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

Proposition : *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,*

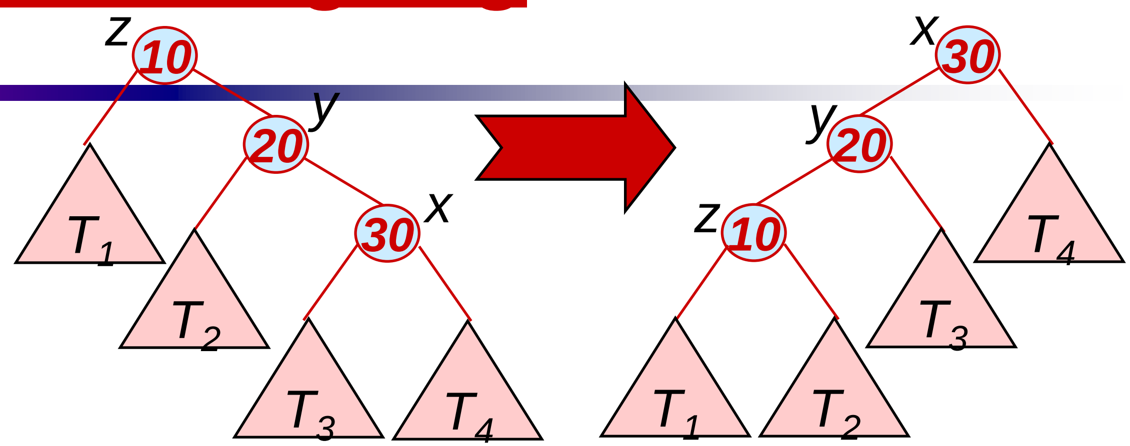
- $\delta \leq 3(r'(x) - r(x)) - 2$ *if the substep is a zig-zig or a zig-zag;*

- $\delta \leq 3(r'(x) - r(x))$ *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,

$$\delta = r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z)$$
$$= r'(y) + r'(z) - r(x) - r(y)$$
$$\le r'(x) + r'(z) - 2r(x). \quad \textbf{(1)}$$

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

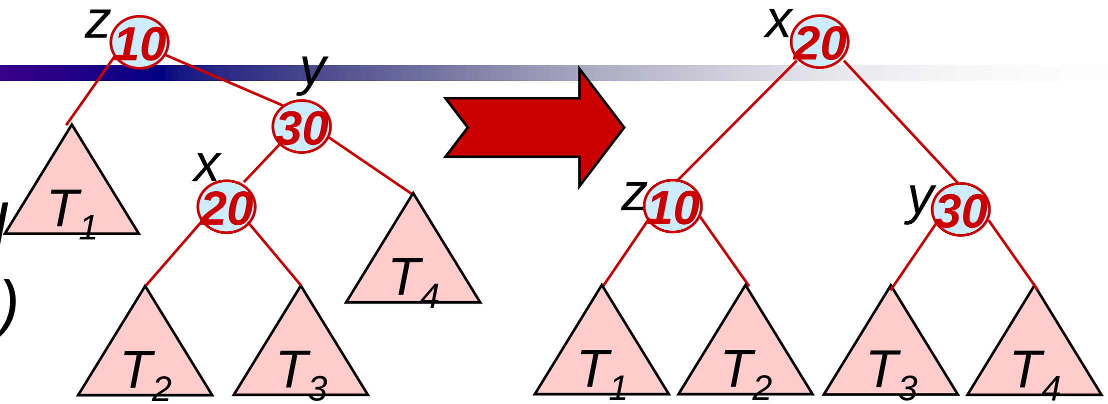
If $a > 0$, $b > 0$, and $c \ge a + b$, then $\lg a + \lg b \le 2 \lg c - 2$.

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

By (1) and (2), $\delta \le r'(x) + (2r'(x) - r(x) - 2) - 2r(x)$

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

Case 2: zig-zag



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

$$\delta = r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z)$$
$$= r'(y) + r'(z) - r(x) - r(y)$$
$$\leq r'(y) + r'(z) - 2r(x). \quad \textbf{(1)}$$

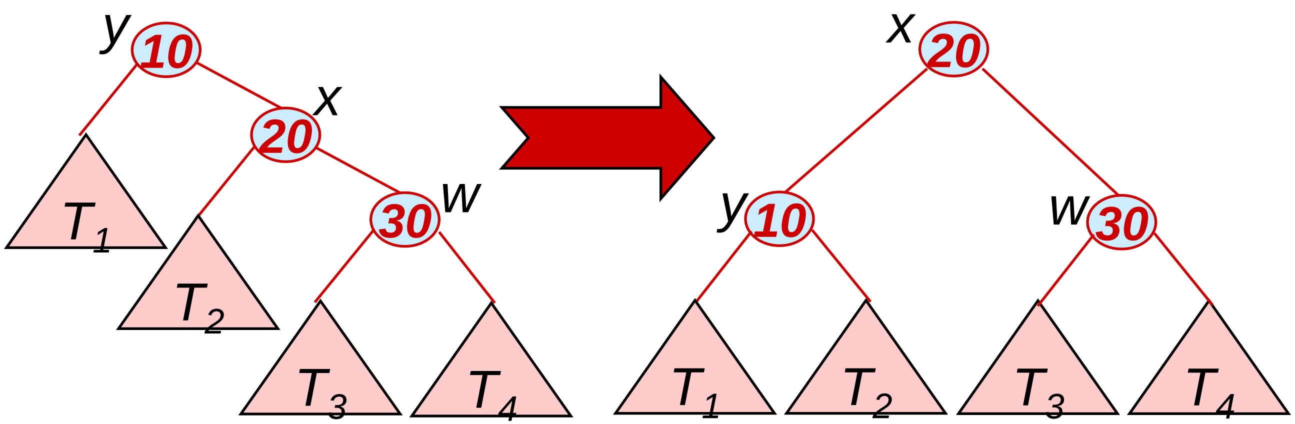
Also, $n'(y) + n'(z) \leq n'(x)$, which (by property of lg), implies

