Splay Trees

02-713

Dictionary Abstract Data Type (ADT)

- Most basic and most useful ADT:
  - `insert(key, value)`
  - `delete(key)`
  - `value = find(key)`

- Many languages have it built in:

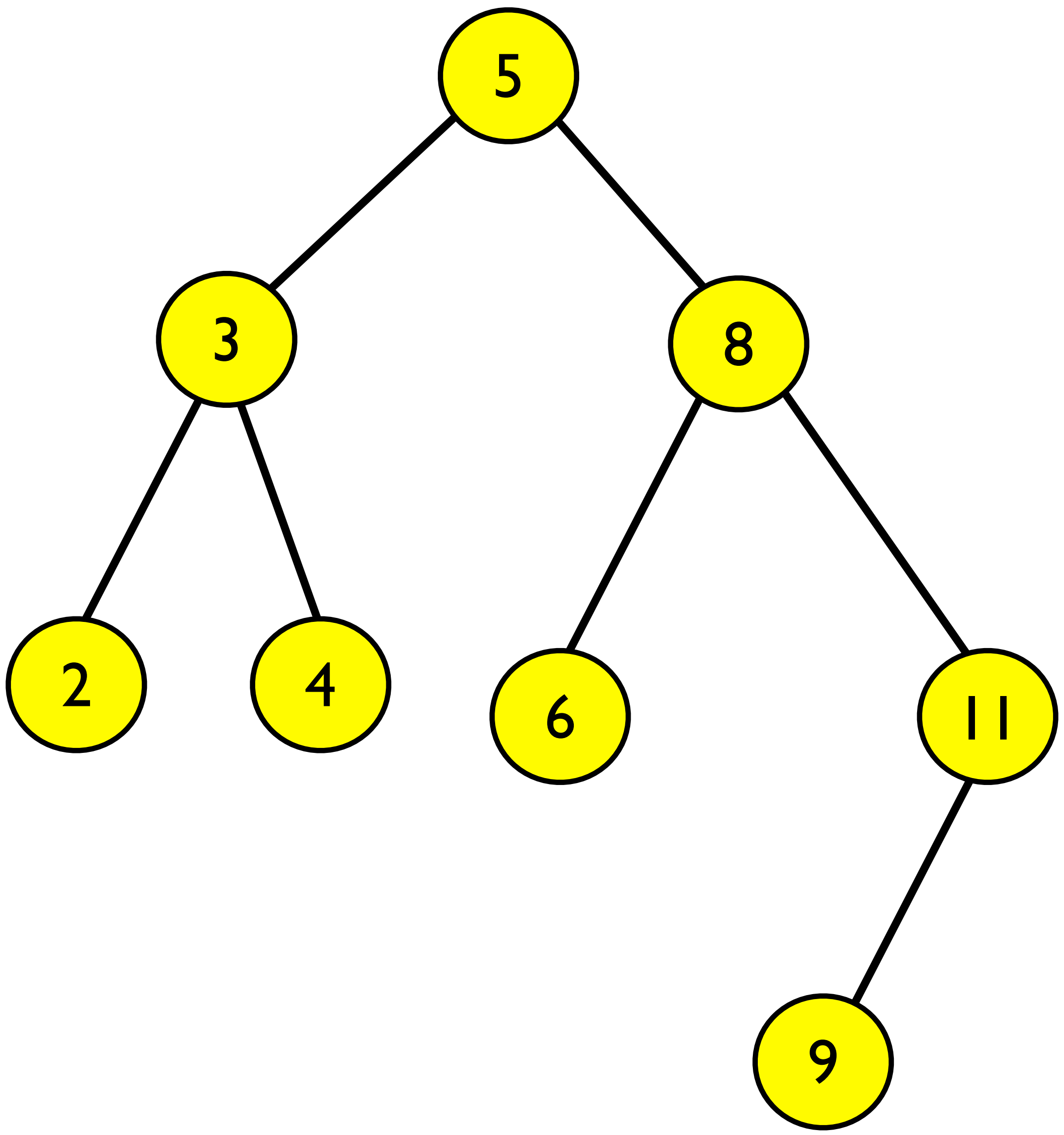
awk:	D["AAPL"] = 130	# associative array
perl:	`my %D; $D["AAPL"] = 130;`	# hash
python:	`D = {}; D["AAPL"] = 130`	# dictionary
C++:	`map<string,string> D = new map<string, string>();`	
	`D["AAPL"] = 130;`	// map

- **Insert**, **delete, find** each either O(log n) [C++] or expected constant [perl, python]

- How can such dictionaries are implemented?

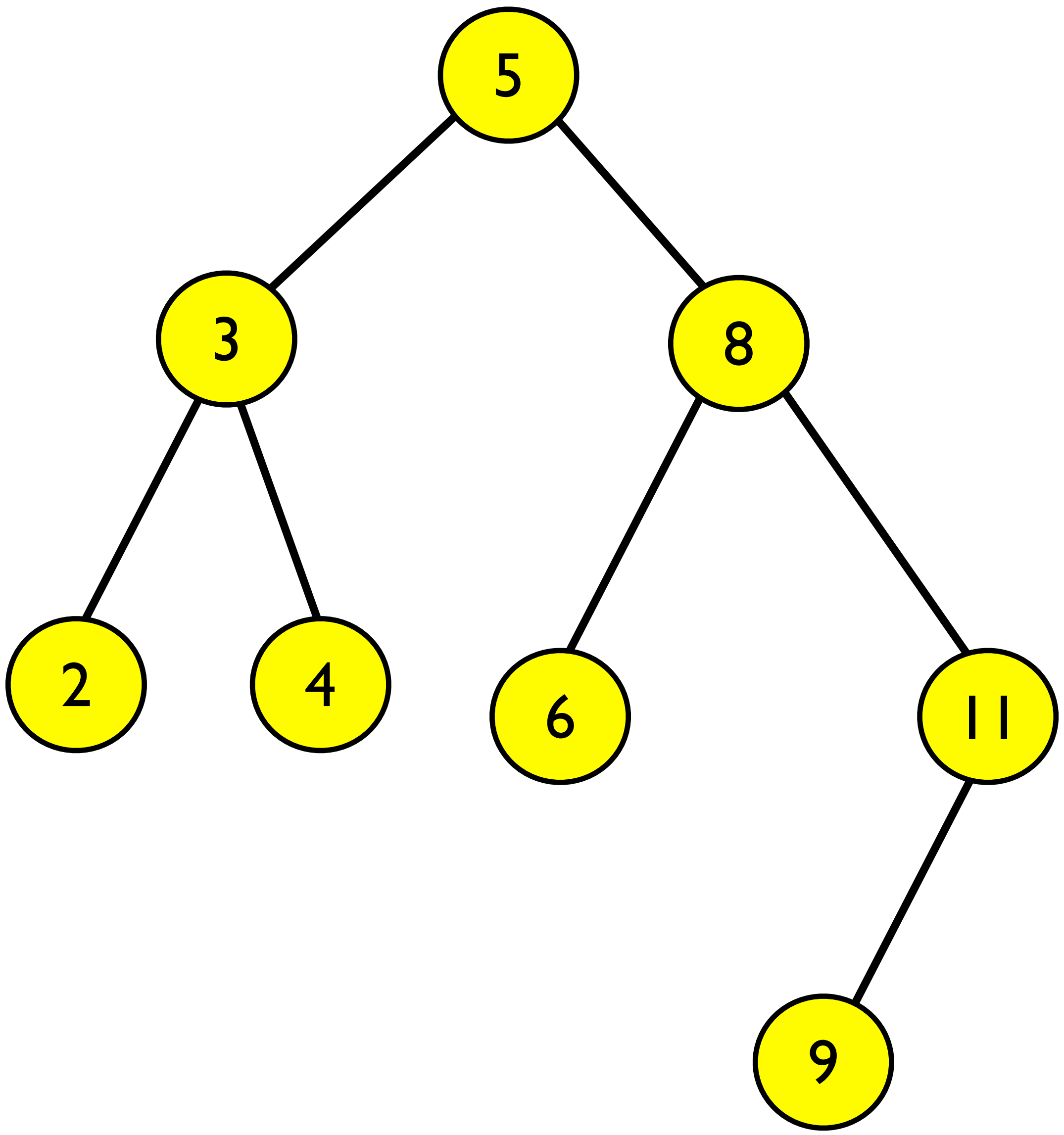
Binary Search Trees

- **BST Property:** If a node has key k then keys in the **left** subtree are $< k$ and keys in the **right** subtree are $> k$.
- For convenience, we disallow duplicate keys.
- Good for implementing the *dictionary ADT* we've already seen: insert, delete, find.



BST Find

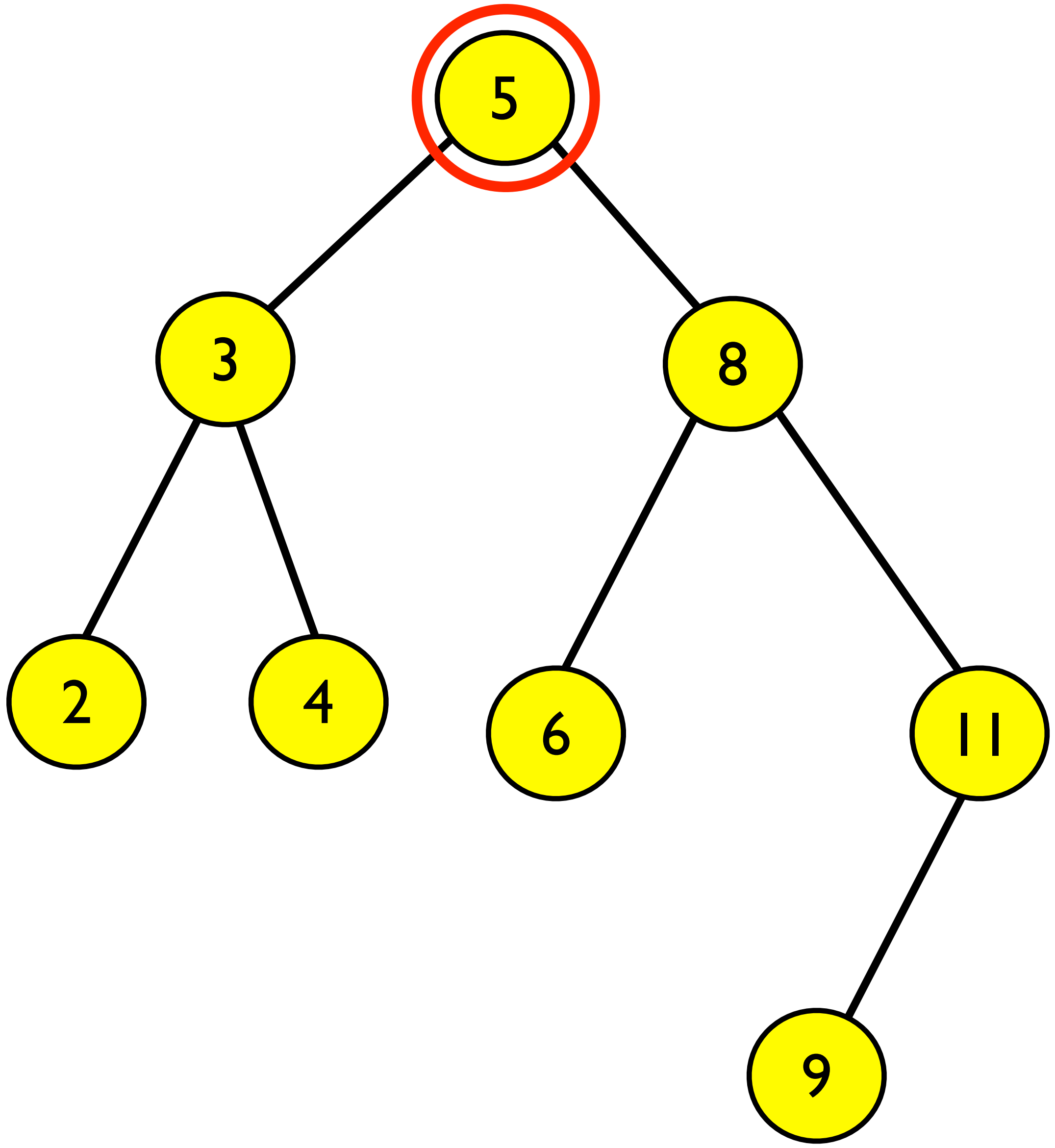
Find $k = 6$:



BST Find

Find $k = 6$:

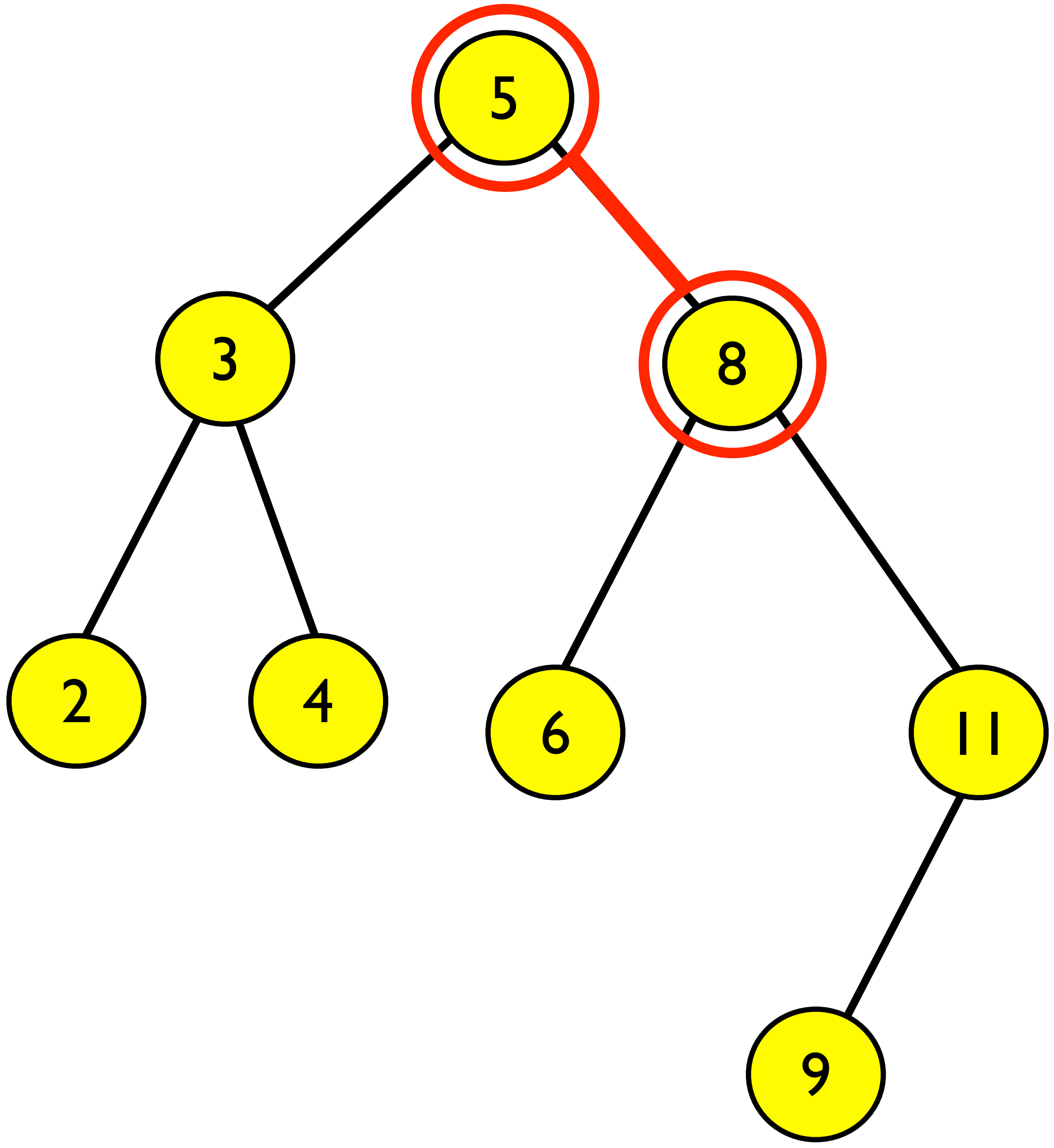
Is $k < 5$?



BST Find

Find $k = 6$:

Is $k < 5$? No, go right

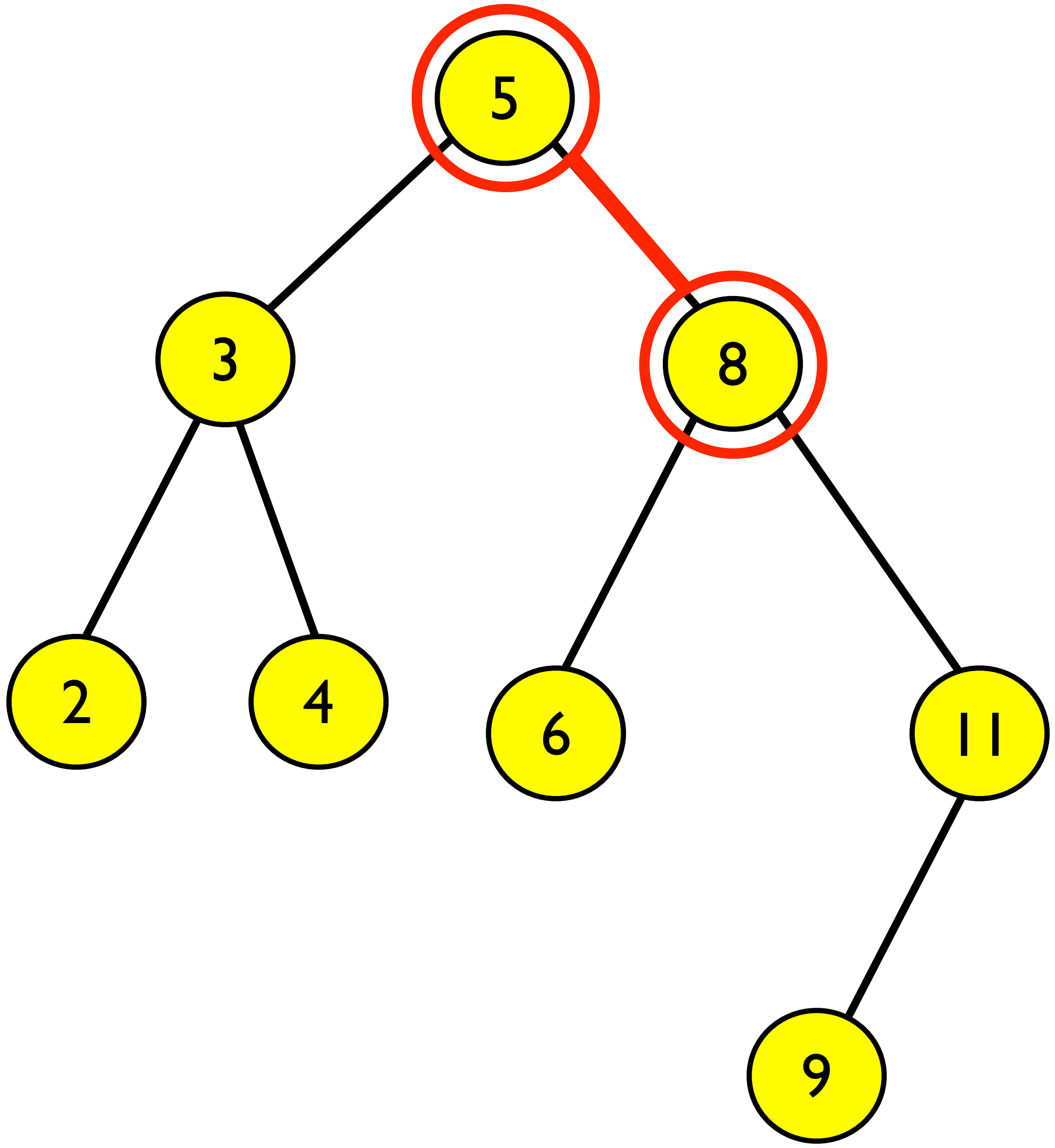


BST Find

Find $k = 6$:

Is $k < 5$? No, go right

Is $k < 8$?

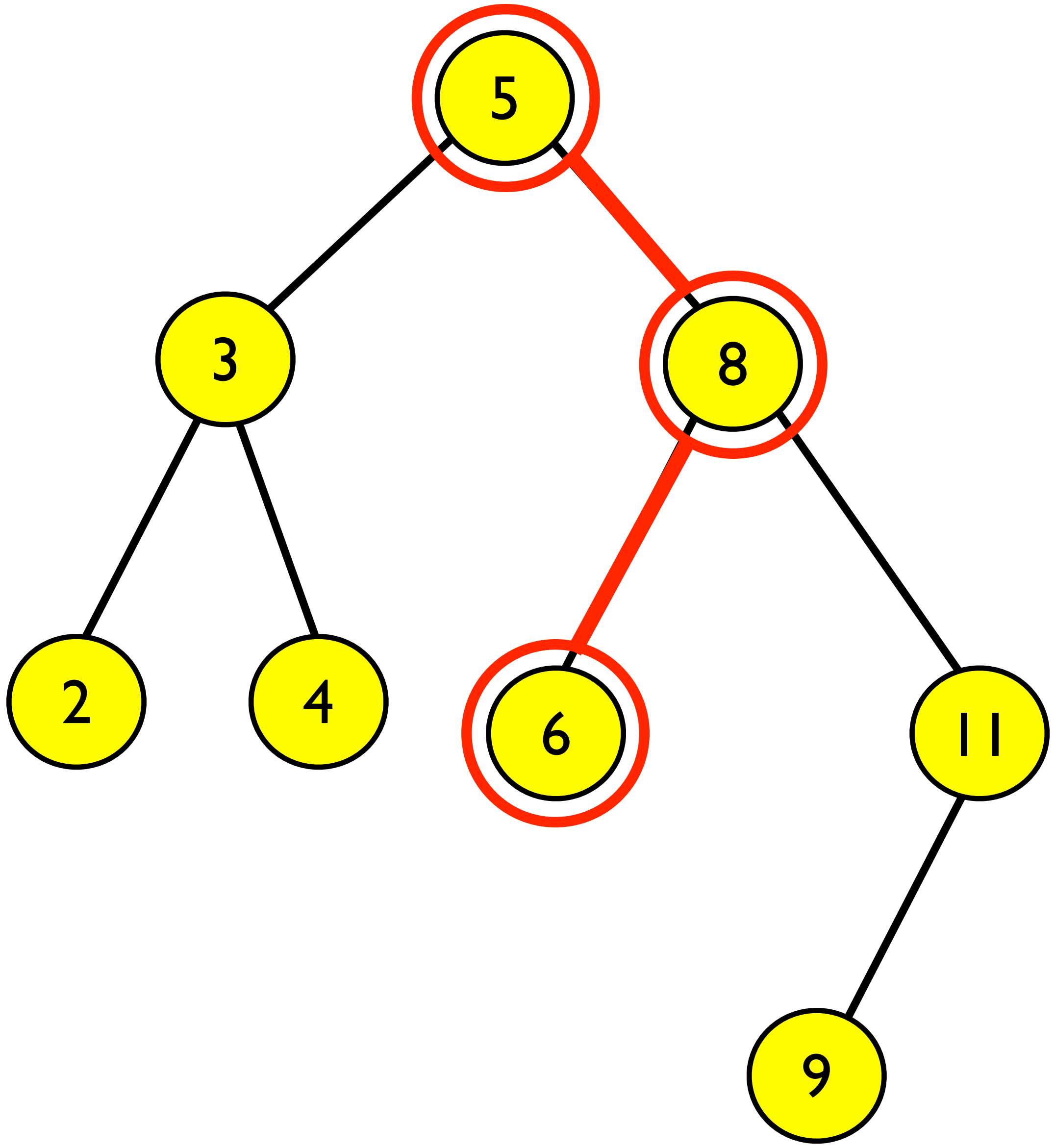


BST Find

Find $k = 6$:

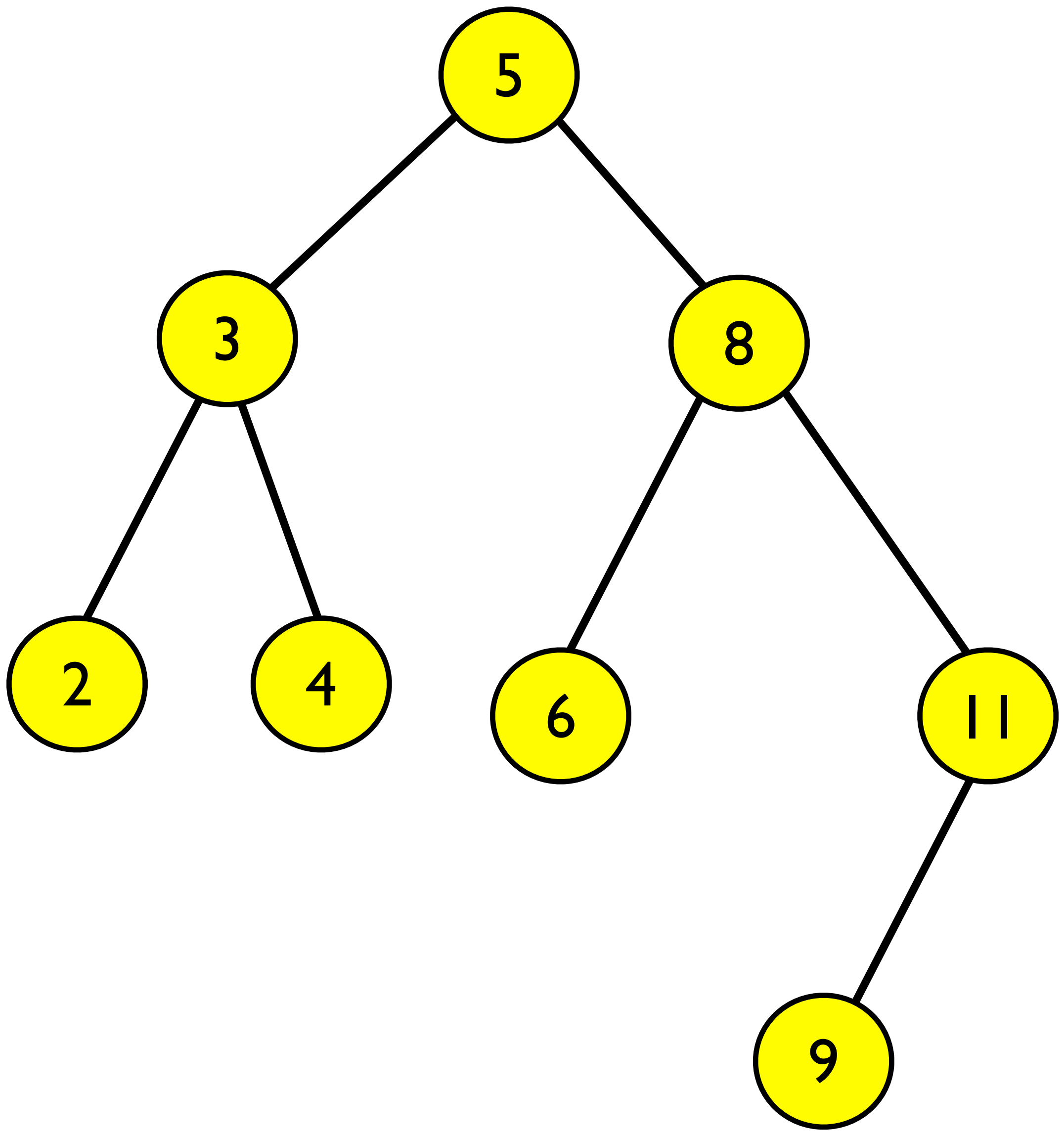
Is $k < 5$? No, go right

Is $k < 8$? Yes, go left



BST Find

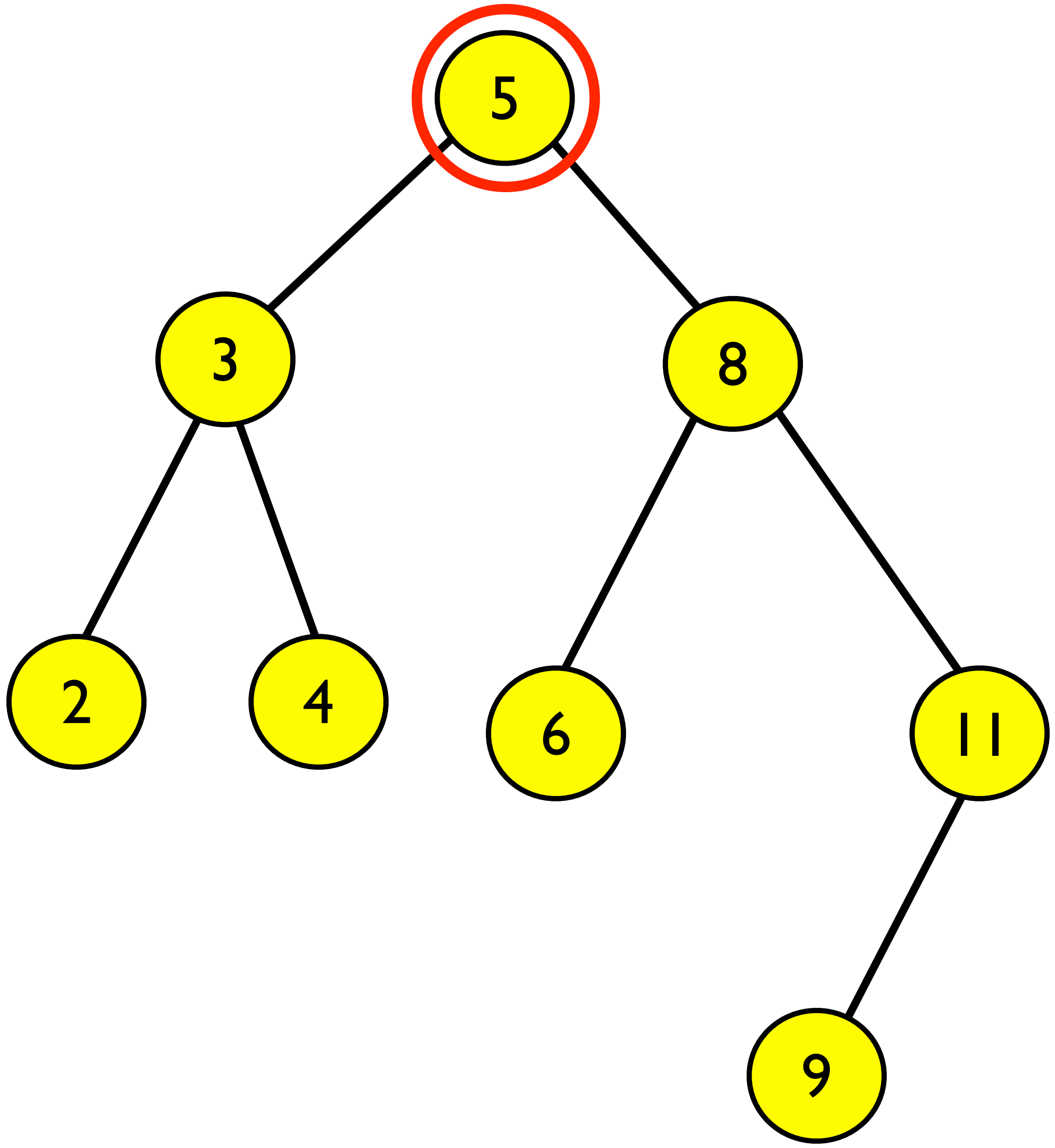
Find $k = 9$:



BST Find

Find $k = 9$:

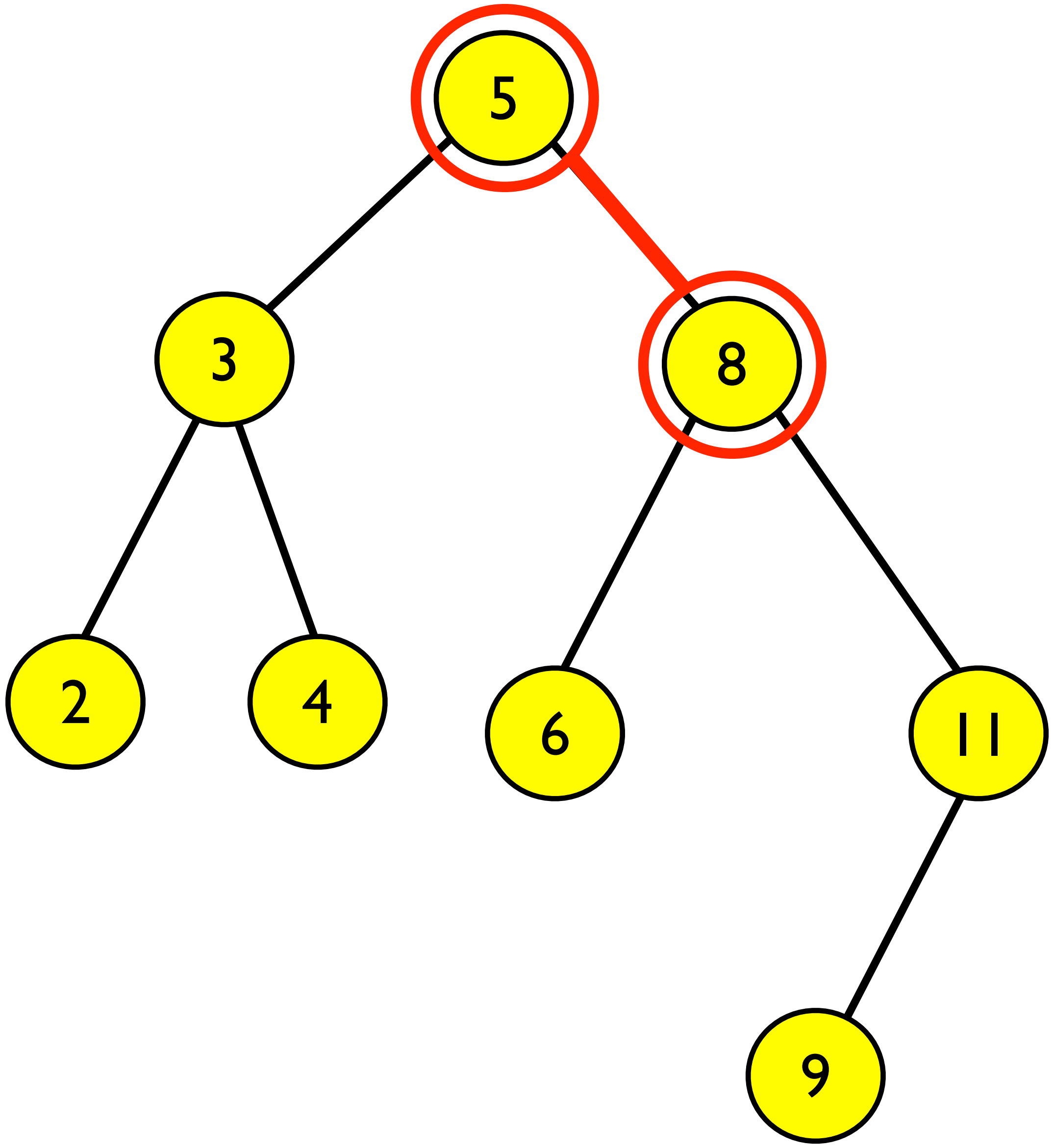
Is $k < 5$?



BST Find

Find $k = 9$:

Is $k < 5$? No, go right

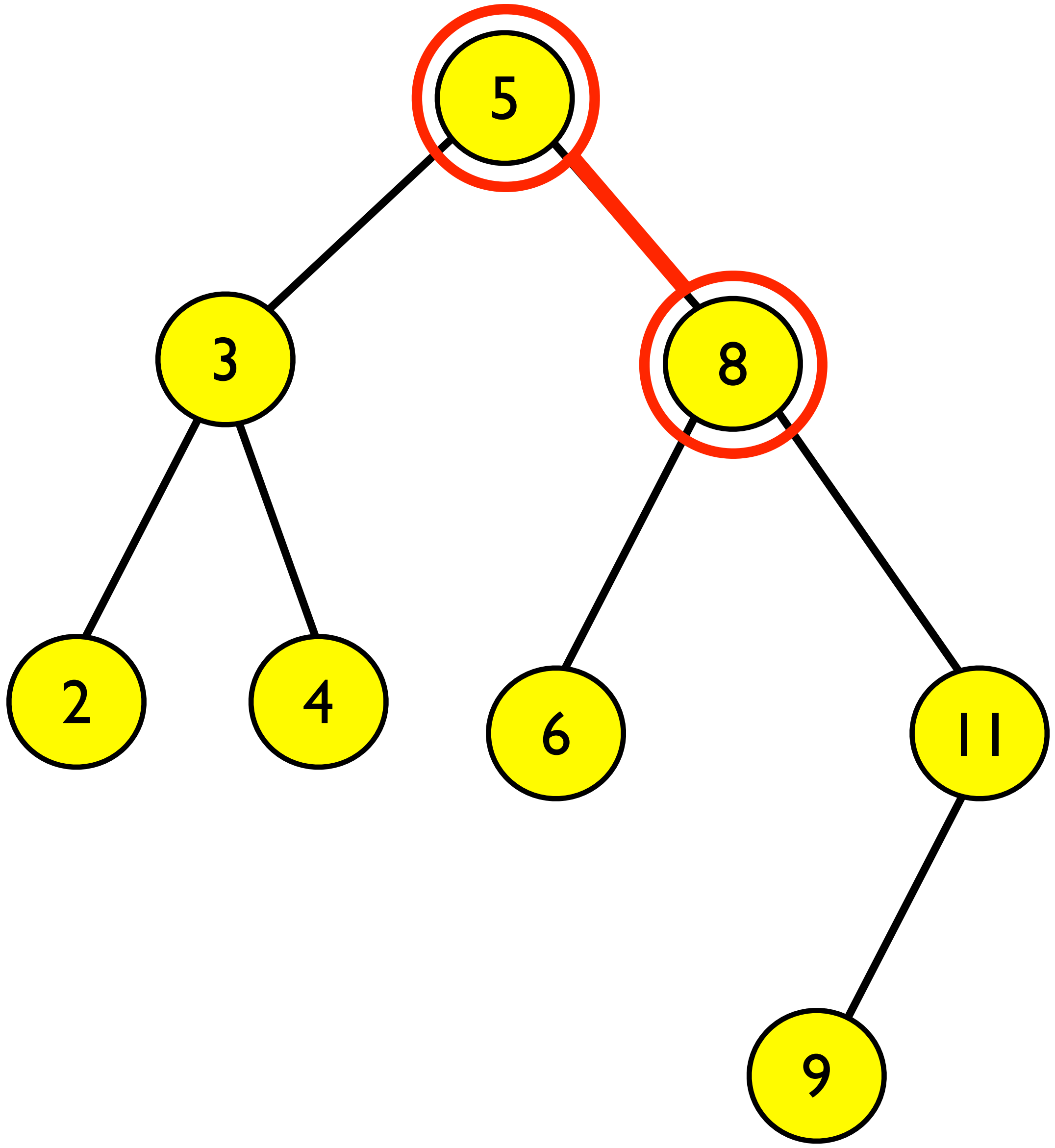


BST Find

Find $k = 9$:

Is $k < 5$? No, go right

Is $k < 8$?

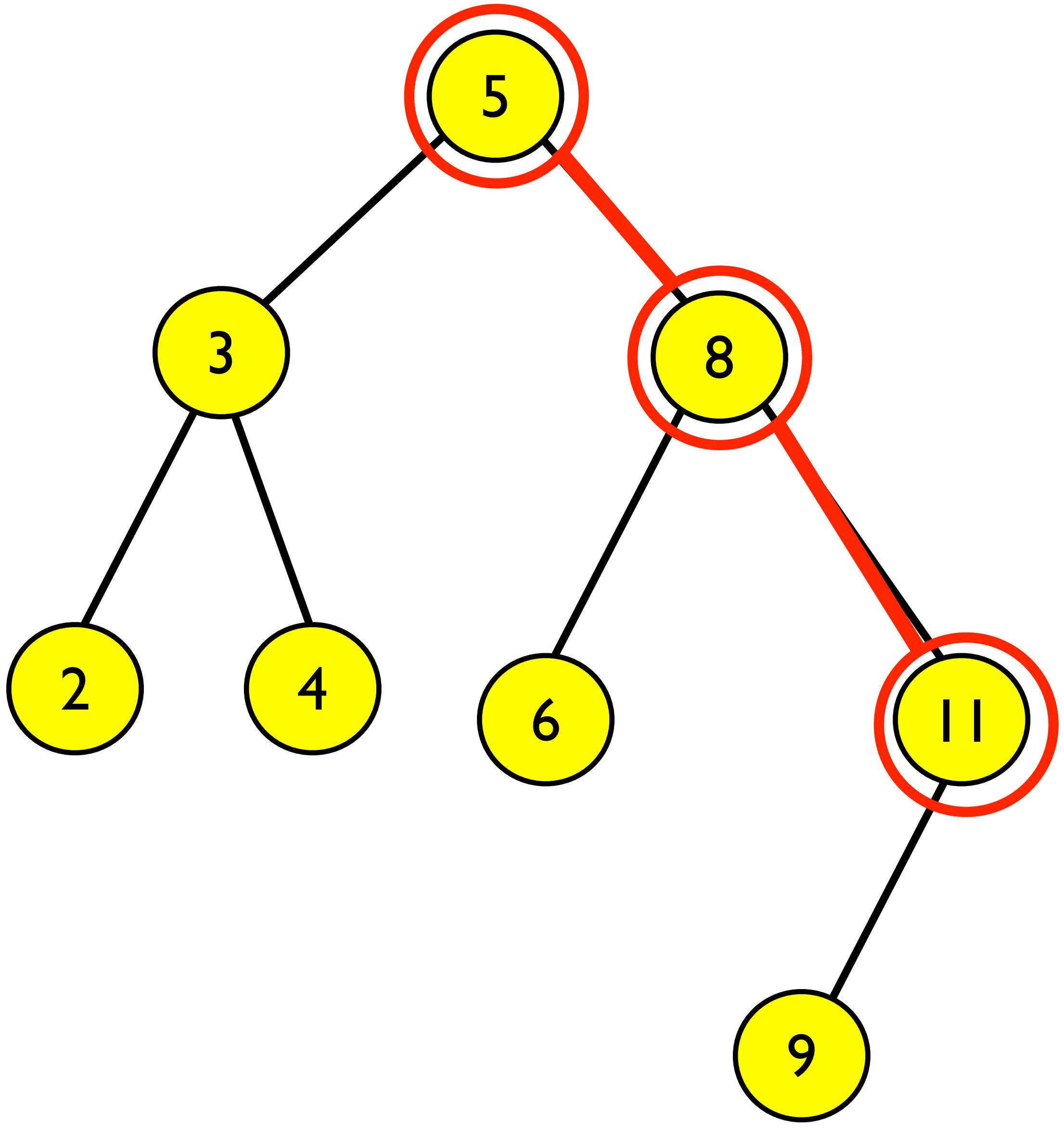


BST Find

Find $k = 9$:

Is $k < 5$? No, go right

Is $k < 8$? No, go right



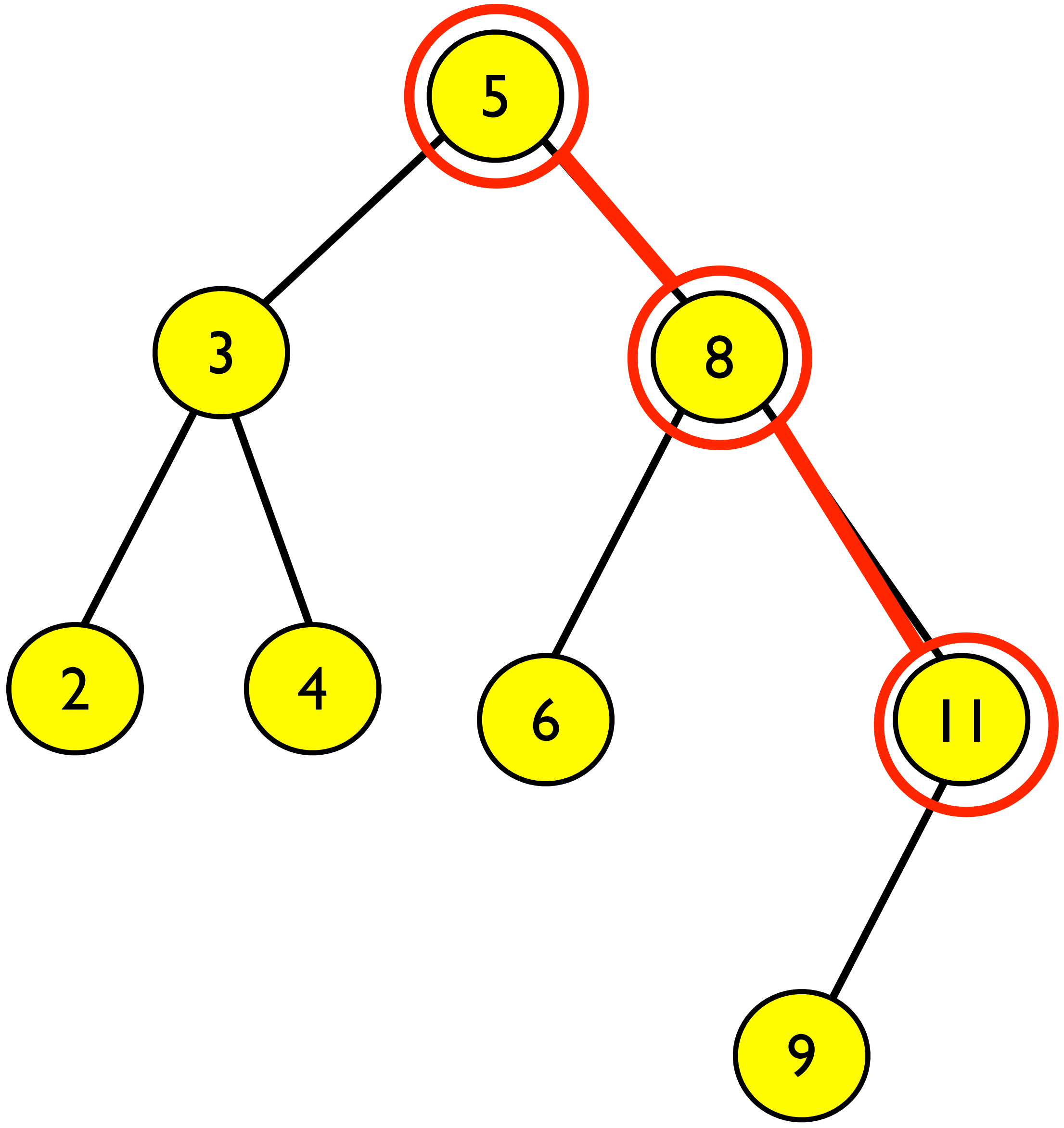
BST Find

Find $k = 9$:

Is $k < 5$? No, go right

Is $k < 8$? No, go right

Is $k < 11$?



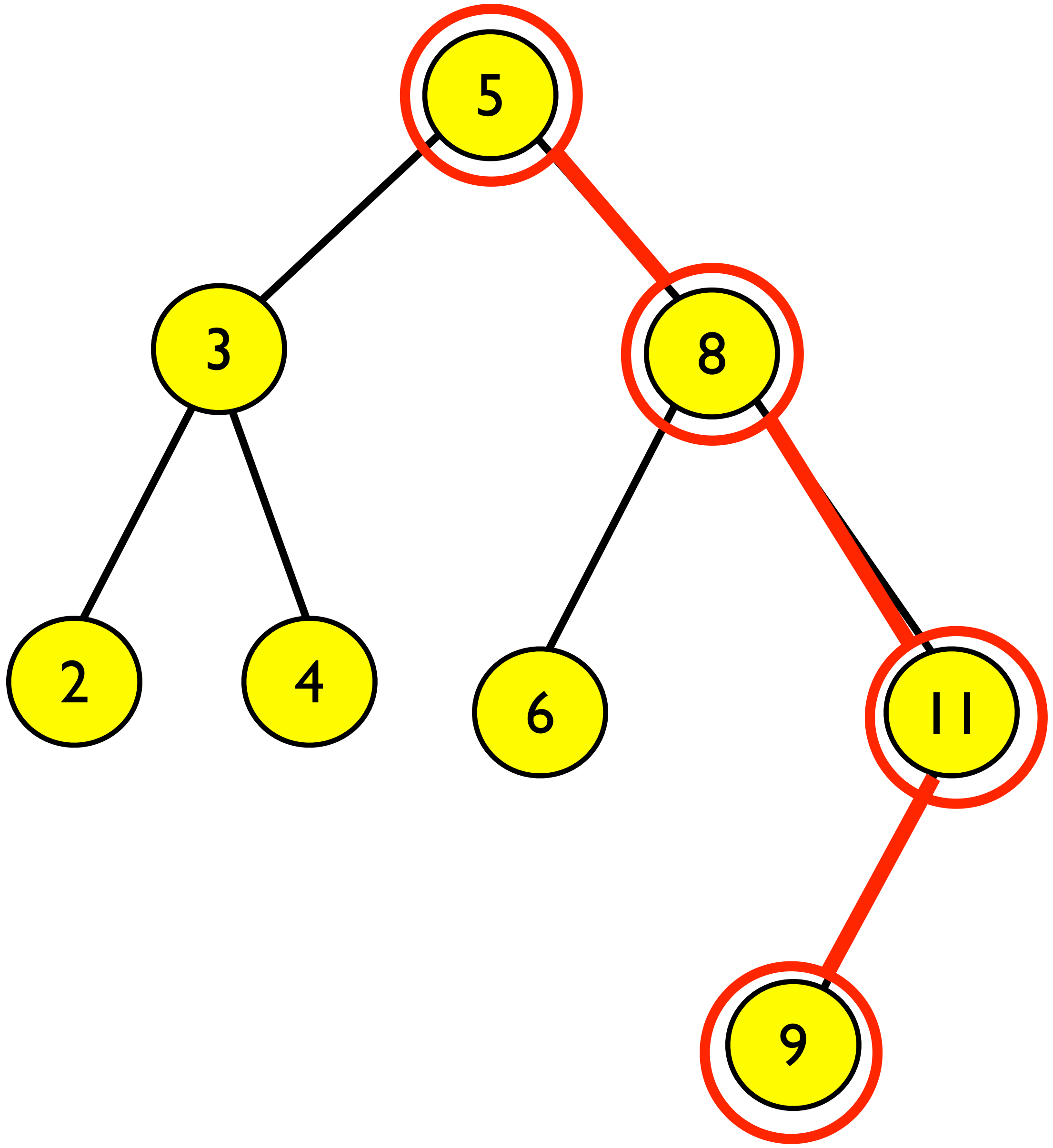
BST Find

Find $k = 9$:

Is $k < 5$? No, go right

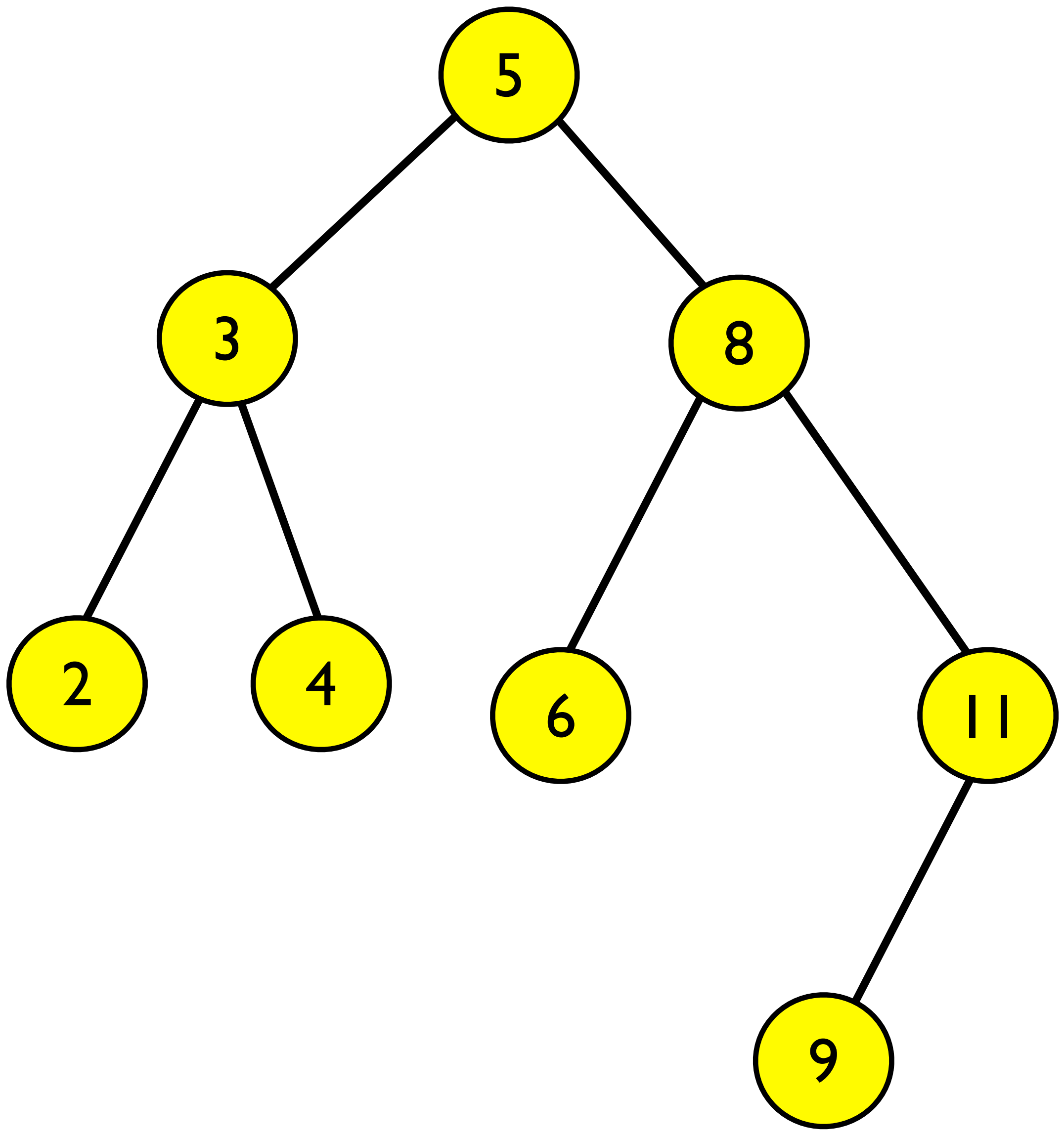
Is $k < 8$? No, go right

Is $k < 11$? Yes, go left



BST Find

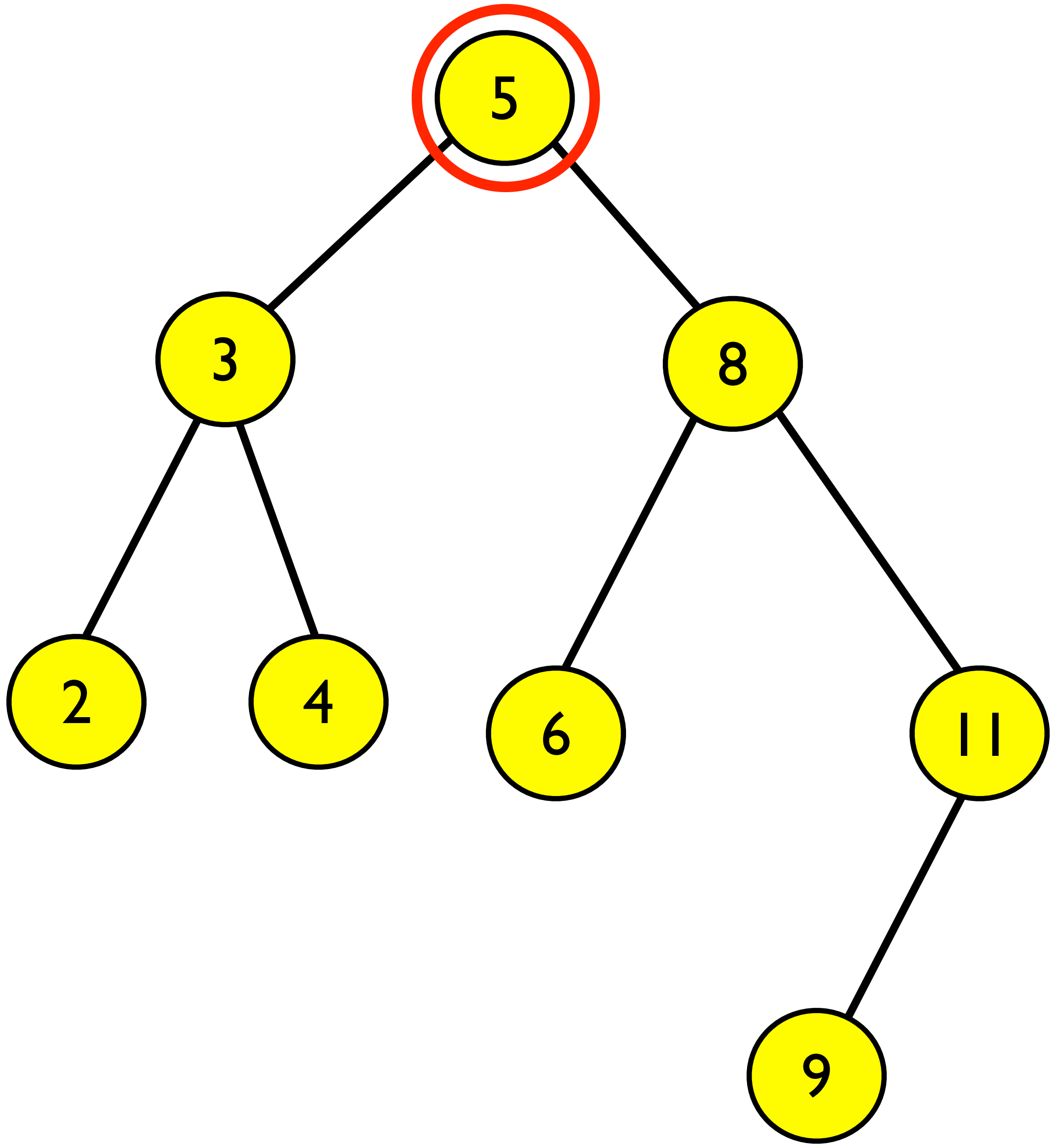
Find $k = 13$:



BST Find

Find $k = 13$:

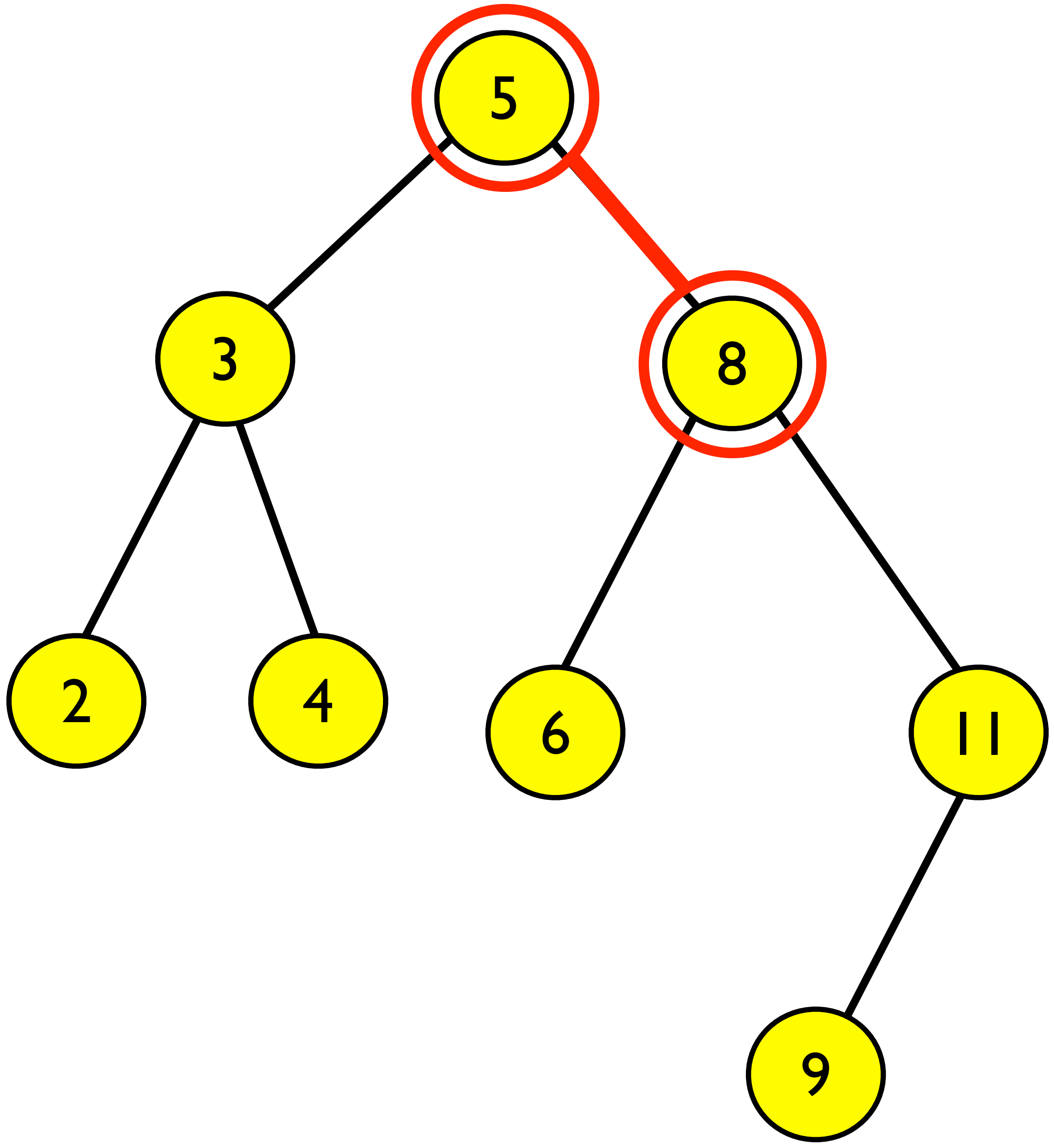
Is $k < 5$?



BST Find

Find $k = 13$:

Is $k < 5$? No, go right

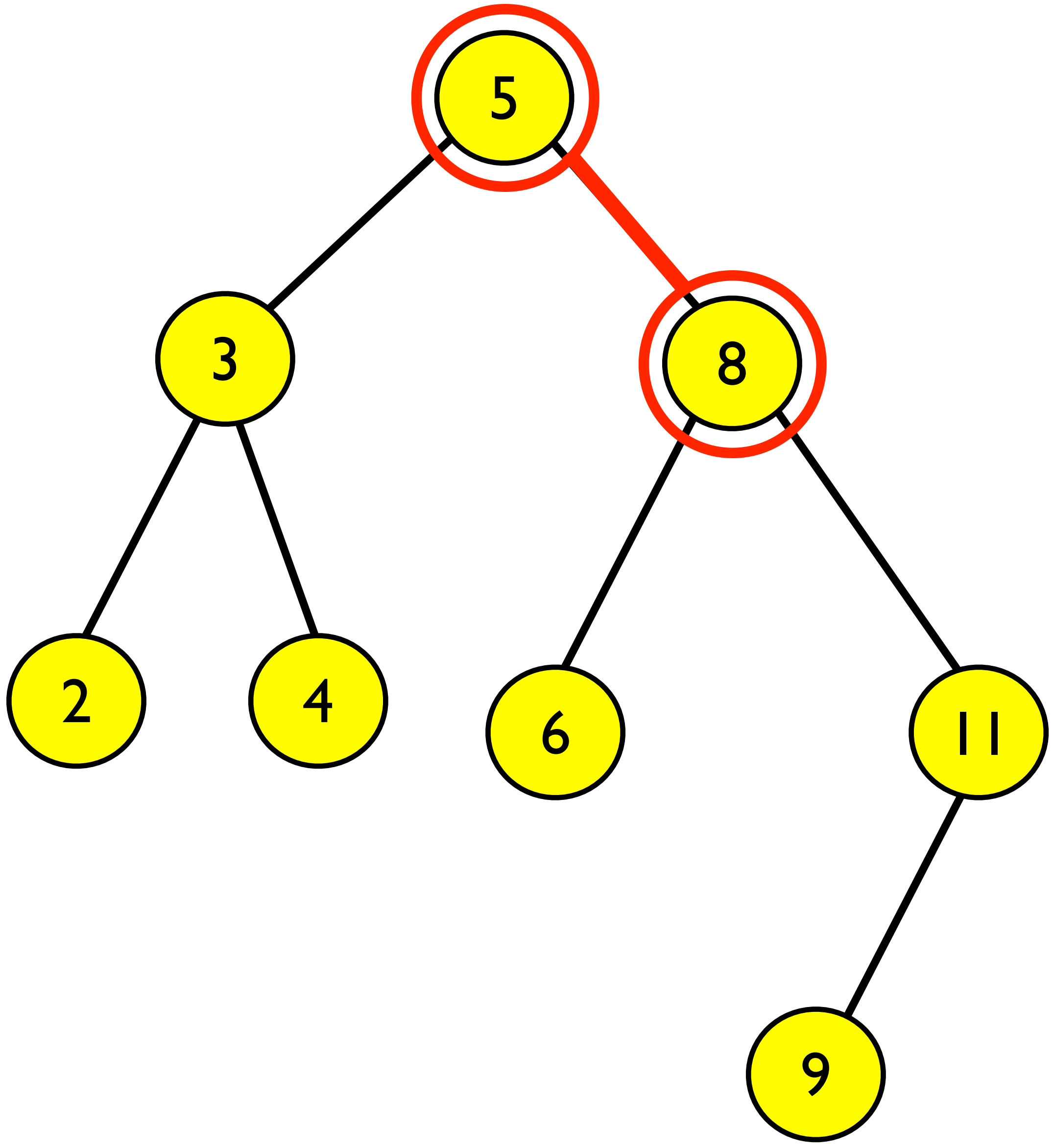


BST Find

Find $k = 13$:

Is $k < 5$? No, go right

Is $k < 8$?

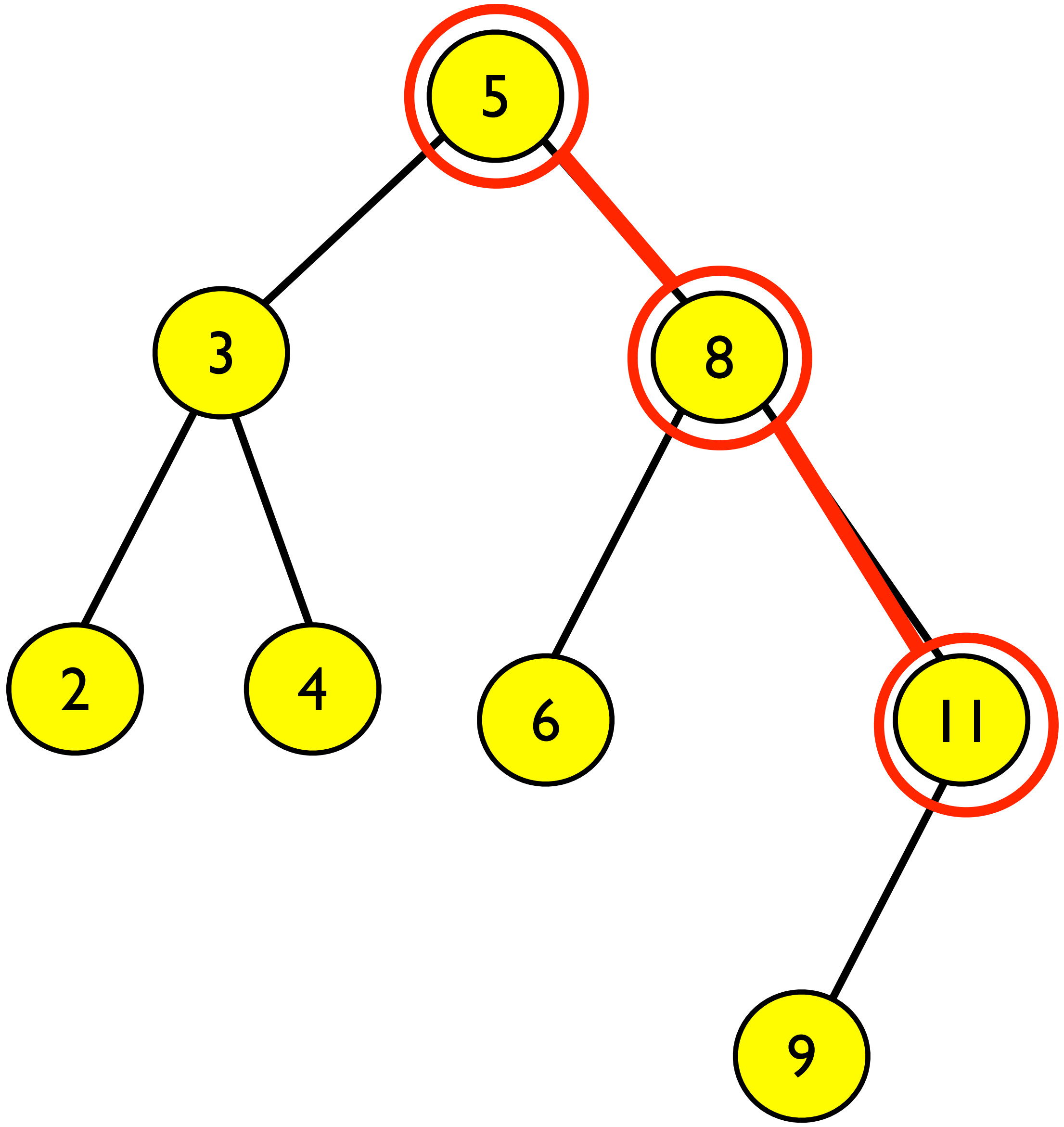


BST Find

Find $k = 13$:

Is $k < 5$? No, go right

Is $k < 8$? No, go right



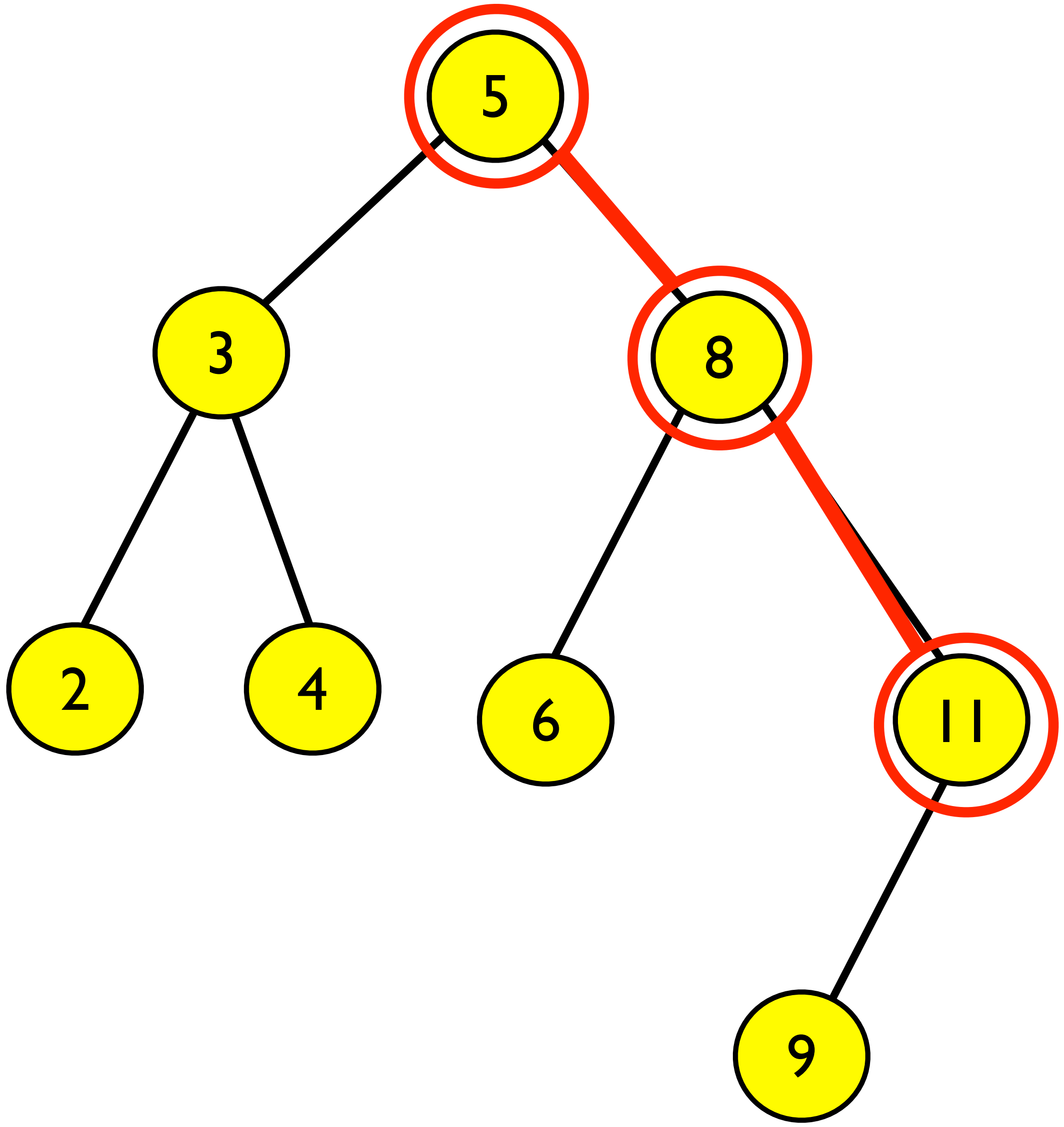
BST Find

Find $k = 13$:

Is $k < 5$? No, go right

Is $k < 8$? No, go right

Is $k < 11$?



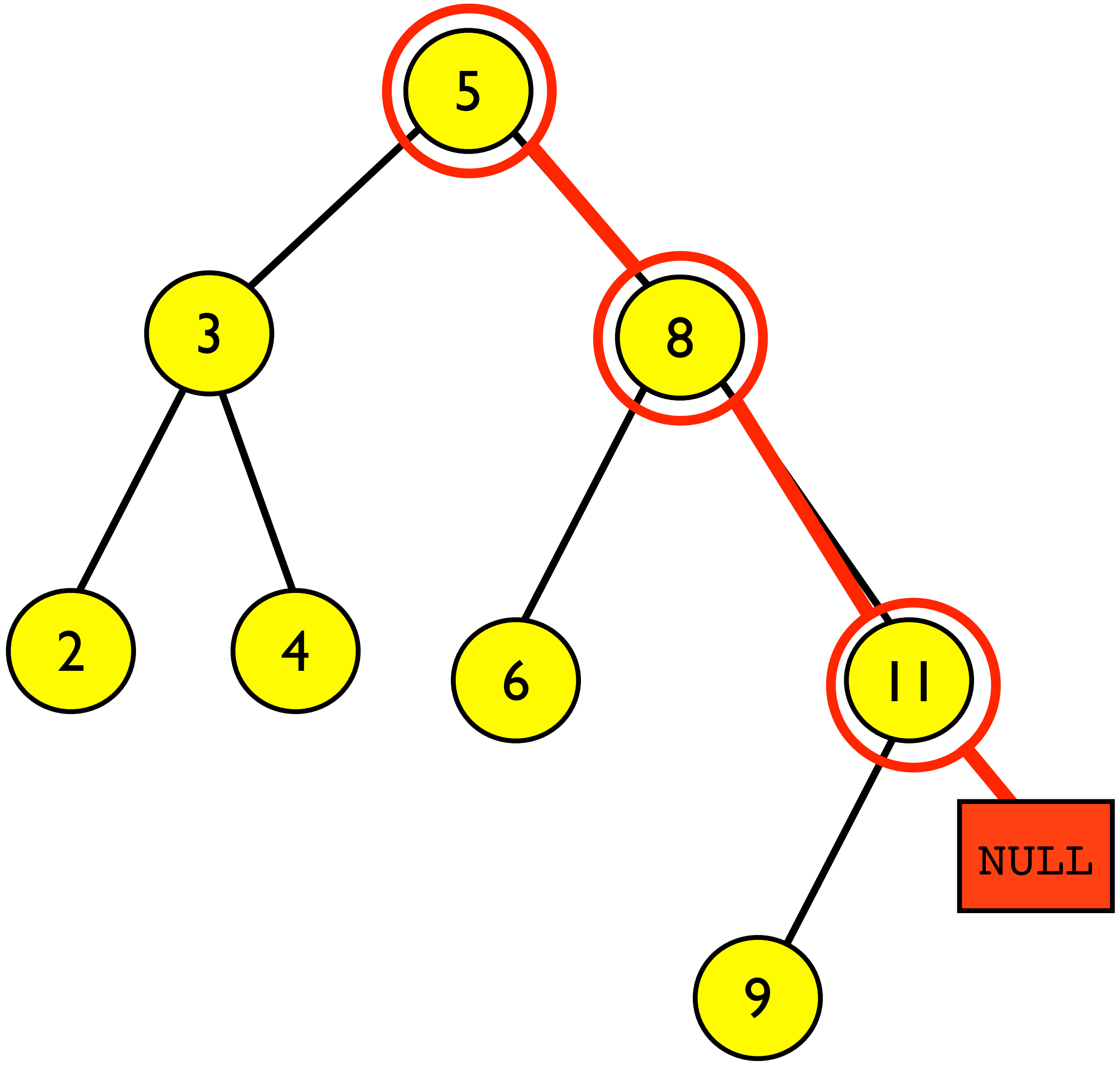
BST Find

Find $k = 13$:

Is $k < 5$? No, go right

Is $k < 8$? No, go right

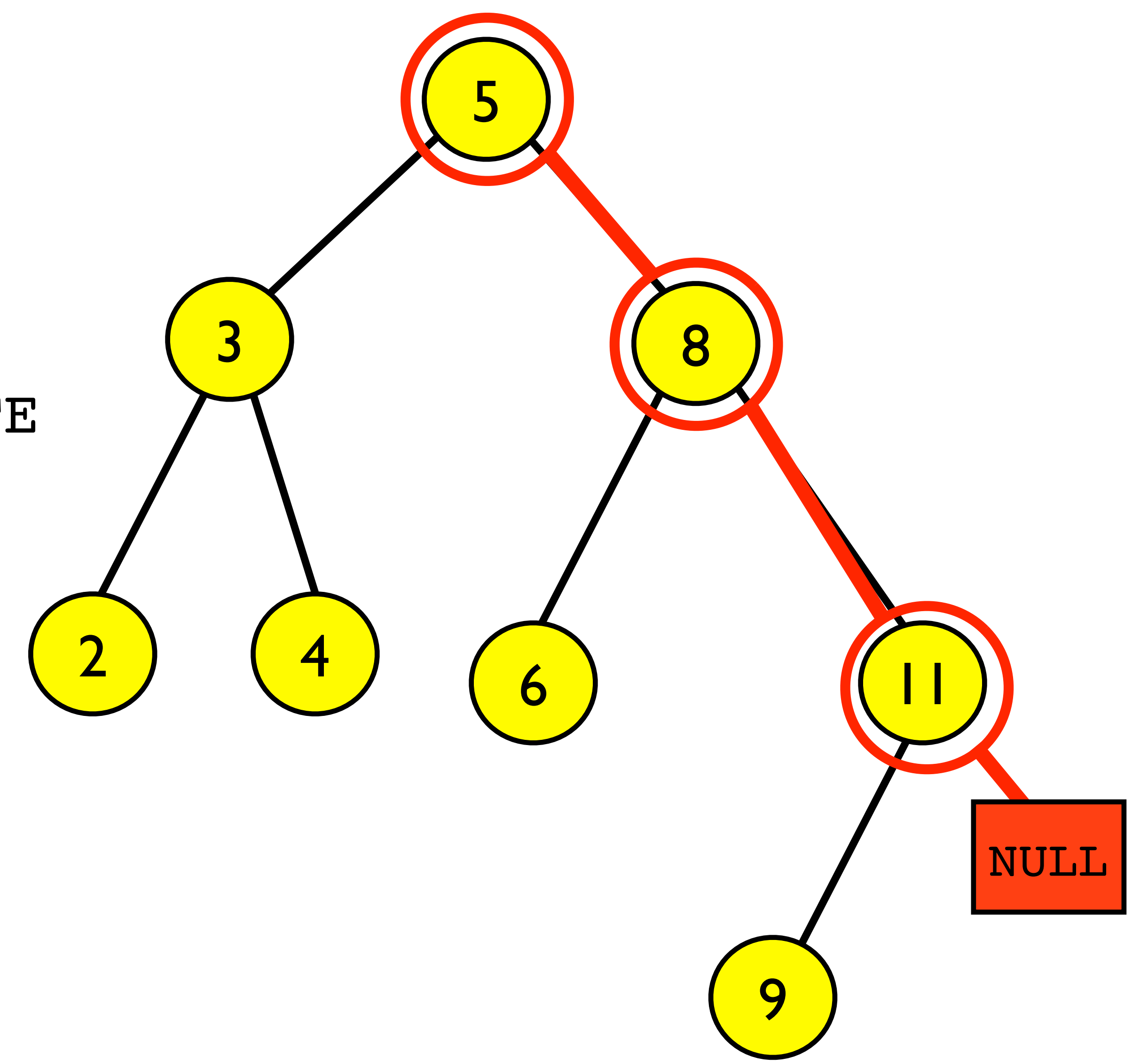
Is $k < 11$? No, go right



BST Insert

```
insert(T, K):
   q = NULL
   p = T
   while p != NULL and p.key != K:
      q = p
      if p.key < K:
        p = p.right
      else if p.key > K:
        p = p.left

   if p != NULL: error DUPLICATE

   N = new Node(K)
   if q.key > K:
     q.left = N
   else:
     q.right = N
```

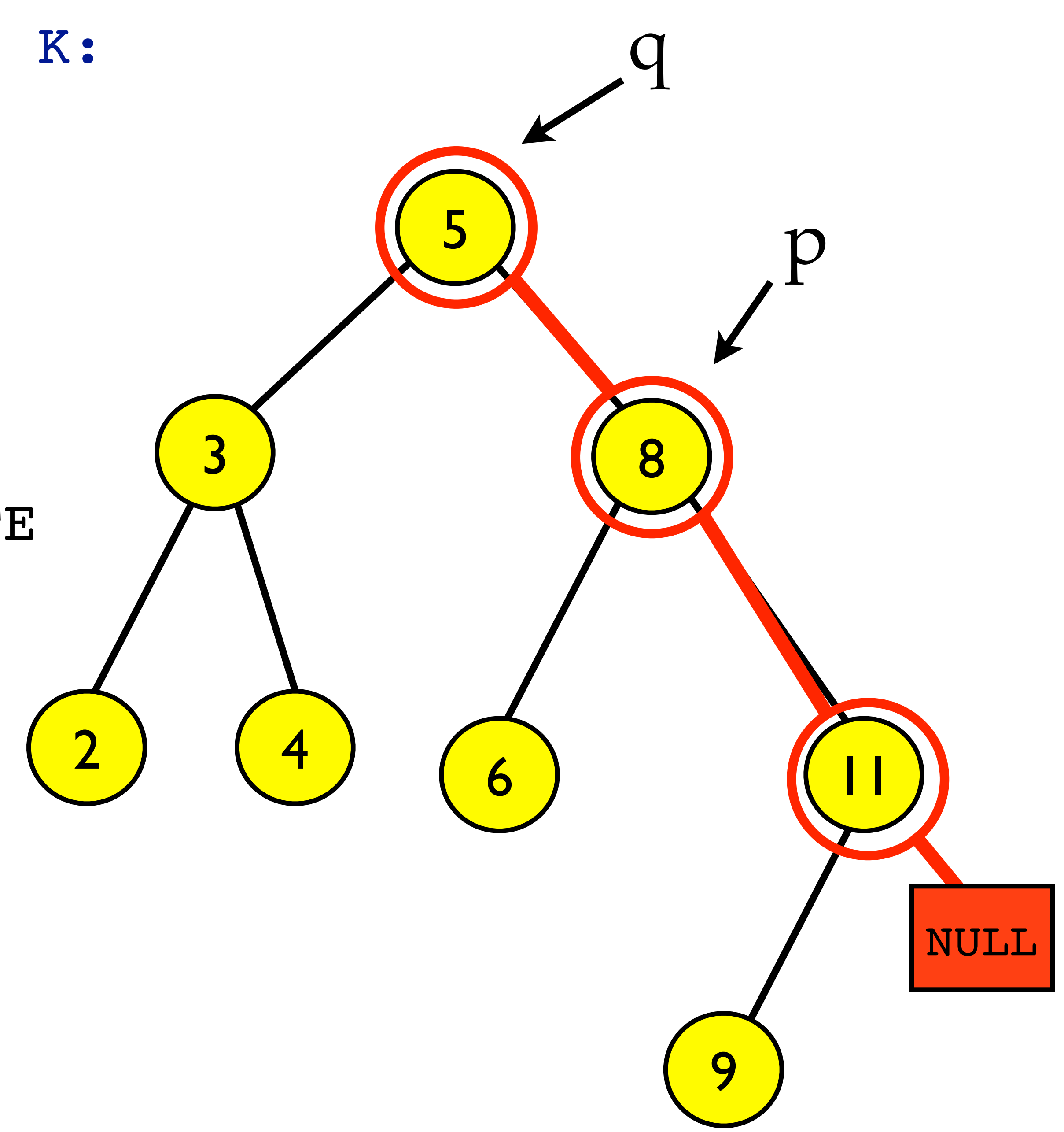
Same idea as BST Find



BST Insert

```
insert(T, K):
    q = NULL
    p = T
    while p != NULL and p.key != K:
        q = p
        if p.key < K:
            p = p.right
        else if p.key > K:
            p = p.left

    if p != NULL: error DUPLICATE

    N = new Node(K)
    if q.key > K:
        q.left = N
    else:
        q.right = N
```

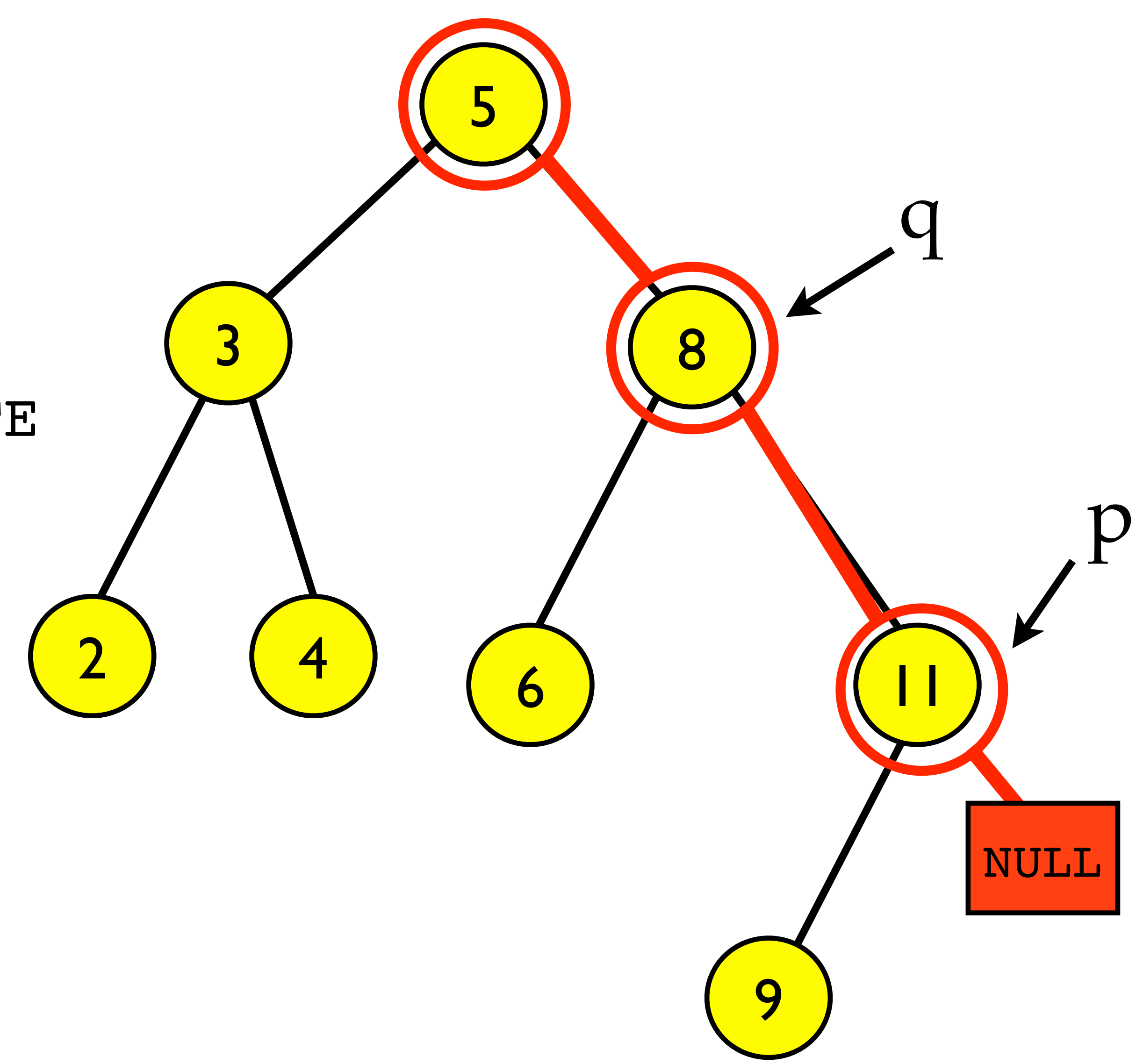
Same idea as BST Find



BST Insert

```
insert(T, K):
   q = NULL
   p = T
   while p != NULL and p.key != K:
      q = p
      if p.key < K:
        p = p.right
      else if p.key > K:
        p = p.left

   if p != NULL: error DUPLICATE

   N = new Node(K)
   if q.key > K:
     q.left = N
   else:
     q.right = N
```

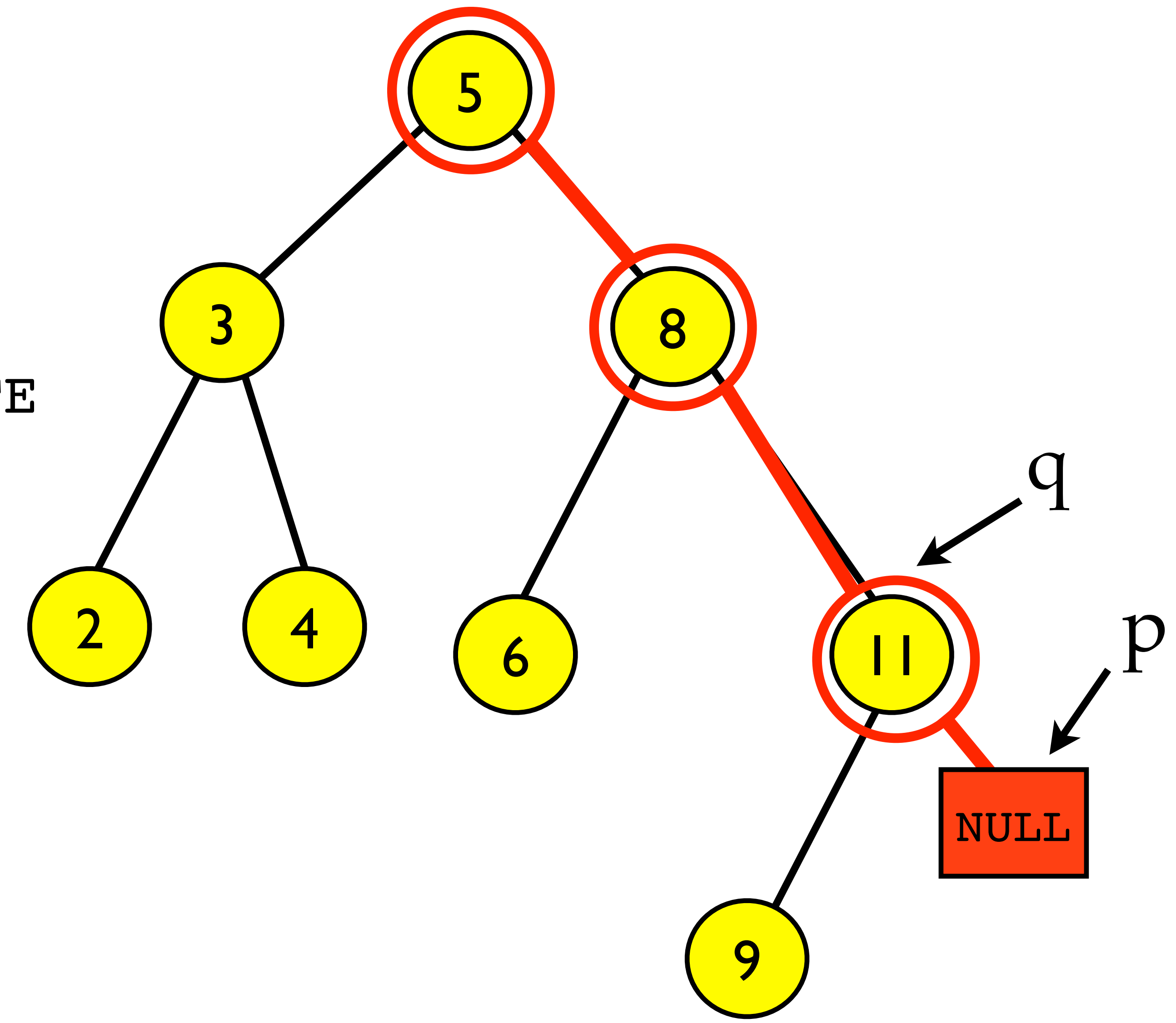
Same idea as BST Find



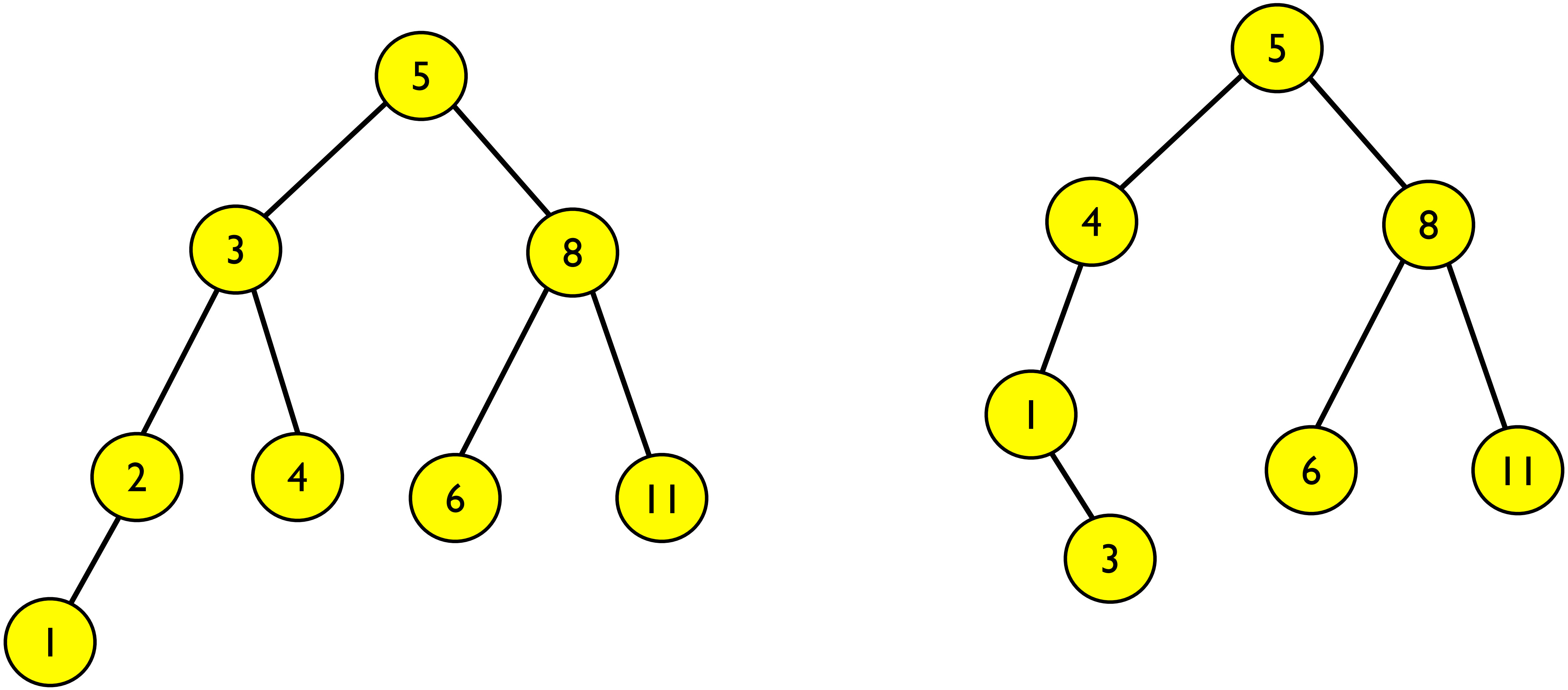
BST Insert

```
insert(T, K):
   q = NULL
   p = T
   while p != NULL and p.key != K:
      q = p
      if p.key < K:
        p = p.right
      else if p.key > K:
        p = p.left

   if p != NULL: error DUPLICATE

   N = new Node(K)
   if q.key > K:
     q.left = N
   else:
     q.right = N
```

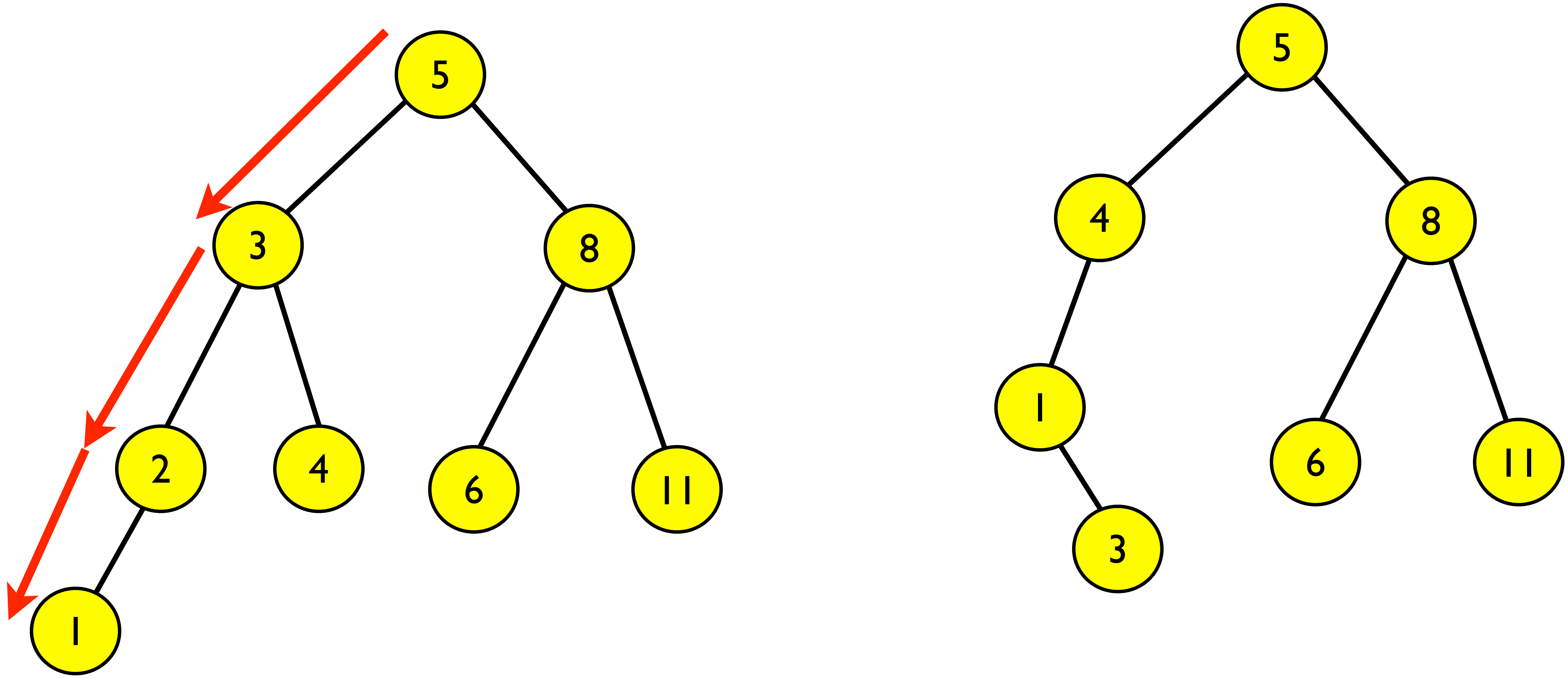
Same idea as BST Find



BST FindMin

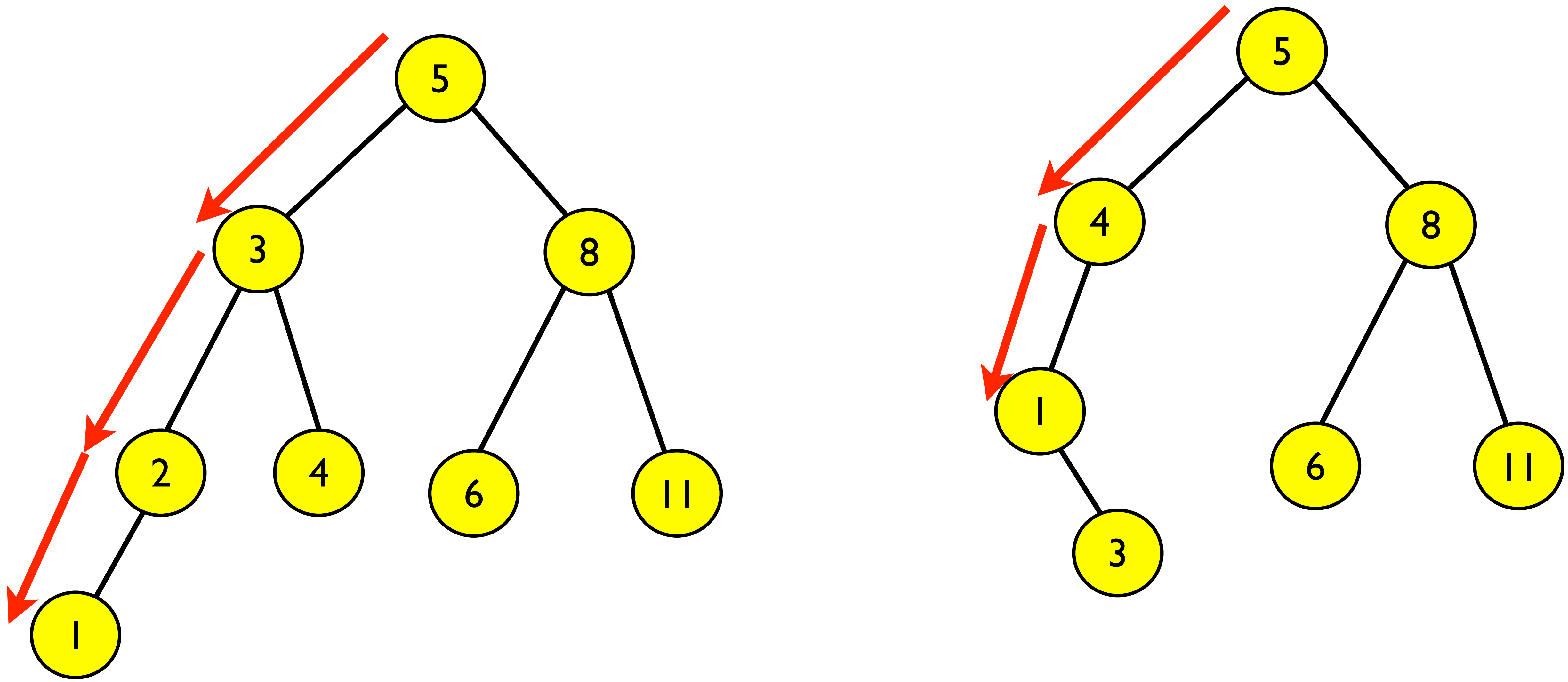


BST FindMin



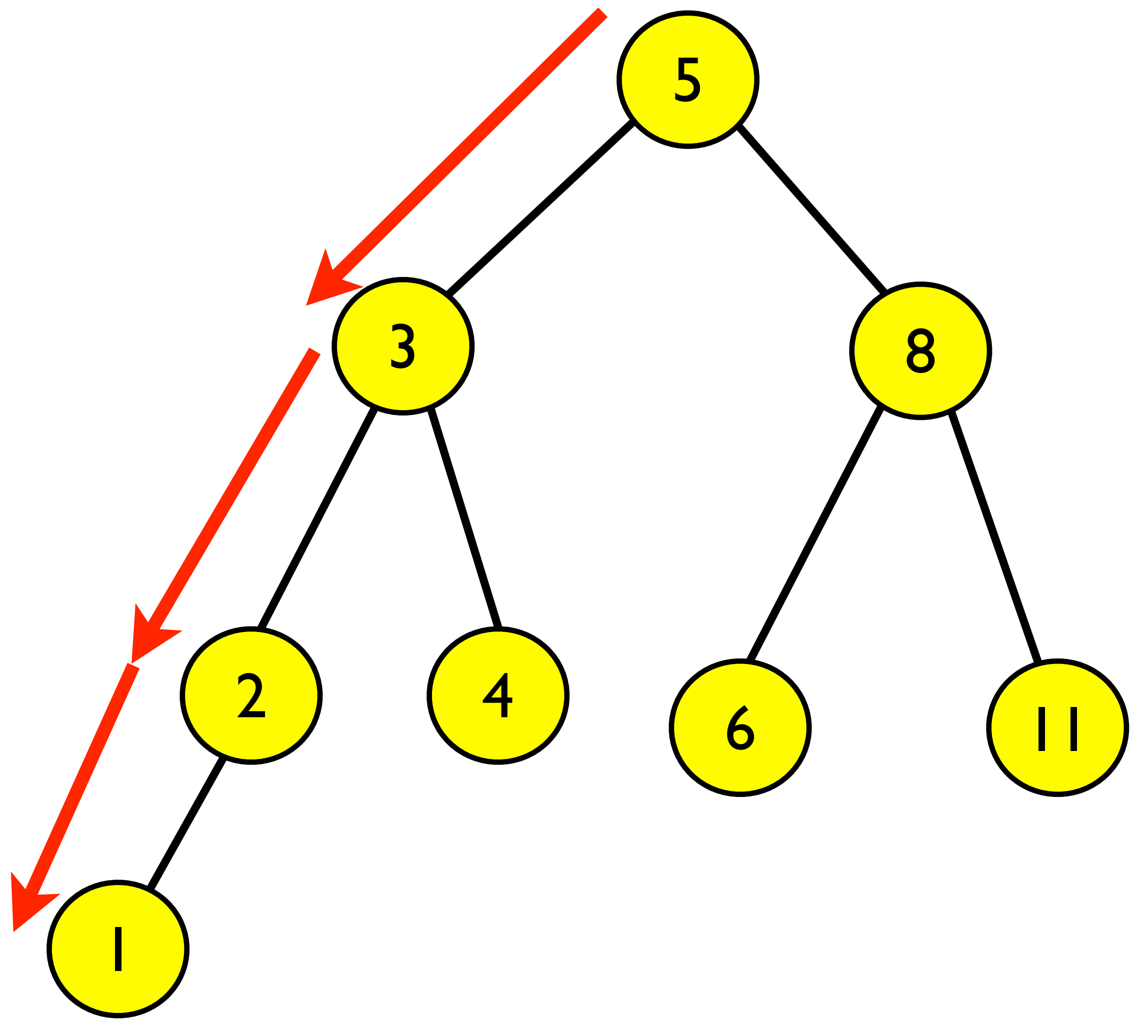
Walk left until you can't go left any more