$$r'(y) + r'(z) \leq 2r'(x) - 2. \quad \textbf{(2)}$$

By (1) and (2), $\delta \leq 2r'(x) - 2 - 2r(x)$

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

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

Proposition : *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. The*

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

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 $1 \le i \le p$.

By Proposition

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

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

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

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 $r(t) \leq 3\lg(2n + 1)$

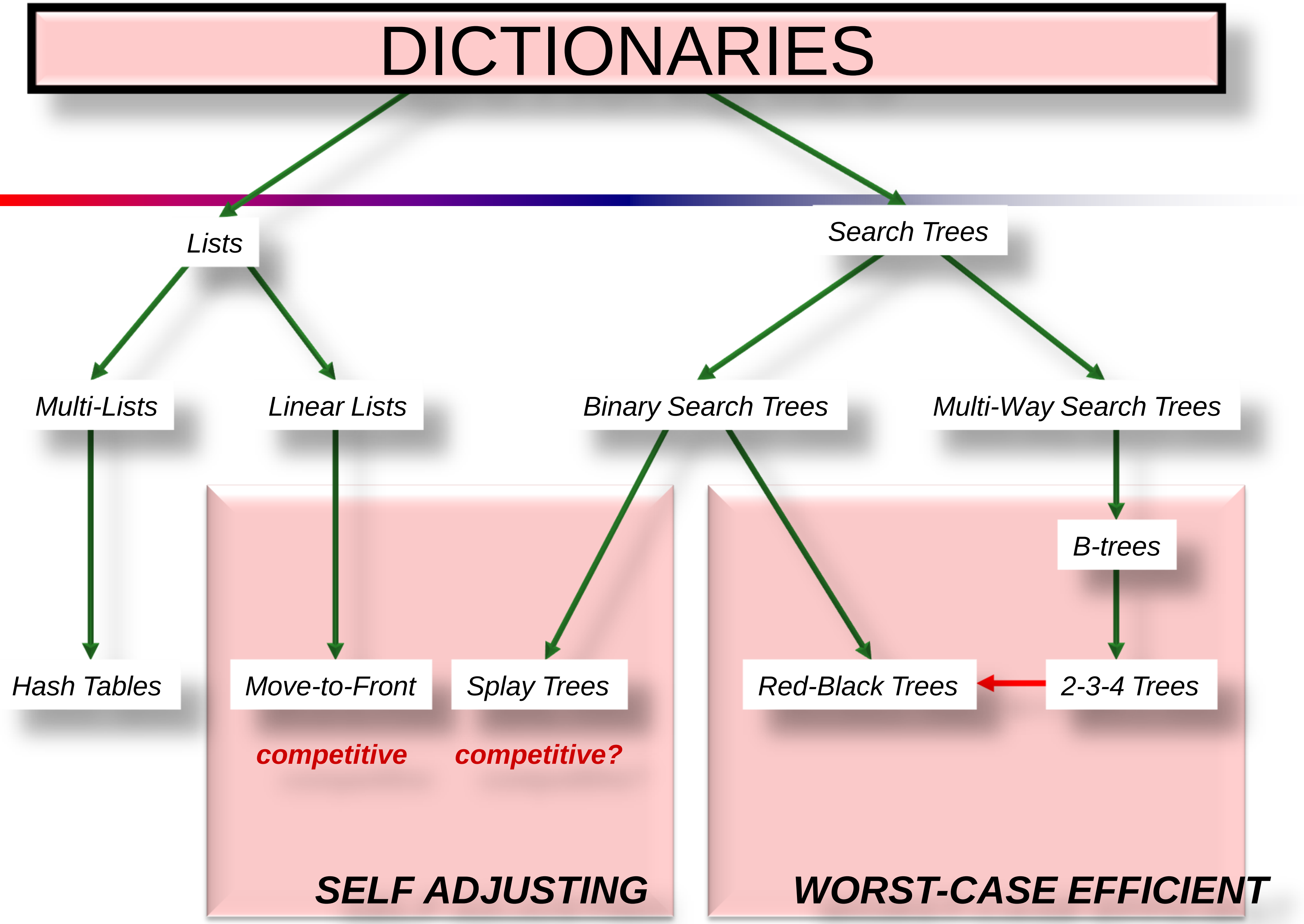
Comparisons

Advantages:

- The amortized run times are similar to that of AVL trees and red-black trees
- The implementation is easier
- No additional information (height/colour) is required

Disadvantages:

- The tree will change with read-only operations


DICTIONARIES
Lists
Search Trees
Multi-Lists
Linear Lists
Binary Search Trees
Multi-Way Search Trees
B-trees
Hash Tables
Move-to-Front
Splay Trees
Red-Black Trees
2-3-4 Trees
competitive
competitive?
SELF ADJUSTING
WORST-CASE EFFICIENT