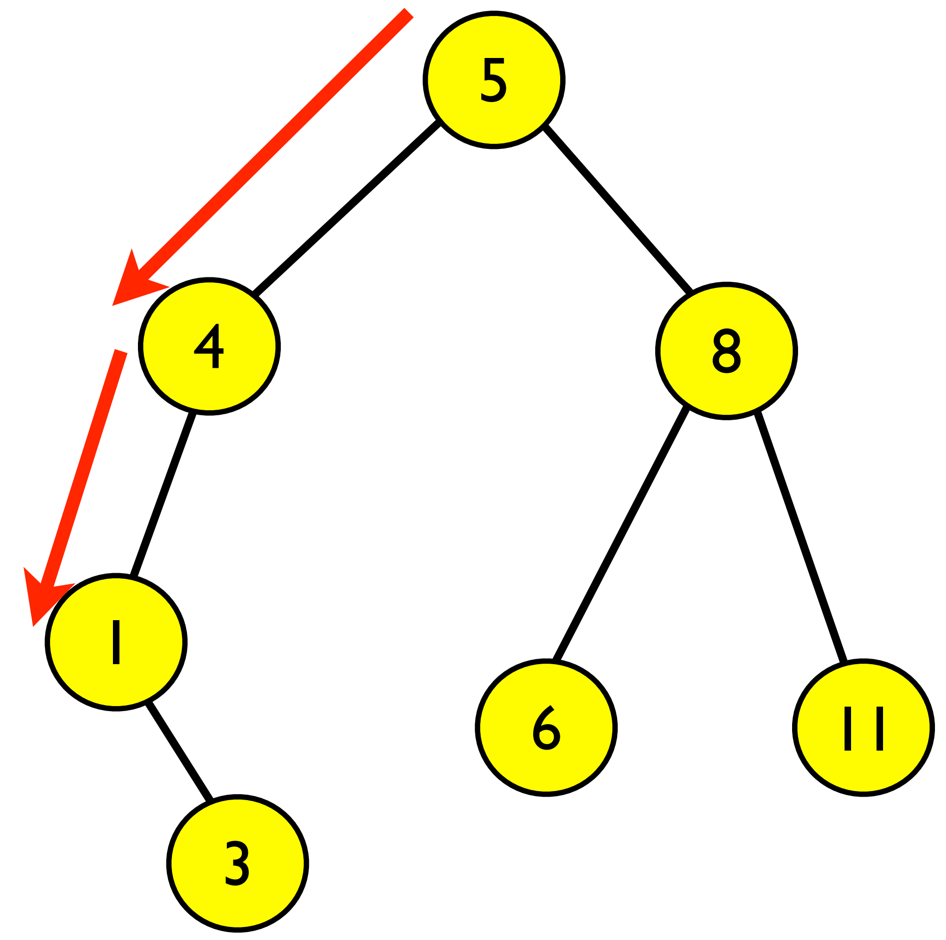
BST FindMin



Walk left until you can't go left any more

BST FindMin



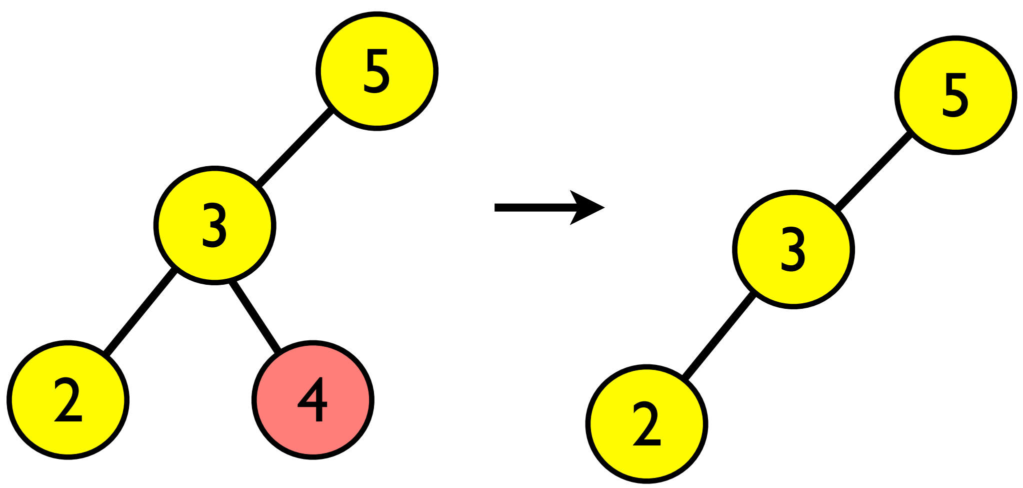


Walk left until you can't go left any more

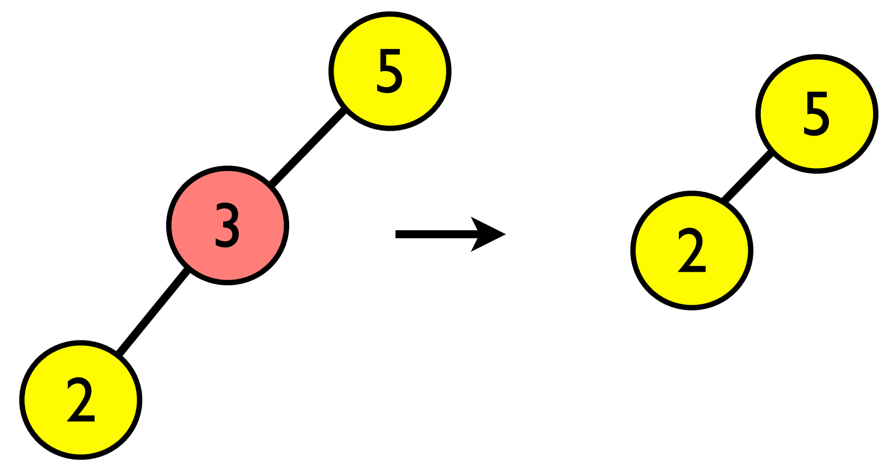
Can you express inorder_successor using find_min?

BST Delete

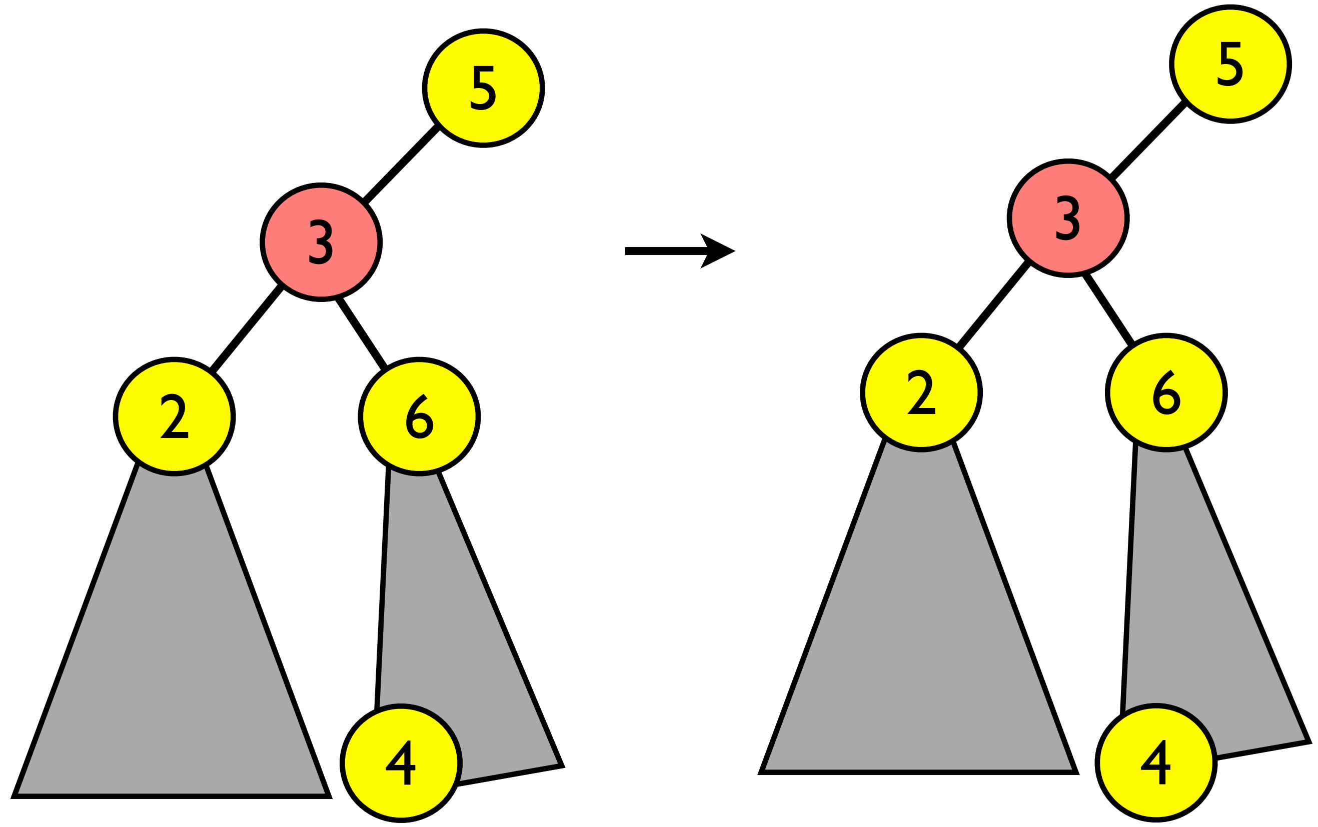
Node is leaf:



Node has 1 child:

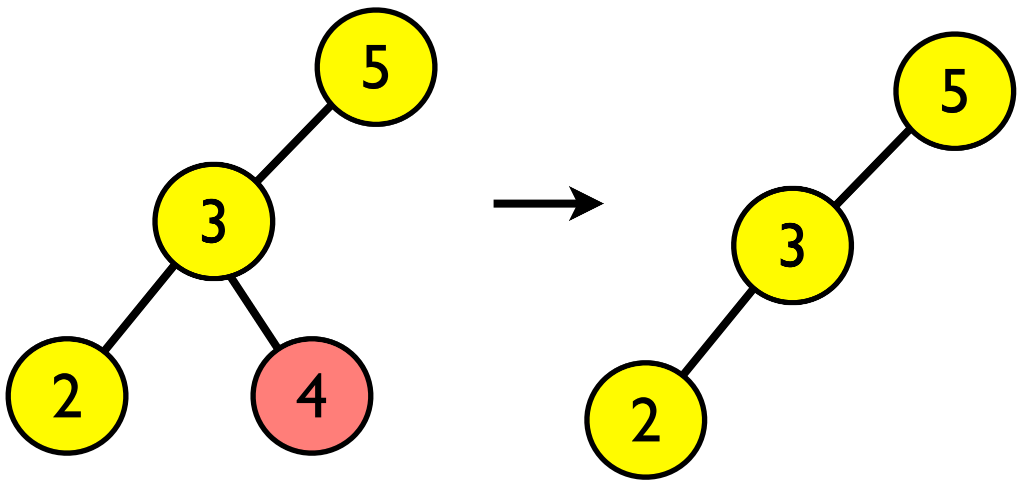


Node has 2 children:

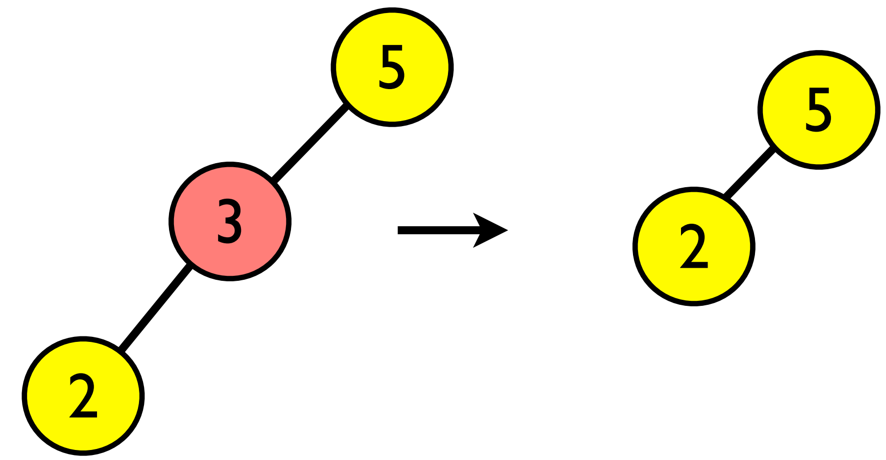


BST Delete

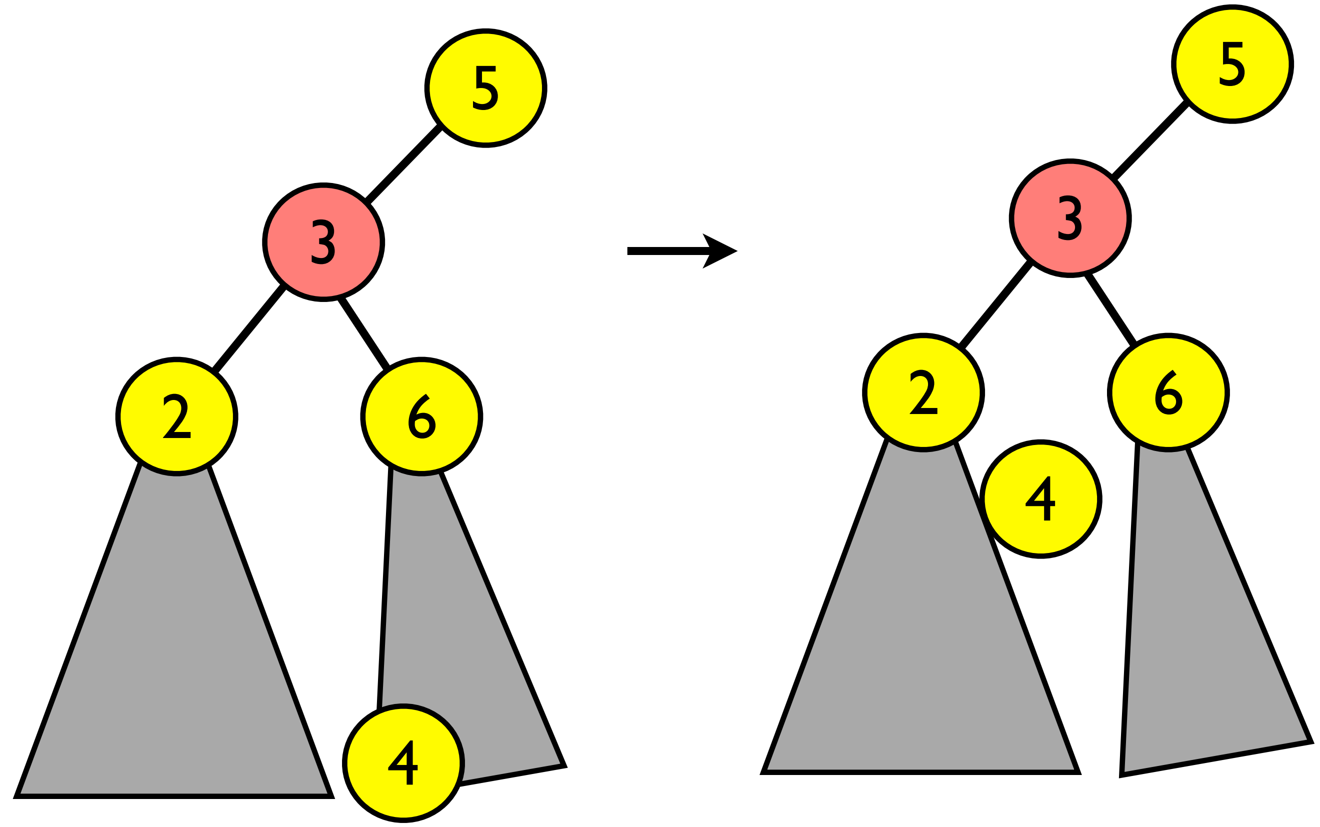
Node is leaf:



Node has 1 child:

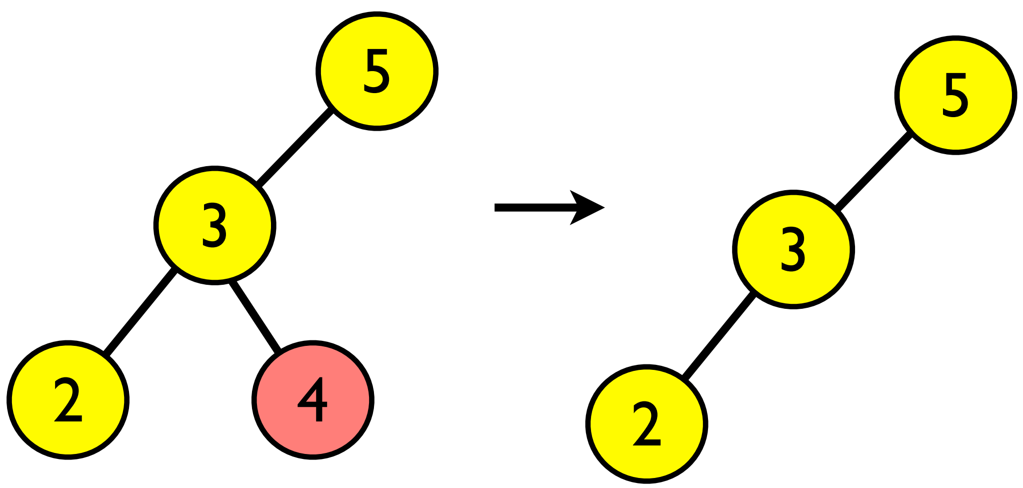


Node has 2 children:

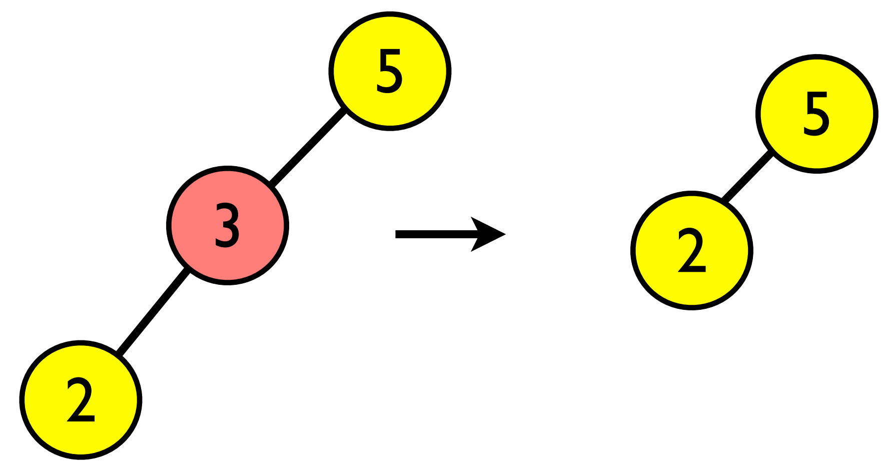


BST Delete

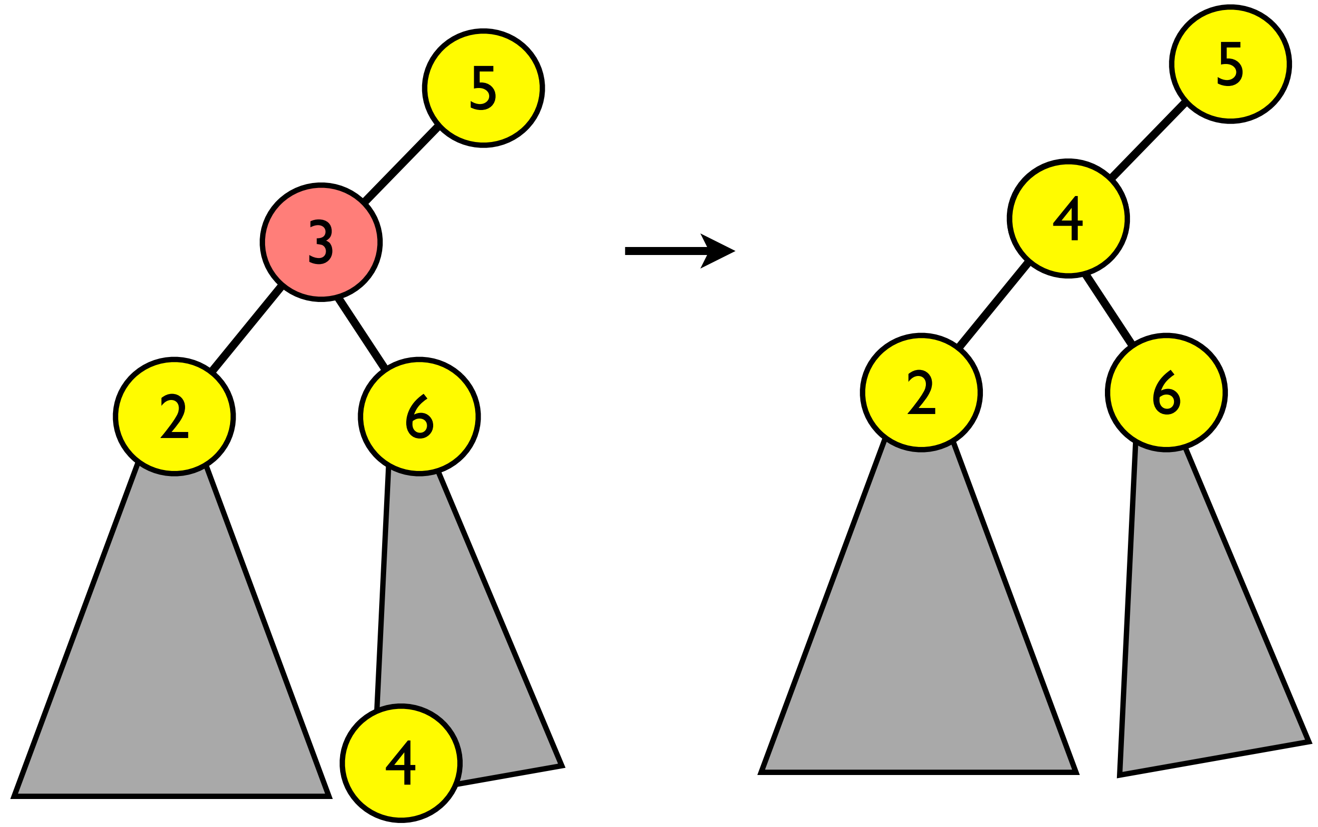
Node is leaf:



Node has 1 child:



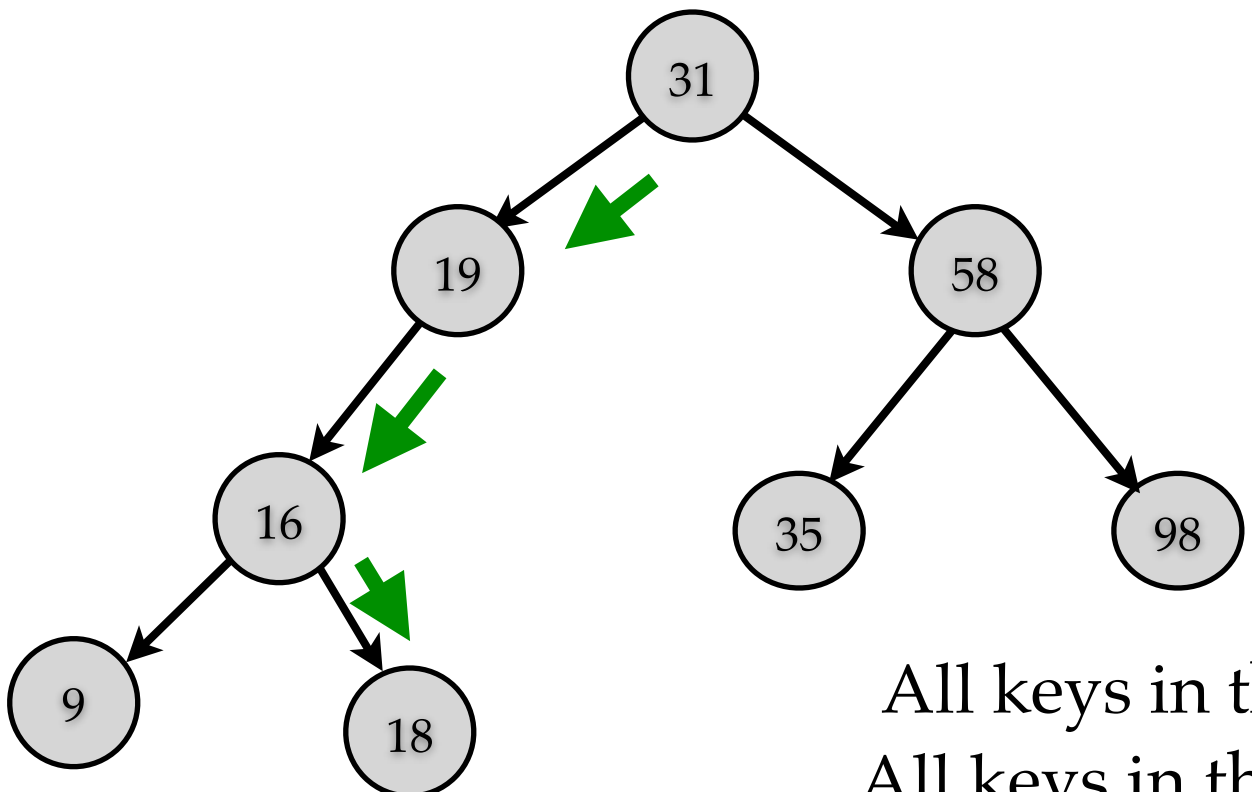
Node has 2 children:



Partitioning

- **Ordering implicitly gives a partitioning based on the "<" relation.**
- **Partitioning usually combined with linking to point to the two halves.**

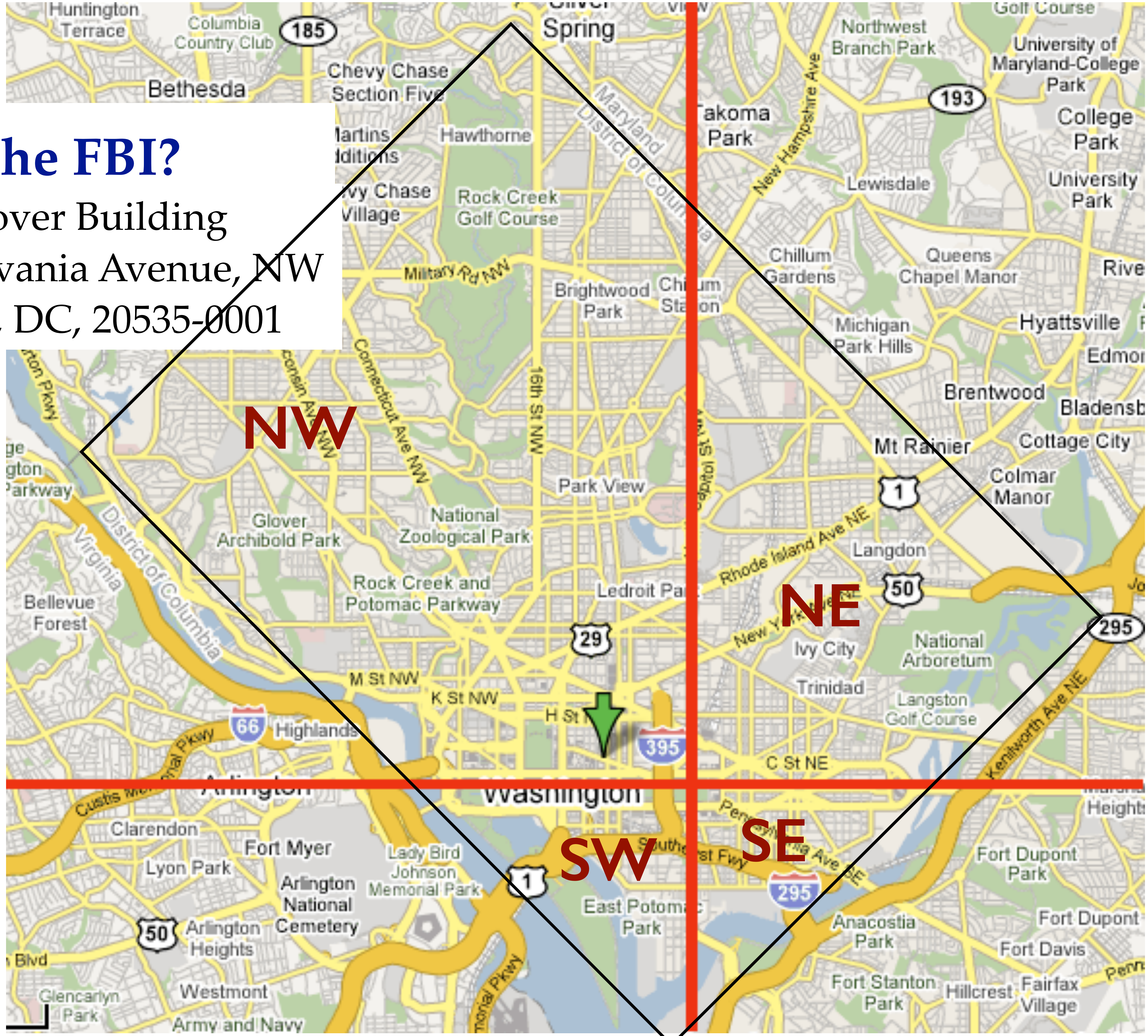
Find 18



All keys in the left subtree are < the root
All keys in the right subtree are ≥ the root

Where’s the FBI?

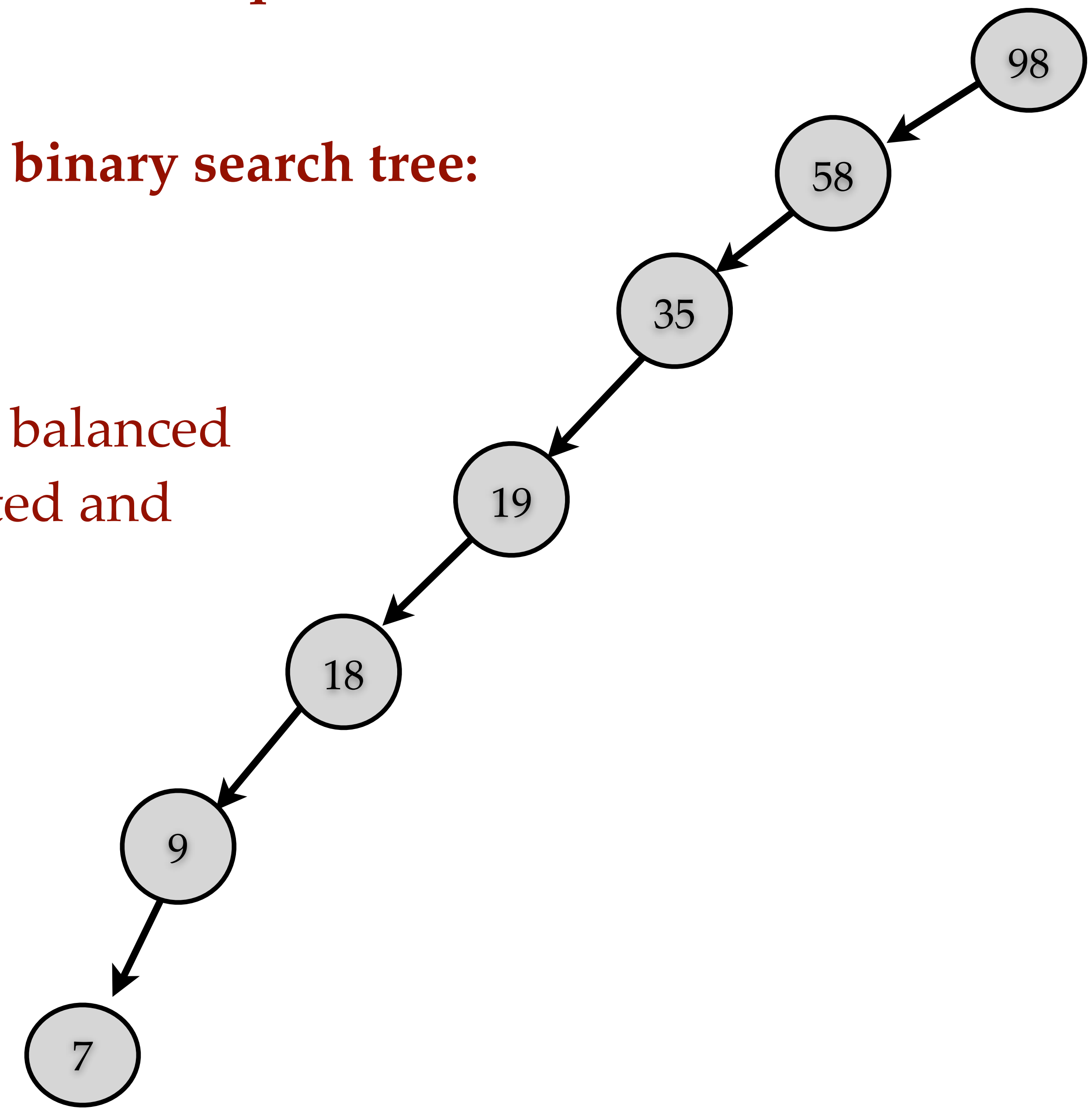
J. Edgar Hoover Building
935 Pennsylvania Avenue, NW
Washington, DC, 20535-0001



Why is the DC partitioning bad?

- **Everything interesting is in the northwest quadrant.**
- **Want a <u>balanced</u> partition!**
- **Another example: an unbalanced binary search tree: (becomes sequential search)**

- How can we force a BST tree to be balanced if items are constantly being inserted and deleted?



Splay Trees (Sleator & Tarjan, 1985)

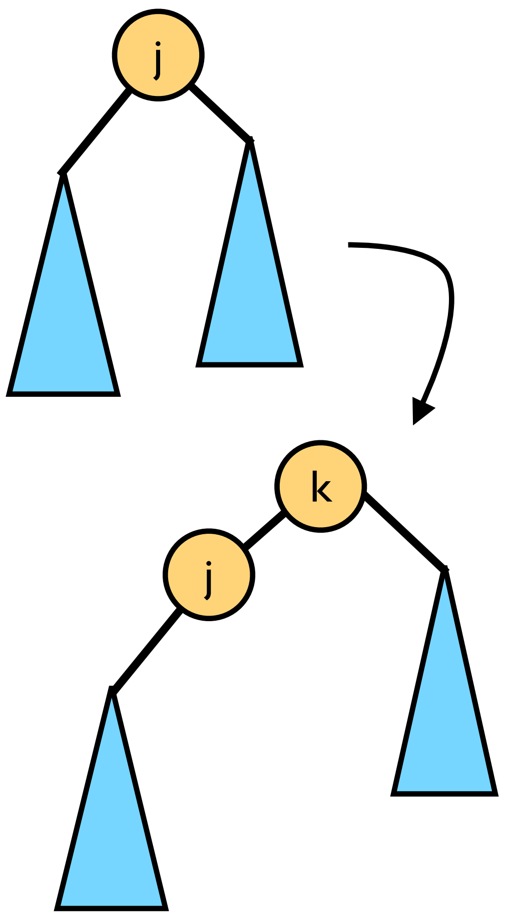
- no extra storage requirement
- simple to implement
- Main idea: move frequently accessed items up in tree
- amortized $O(\log n)$ performance
- worst case single operation is $\Omega(n)$

Splay Trees

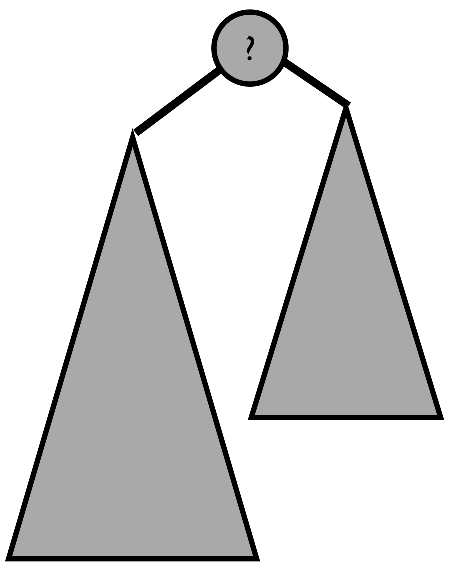
splay(T, *k*): if $k \in T$, then move *k* to the root using a particular set of transformations of the tree. Otherwise, move either the inorder successor or predecessor of *k* to the root.

Without knowing how *splay* is implemented, we can implement the dictionary ADT as follows:

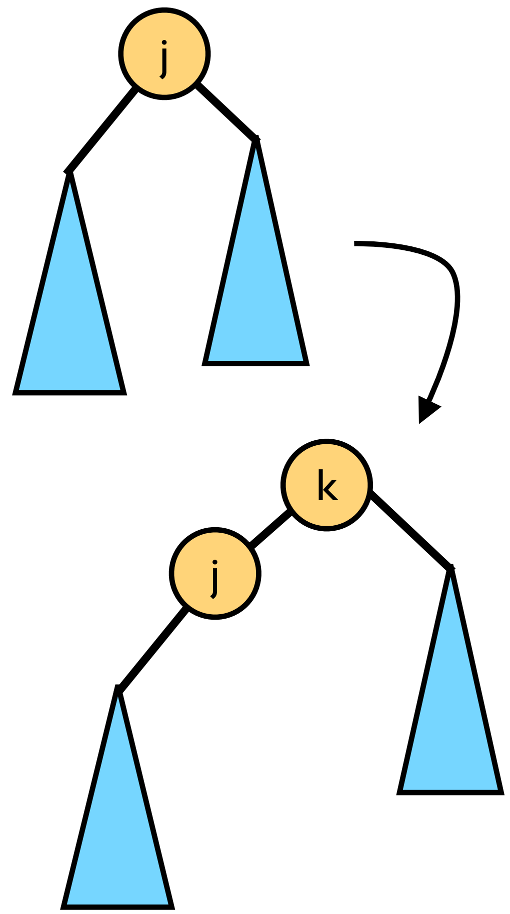
- *find*(T, k): *splay*(T, k). If *root*(T) = *k*, return *k*, otherwise return **not found**.
- *insert*(T, k): *splay*(T, k). If *root*(T) = *k*, return **duplicate!**; otherwise, make *k* the root and add children as in figure:
- *concat*(T_1, T_2): Assumes all keys in T_1 are < all keys in T_2. *Splay*(T_1, ∞). Now root T_1 contains the largest item, and has no right child. Make T_2 right child of T_1.
- *delete*(T, *k*): *splay*(T, *k*). If root *r* contains *k*, *concat*(LEFT(*r*), RIGHT(*r*)).



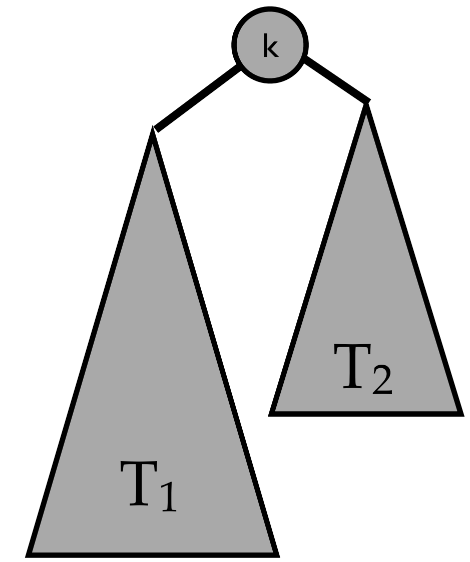
Dictionary Operations, in pictures



find(T, *k*): splay & check root



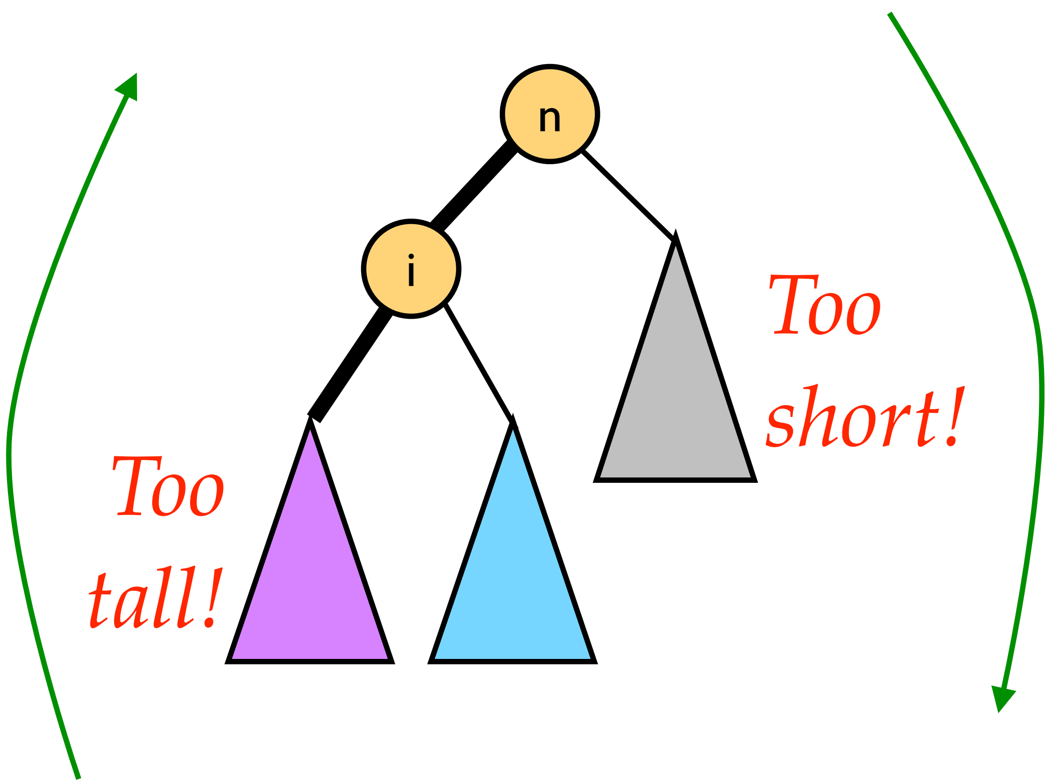
insert(T, k): splay and insert just below root



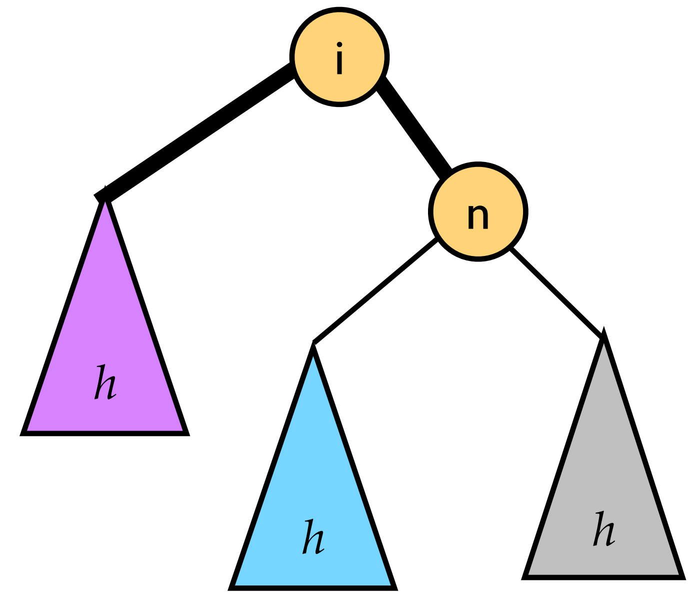
delete(T, *k*): splay & concat left & right subtrees

Implementing the Splay operation

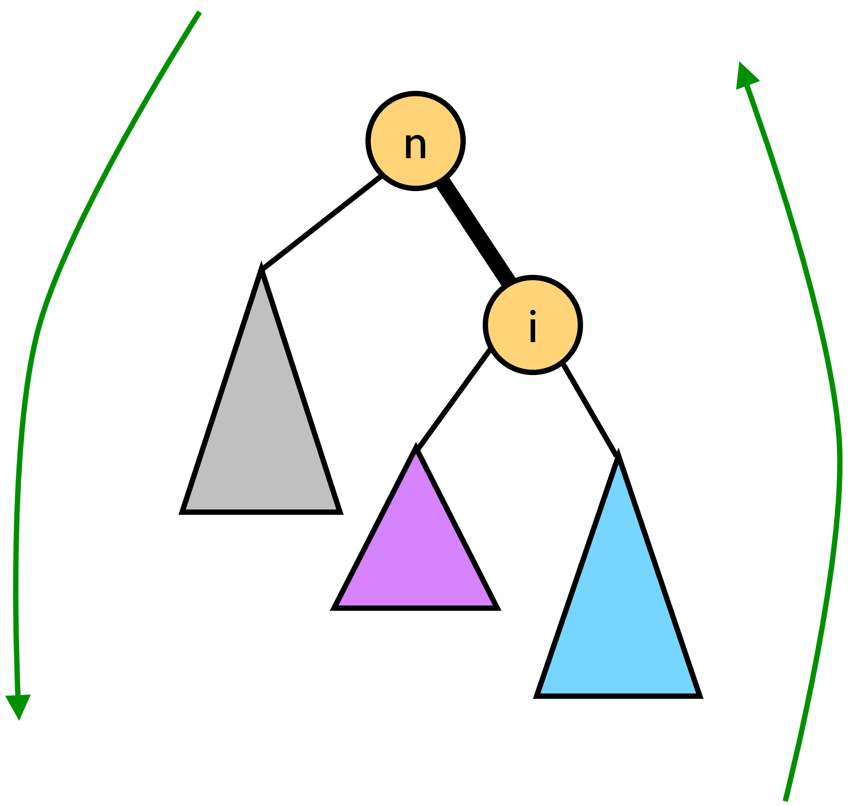
Right rotation (at n)

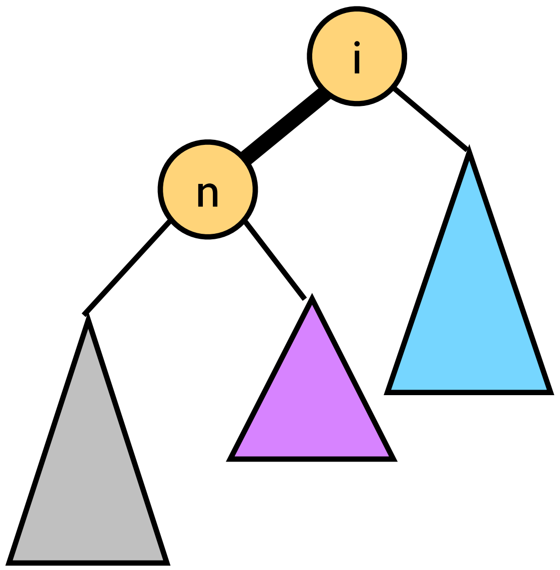


Right rotation
(aka clockwise rotation)



Left Rotation (at n)

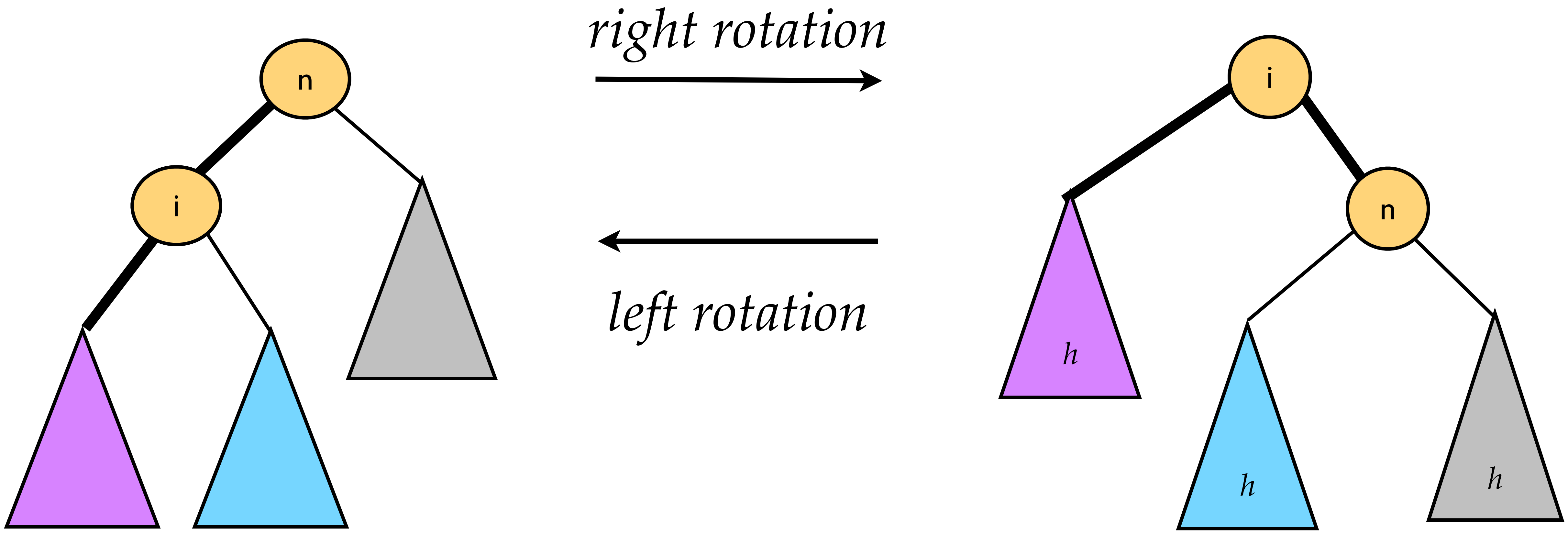




Left rotation
(aka counterclockwise rotation)

Only a constant # of pointers need to be updated for a rotation: O(1) time

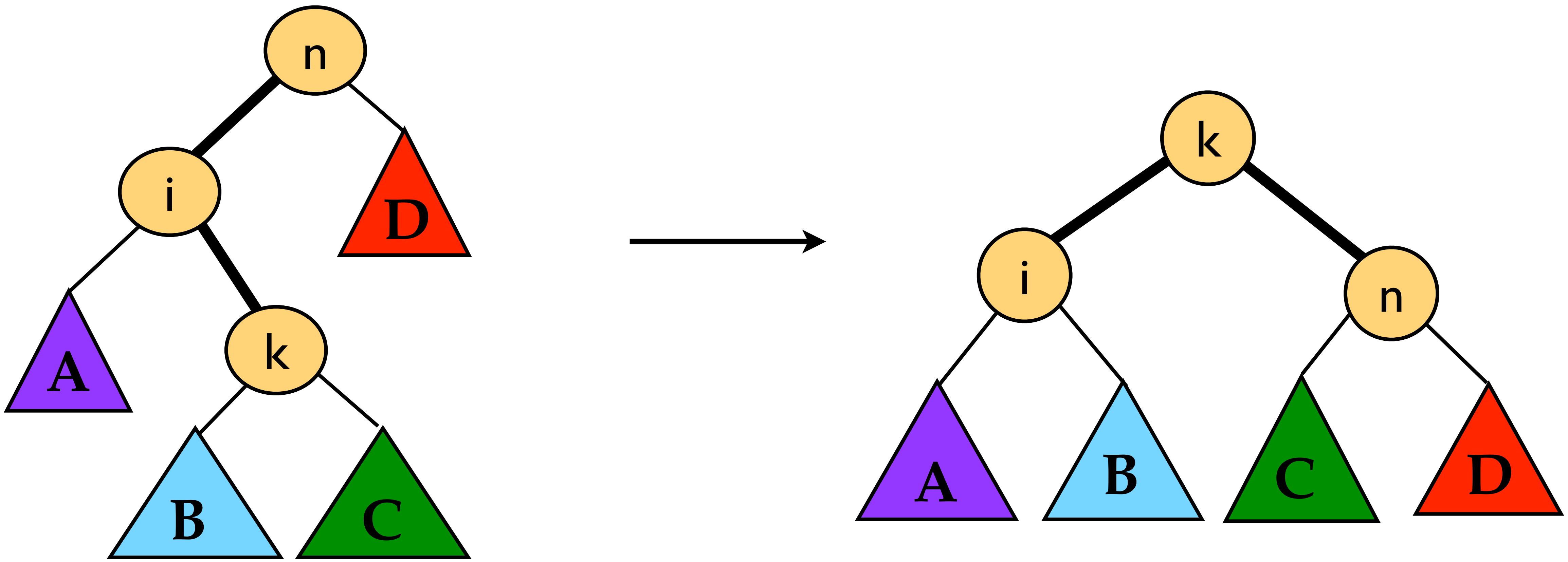
Right & Left Rotations are Inverses



i moves toward the root

n moves toward the root

Double Rotation



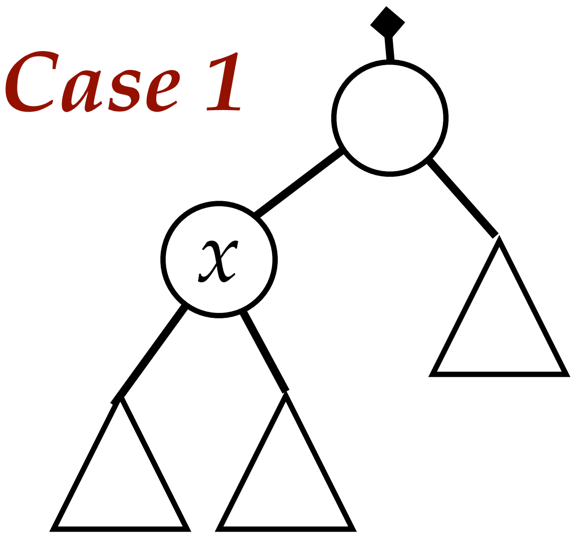
k moves toward the root

Splay Operation

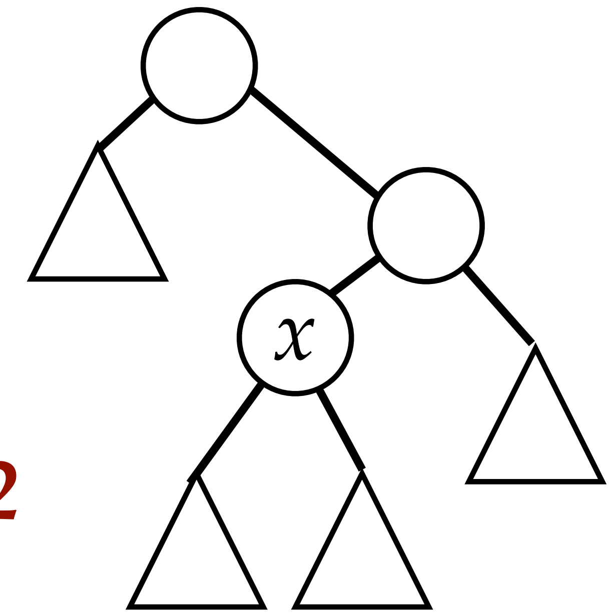
- *Splay*(T, *k*): find *k*, walk back up root. Let x be the current node.
- Cases:
  1. x has no grandparent
  2. x is left child of parent(x), which is the right child of parent^2(x).
  3. x is left child of parent(x), which is the left child of parent(parent(x)) = parent^2(x)

Rotations with goal: move x toward the root

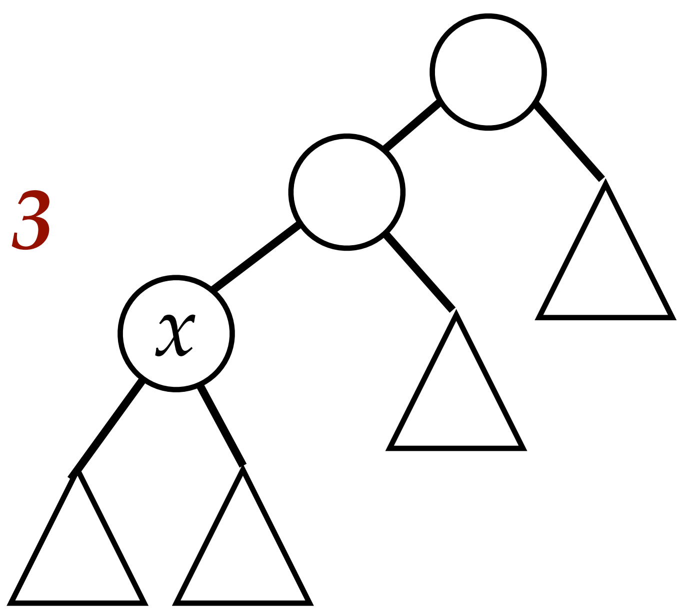
Case 1



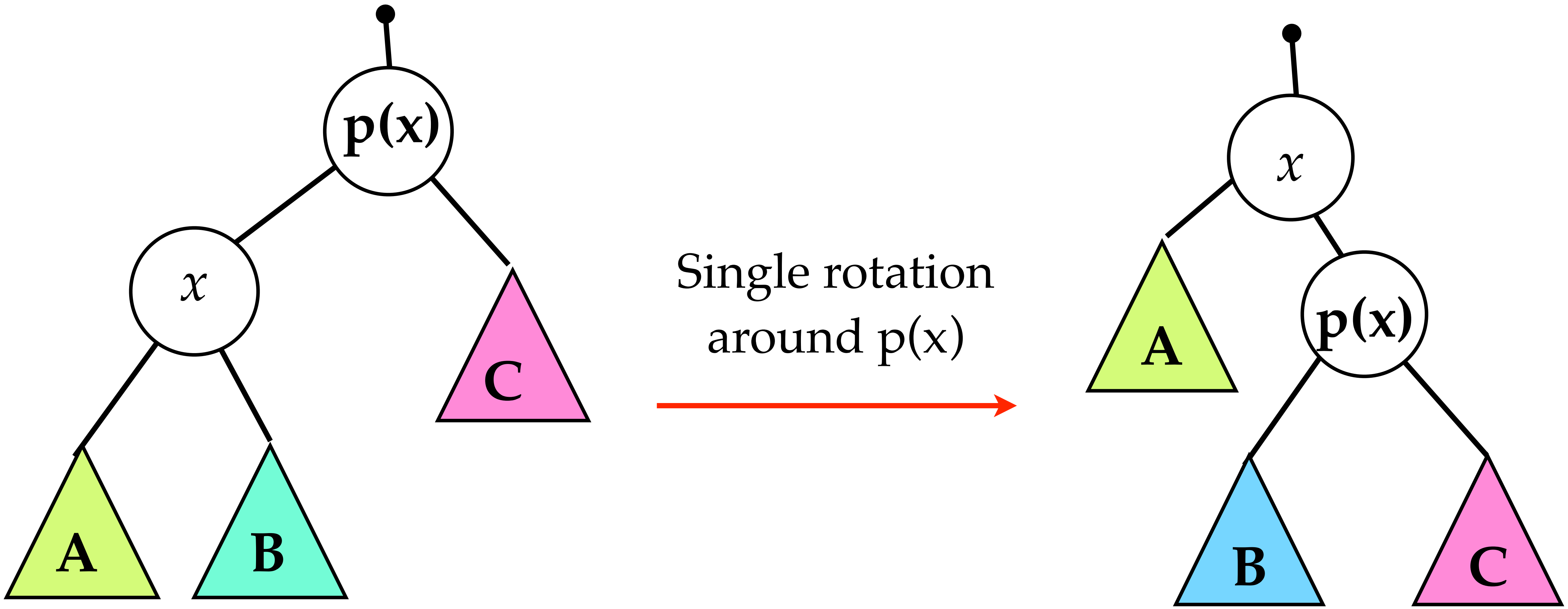
Case 2



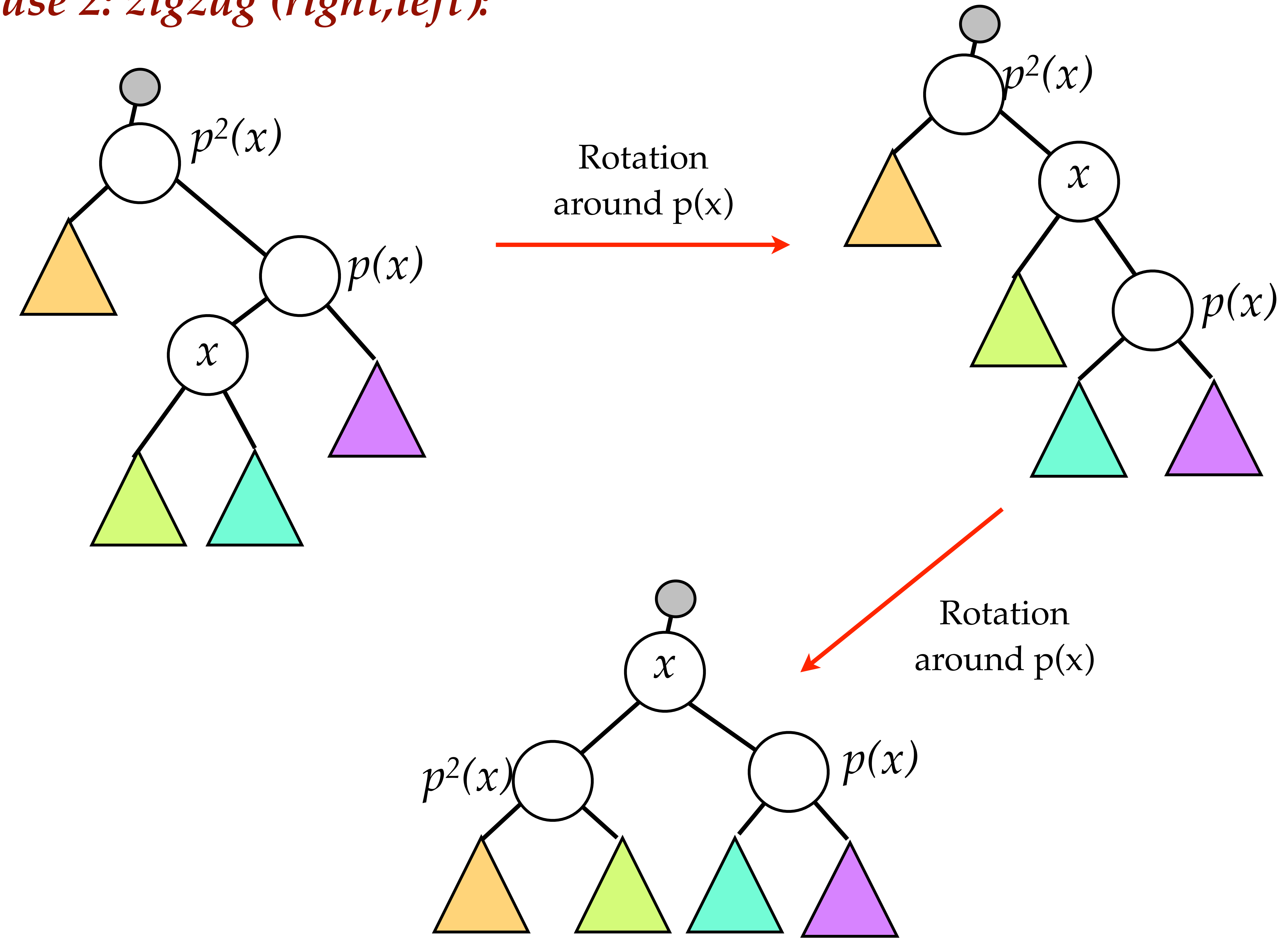
Case 3



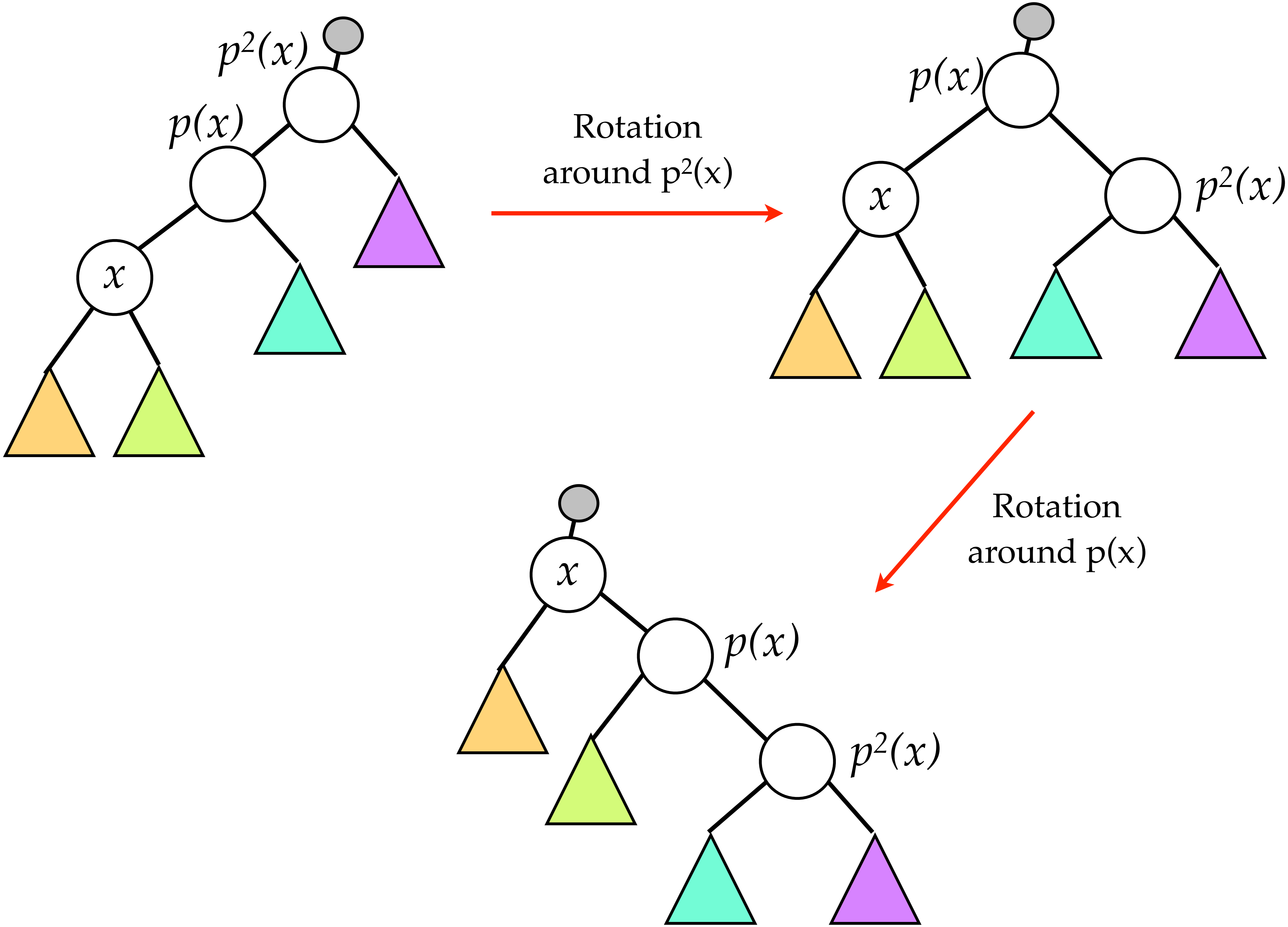
Case 1: no grandparent:



Case 2: zigzag (right,left):



Case 3: zigzig (left, left):



Splay Idea

- The find/insert/delete operations can be written in terms of the "splay" operation.

- Splay is implemented by doing a standard BST "find" and then applying particular rotations **walking back up toward the root.**

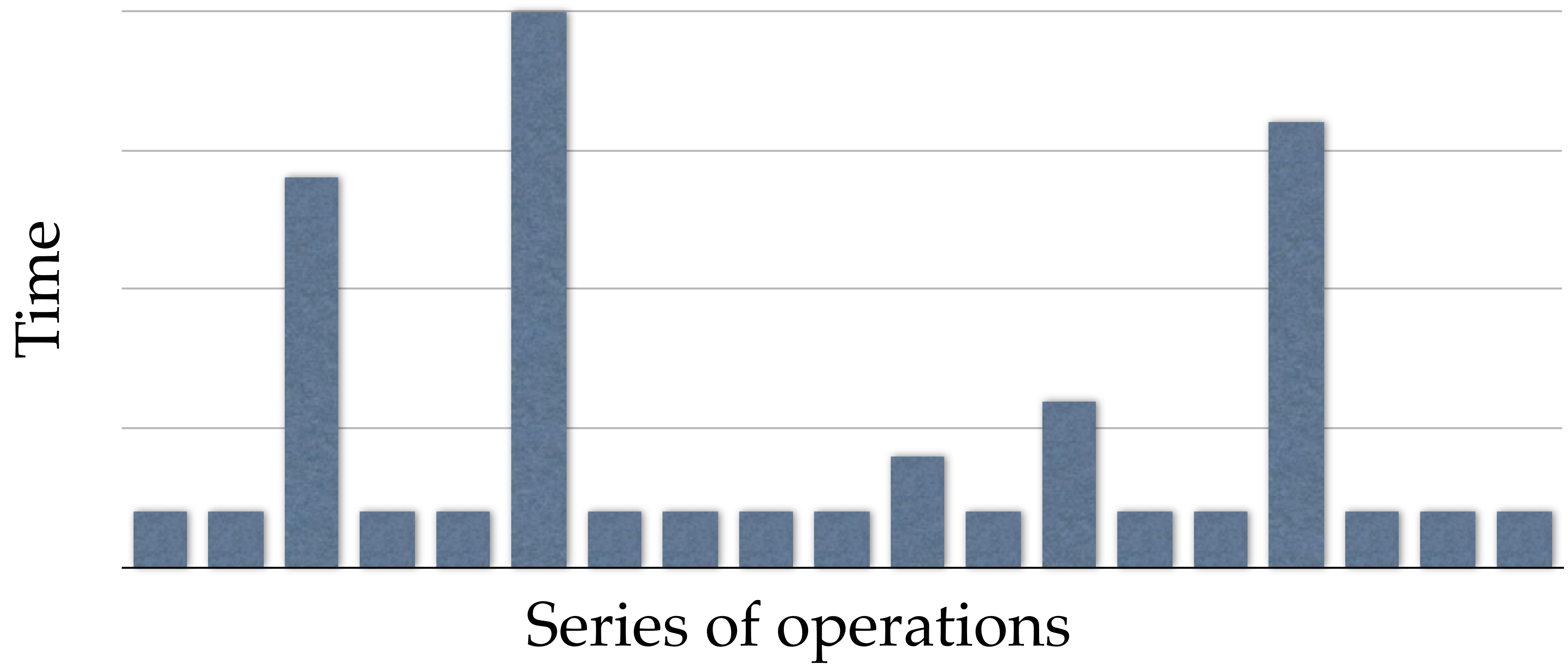
- This is somewhat like the idea of "path compression" for the tree-based union-find data structure: during a find, you flatten out the tree.

- Here: splay may actually make the tree worse, but over a series of operations the tree always gets better (e.g. a slow find results in a long splay, but this long splay tends to flatten the tree a lot).

Splay Notes

- Might make tree less balanced
- Might make tree taller
- So, how can they be good?

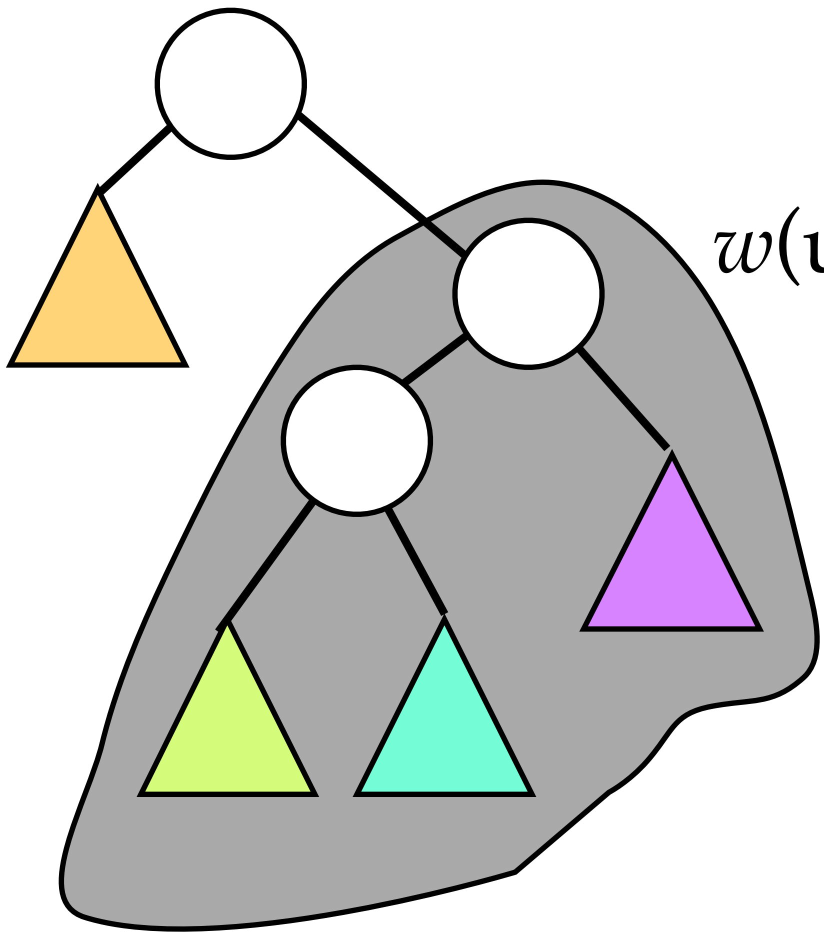
Amortized Analysis – Concept



- Some operations will be costly, some will be cheap
- Total area of *m* bars bounded by some function f(*m*,*n*).
  - m = number of operations, n = number of elements
- E.g. if area = O(m log n), each operation takes O(log n) amortized time

Analysis of the Splay operation

Node Ranks & Money Invariant



w(u) := weight of u

:= # nodes in the subtree rooted at u

rank(u) := [log $w(u)$]

[x] means floor(x)

Money Invariant: we will always keep *rank*(u) dollars stored at every node.

Each rotation/double rotation costs \$1.

O(1) amount of work

Also have to spend \$ to maintain invariant.

Idea:

> **Thm.** It costs at most 3[log n] + 1 new dollars to splay, keeping the money invariant

- So, for every splay, we're going to spend O(log n) new dollars; we might do more *work* than that if we use some of the $ already in the tree.
- If we start with an empty tree, after m splay operations, we'll have spent ≤ m(3[log n] +1) dollars.
- The dollars pay for both:
  - the money invariant
  - cost of all the rotations (time)
- So, total time for m splay operations is O(m log n).

Thm. It costs 3[log n] + 1 new dollars to splay, keeping the money invariant.

Suppose zig/zigzig/zigzag at x costs the following:

zig: $3(\mathit{rank}^1(x) - \mathit{rank}(x)) + 1$

zigzag: $3(\mathit{rank}^1(x) - \mathit{rank}(x))$

zigzig: $3(\mathit{rank}^1(x) - \mathit{rank}(x))$

Then cost of a whole splay =

$$
\begin{aligned}
&3(\mathit{rank}^1(x) - \mathit{rank}(x)) \\
&+ 3(\mathit{rank}^2(x) - \mathit{rank}^1(x)) \\
&+ 3(\mathit{rank}^3(x) - \mathit{rank}^2(x)) \\
&\\
&+ 3(\mathit{rank}^k(x) - \mathit{rank}^{(k-1)}(x)) + 1
\end{aligned}
$$

Then cost of a whole splay

$$
\begin{aligned}
&= 3(\mathit{rank}^k(x) - \mathit{rank}(x)) + 1 \\
&\leq 3(\mathit{rank}^k(x)) + 1 \\
&\leq 3[\log n] + 1
\end{aligned}
$$

Thm. It costs 3[log n] + 1 new dollars to splay, keeping the money invariant.

Suppose zig/zigzig/zigzag at x costs the following:

$$\text{zig:}\quad 3(\mathit{rank}^1(x) - \mathit{rank}(x)) + 1$$
$$\text{zigzag:}\quad 3(\mathit{rank}^1(x) - \mathit{rank}(x))$$
$$\text{zigzig:}\quad 3(\mathit{rank}^1(x) - \mathit{rank}(x))$$

Then cost of a whole splay =

$$\begin{aligned}
&3(\cancel{\mathit{rank}^1(x)} - \mathit{rank}(x)) \\
&+ 3(\mathit{rank}^2(x) - \cancel{\mathit{rank}^1(x)}) \\
&+ 3(\mathit{rank}^3(x) - \mathit{rank}^2(x)) \\
&+ 3(\mathit{rank}^k(x) - \mathit{rank}^{(k-1)}(x)) + 1
\end{aligned}$$

Then cost of a whole splay

$$= 3(\mathit{rank}^k(x) - \mathit{rank}(x)) + 1$$
$$\leq 3(\mathit{rank}^k(x)) + 1$$
$$\leq 3[\log n] + 1$$

Thm. It costs 3[log n] + 1 new dollars to splay, keeping the money invariant.

Suppose zig/zigzig/zigzag at x costs the following:

$$
\begin{aligned}
\text{zig:}\quad & 3(\mathit{rank}^1(x) - \mathit{rank}(x)) + 1 \\
\text{zigzag:}\quad & 3(\mathit{rank}^1(x) - \mathit{rank}(x)) \\
\text{zigzig:}\quad & 3(\mathit{rank}^1(x) - \mathit{rank}(x))
\end{aligned}
$$

Then cost of a whole splay =

$$
\begin{aligned}
& 3(\cancel{\mathit{rank}^1(x)} - \mathit{rank}(x)) \\
+\ & 3(\cancel{\mathit{rank}^2(x)} - \cancel{\mathit{rank}^1(x)}) \\
+\ & 3(\mathit{rank}^3(x) - \cancel{\mathit{rank}^2(x)}) \\
\\
+\ & 3(\mathit{rank}^{k}(x) - \mathit{rank}^{(k-1)}(x)) + 1
\end{aligned}
$$

Then cost of a whole splay

$$
\begin{aligned}
&= 3(\mathit{rank}^k(x) - \mathit{rank}(x)) + 1 \\
&\leq 3(\mathit{rank}^k(x)) + 1 \\
&\leq 3[\log n] + 1
\end{aligned}
$$

Thm. It costs 3[log n] + 1 new dollars to splay, keeping the money invariant.

Suppose zig/zigzig/zigzag at x costs the following:

zig: $3(rank^1(x) - rank(x)) + 1$

zigzag: $3(rank^1(x) - rank(x))$

zigzig: $3(rank^1(x) - rank(x))$

Then cost of a whole splay =

$$
\begin{aligned}
& 3(\cancel{rank^1(x)} - rank(x)) \\
&+ 3(\cancel{rank^2(x)} - \cancel{rank^1(x)}) \\
&+ 3(\cancel{rank^3(x)} - \cancel{rank^2(x)}) \\
&\\
&+ 3(rank^k(x) - \cancel{rank^{(k-1)}(x)}) + 1
\end{aligned}
$$

Then cost of a whole splay

$= 3(rank^k(x) - rank(x)) + 1$

$\leq 3(rank^k(x)) + 1$

$\leq$ 3[log n] + 1

Thm. It costs 3[log n] + 1 new dollars to splay, keeping the money invariant.

Suppose zig/zigzig/zigzag at x costs the following:

zig: $3(rank^1(x) - rank(x)) + 1$
zigzag: $3(rank^1(x) - rank(x))$
zigzig: $3(rank^1(x) - rank(x))$

Then cost of a whole splay =

$$
\begin{aligned}
& 3(\cancel{rank^1(x)} - rank(x)) \\
&+ 3(\cancel{rank^2(x)} - \cancel{rank^1(x)}) \\
&+ 3(\cancel{rank^3(x)} - \cancel{rank^2(x)}) \\
&+ 3(rank^k(x) - \cancel{rank^{(k-1)}(x)}) + 1
\end{aligned}
$$

Telescoping sum

Then cost of a whole splay

$= 3(rank^k(x) - rank(x)) + 1$

$\leq 3(rank^k(x)) + 1$

$\leq 3[\log n] + 1$

Thm. It costs 3[log n] + 1 new dollars to splay, keeping the money invariant.

Suppose zig/zigzig/zigzag at x costs the following:

$$\text{zig:}\quad 3(\mathit{rank}^1(x) - \mathit{rank}(x)) + 1$$
$$\text{zigzag:}\quad 3(\mathit{rank}^1(x) - \mathit{rank}(x))$$
$$\text{zigzig:}\quad 3(\mathit{rank}^1(x) - \mathit{rank}(x))$$

Then cost of a whole splay =

$$
\begin{aligned}
&3(\cancel{\mathit{rank}^1(x)} - \mathit{rank}(x)) \\
&+ 3(\cancel{\mathit{rank}^2(x)} - \cancel{\mathit{rank}^1(x)}) \\
&+ 3(\cancel{\mathit{rank}^3(x)} - \cancel{\mathit{rank}^2(x)}) \\
&\\
&+ 3(\mathit{rank}^k(x) - \cancel{\mathit{rank}^{(k-1)}(x)}) + 1
\end{aligned}
$$

Telescoping sum

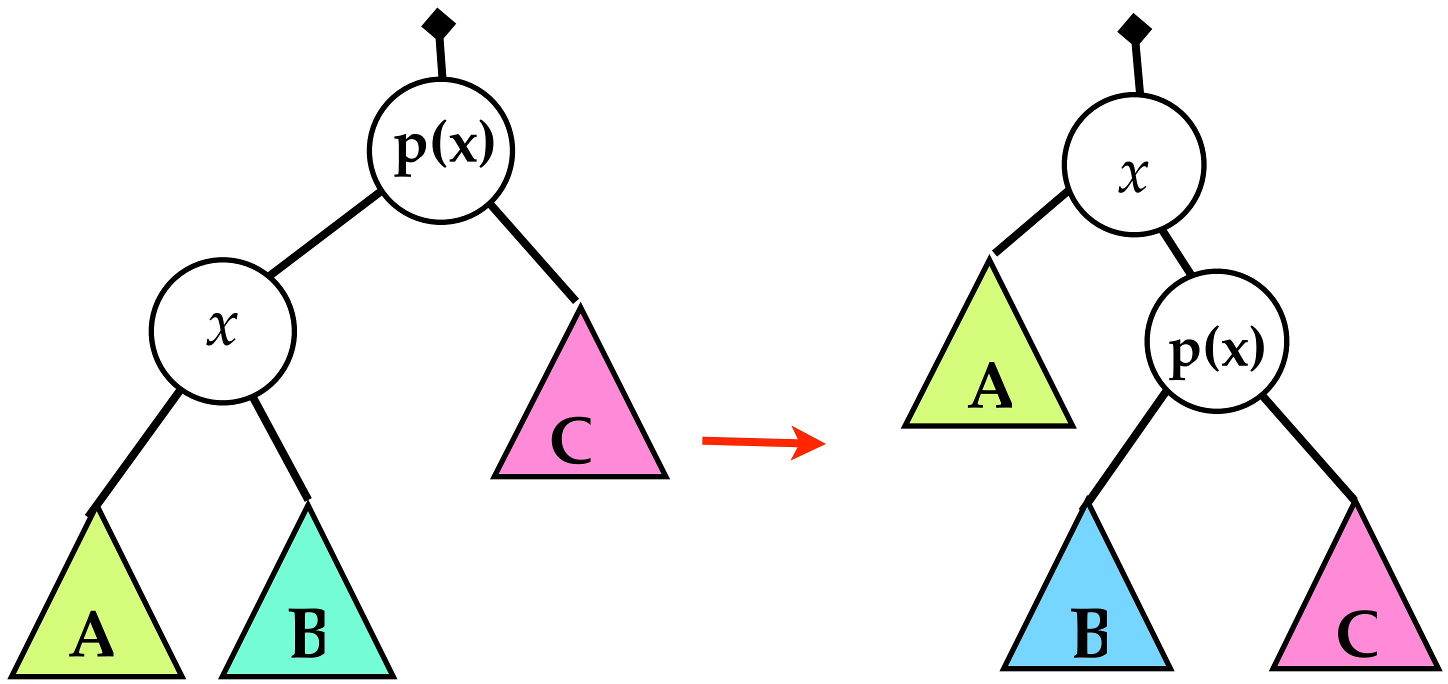
Then cost of a whole splay

$= 3(\mathit{rank}^k(x) - \mathit{rank}(x)) + 1$

$\leq 3(\mathit{rank}^k(x)) + 1$ $\quad$ *$\mathit{rank}^k(x)$ = after k steps, x is at the root*

$\leq 3[\log n] + 1$

case 1: $3(rank^1(x) - rank(x)) + 1$



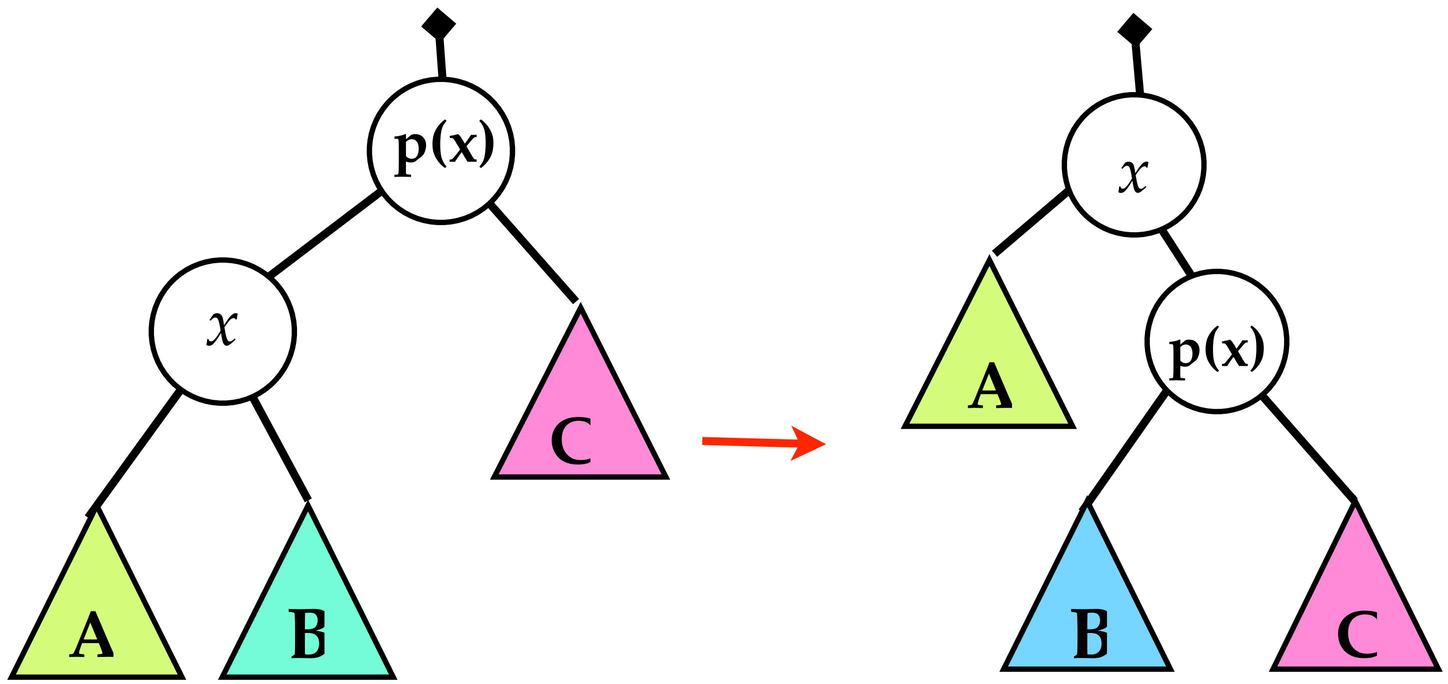
+1 pays for the rotation

$rank^1(x) = rank(p(x))$

Extra $ to keep the invariant is:

$rank^1(x) + rank^1(p(x)) - (rank(x) + rank(p(x))$

case 1: $3(rank^1(x) - rank(x)) + 1$



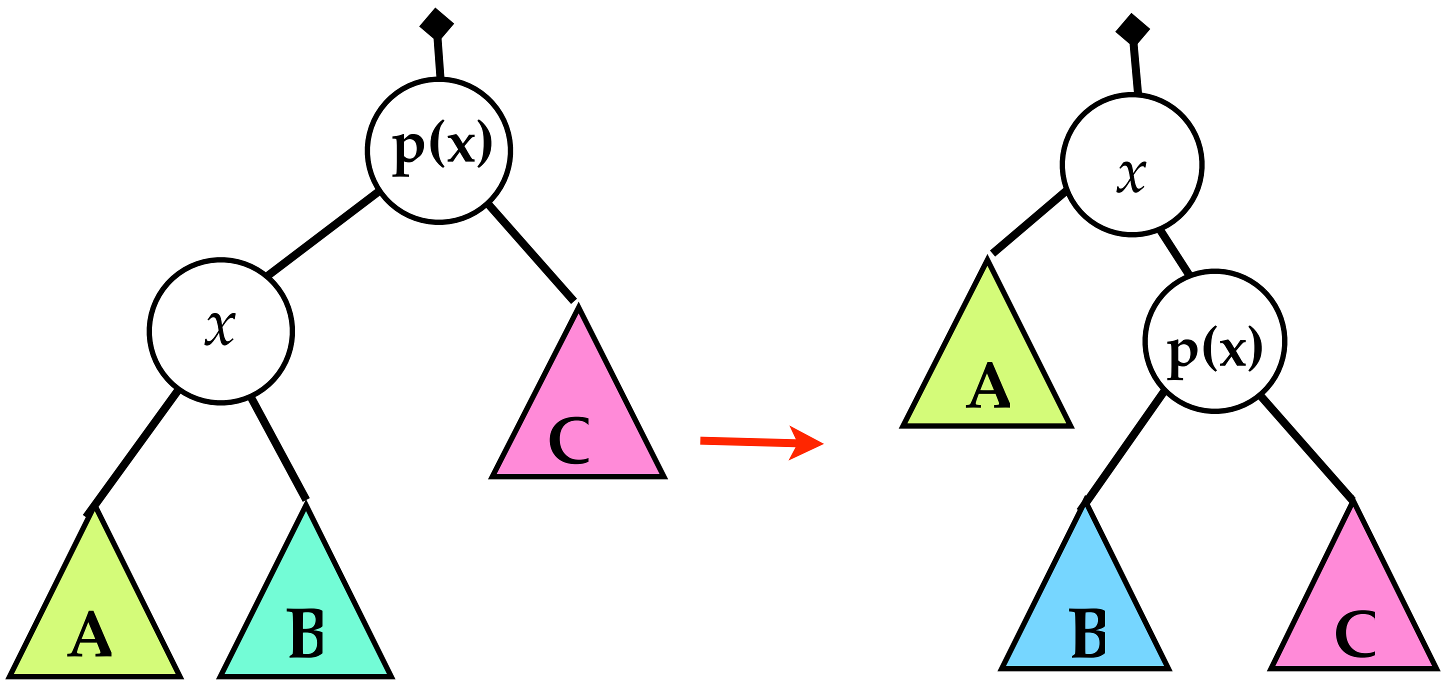
+1 pays for the rotation

$rank^1(x) = rank(p(x))$

Extra $ to keep the invariant is:

$\underbrace{\boxed{rank^1(x) + rank^1(p(x))}}_{\text{\$ needed for x and p(x)}} - (rank(x) + rank(p(x)))$

case 1: $3(rank^1(x) - rank(x)) + 1$



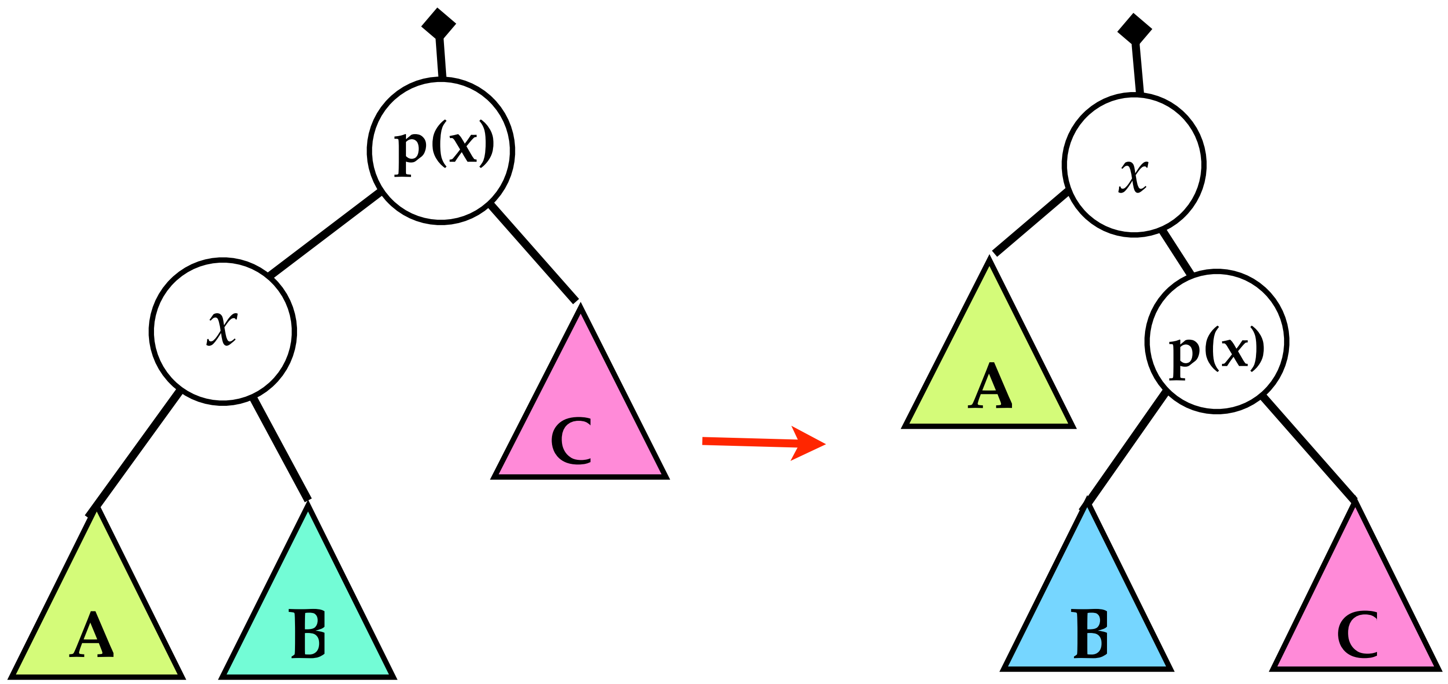
+1 pays for the rotation

$rank^1(x) = rank(p(x))$

Extra $ to keep the invariant is:

$\underbrace{\boxed{rank^1(x) + rank^1(p(x))}}_{\text{\$ needed for x and p(x)}} - \underbrace{\boxed{(rank(x) + rank(p(x))}}_{\text{\$ already on x and p(x)}}$

case 1: $3(\mathit{rank}^1(\text{x}) - \mathit{rank}(\text{x})) + 1$



+1 pays for the rotation

$\text{rank}^1(\text{x}) = \text{rank}(\text{p}(\text{x}))$

Extra \$ to keep the invariant is:

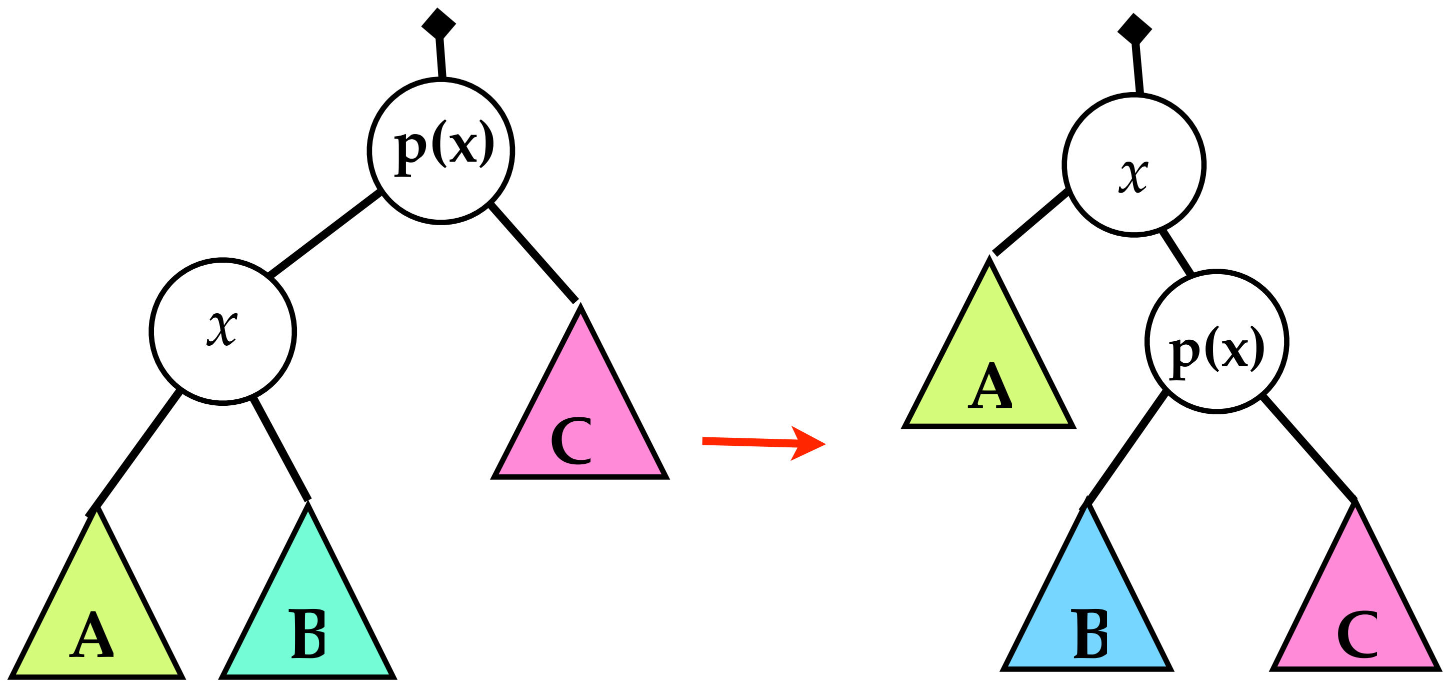
$$\boxed{\cancel{\text{rank}^1(\text{x})} + \text{rank}^1(\text{p}(\text{x}))} - \boxed{(\text{rank}(\text{x}) + \cancel{\text{rank}(\text{p}(\text{x}))})}$$

\$ needed for x and p(x)

\$ already on x and p(x)

r(x) after = r(p(x)) before.

case 1: $3(\mathit{rank}^1(x) - \mathit{rank}(x)) + 1$



+1 pays for the rotation

$\mathrm{rank}^1(x) = \mathrm{rank}(p(x))$

Extra $ to keep the invariant is:

$$\boxed{\cancel{\mathrm{rank}^1(x)} + \mathrm{rank}^1(p(x))} - \boxed{(\mathrm{rank}(x) + \cancel{\mathrm{rank}(p(x))})}$$

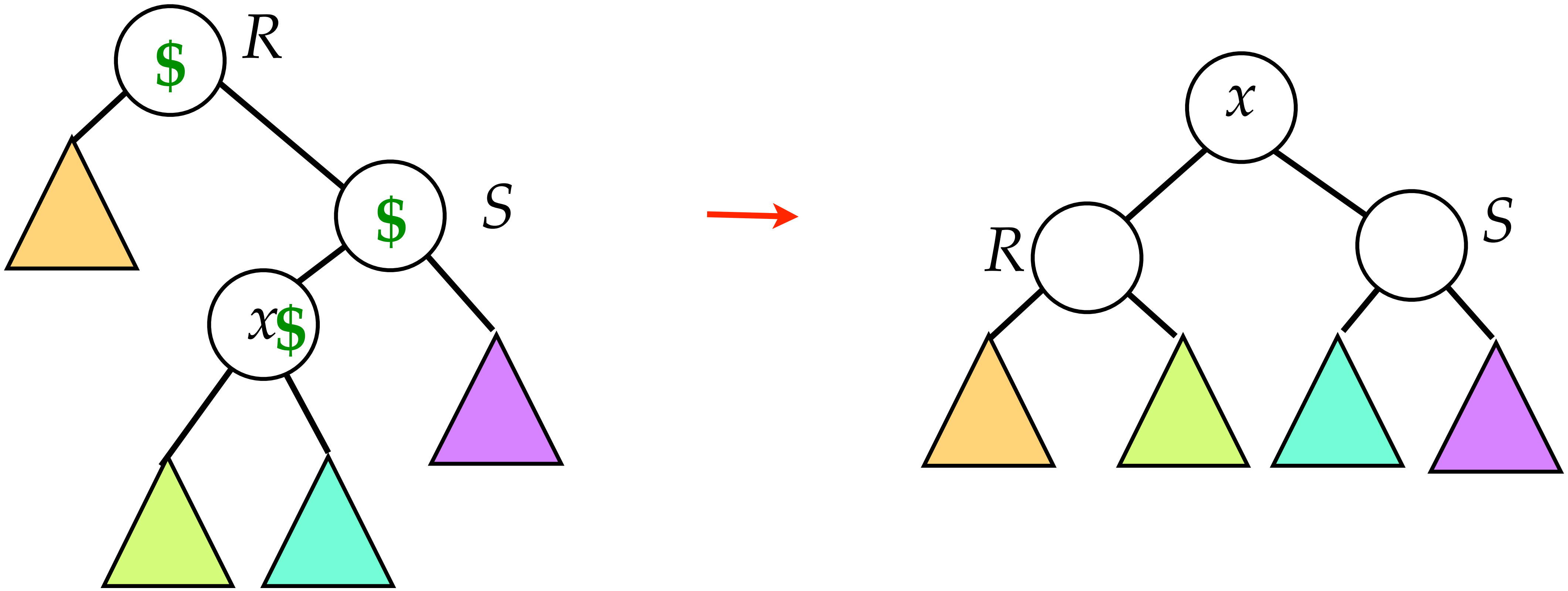
$ needed for x and p(x) $\qquad$ $ already on x and p(x)

r(x) after = r(p(x)) before.

$$= \mathrm{rank}^1(p(x)) - \mathrm{rank}(x)$$

$$\leq \mathrm{rank}^1(x) - \mathrm{rank}(x) \leq 3(\mathrm{rank}^1(x) - \mathrm{rank}(x))$$

case 2: $3(rank^1(x) - rank(x))$



$ *needed to add* $\leq$ $\text{rank}^1(R) - \text{rank}(x)$

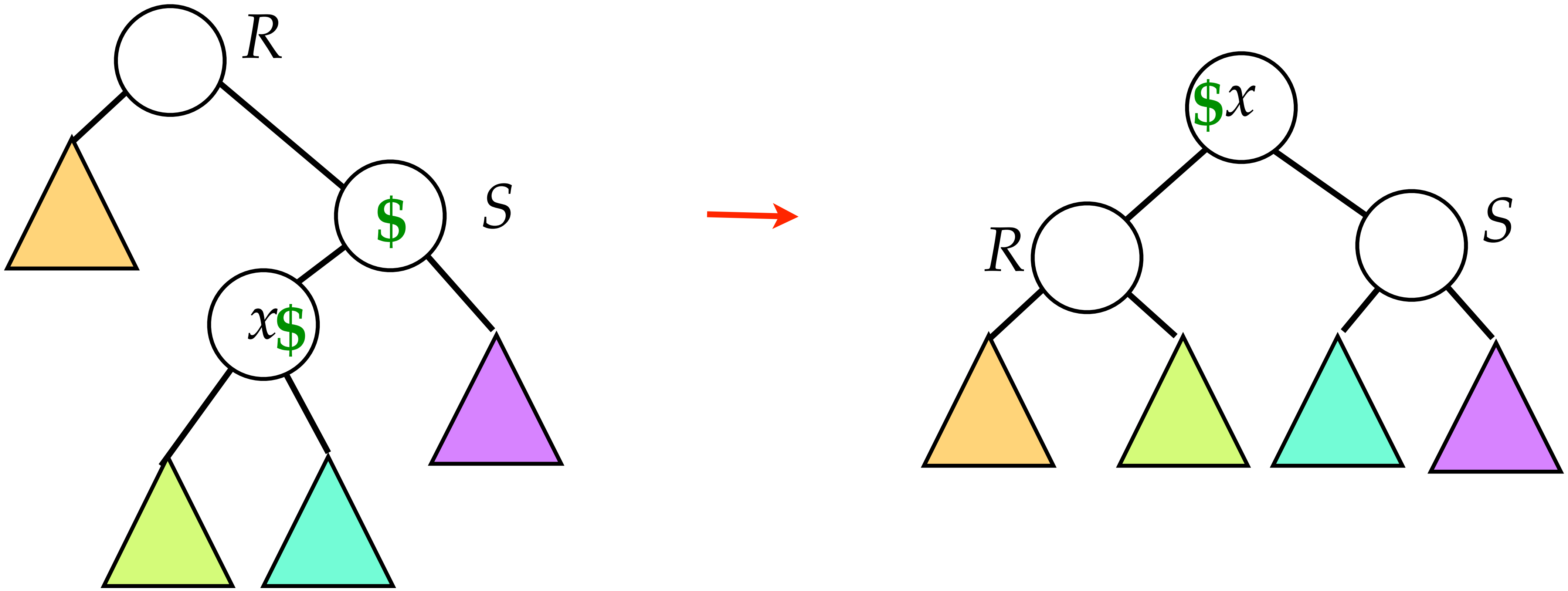
$\leq$ $\text{rank}^1(x) - \text{rank}(x)$

But how do we pay for the rotations?

If $\text{rank}^1(x) - \text{rank}(x) > 0$, then
we need to add $\leq r^1(x) - r(x)$ but we have $3(r^1(x) - r(x)) > 1$
budgeted, so we have at least $1 to pay for the rotations.

Otherwise see next slide.

case 2: $3(\mathit{rank}^1(x) - \mathit{rank}(x))$



$\$$ *needed to add* $\leq \; \mathrm{rank}^1(R) - \mathrm{rank}(x)$

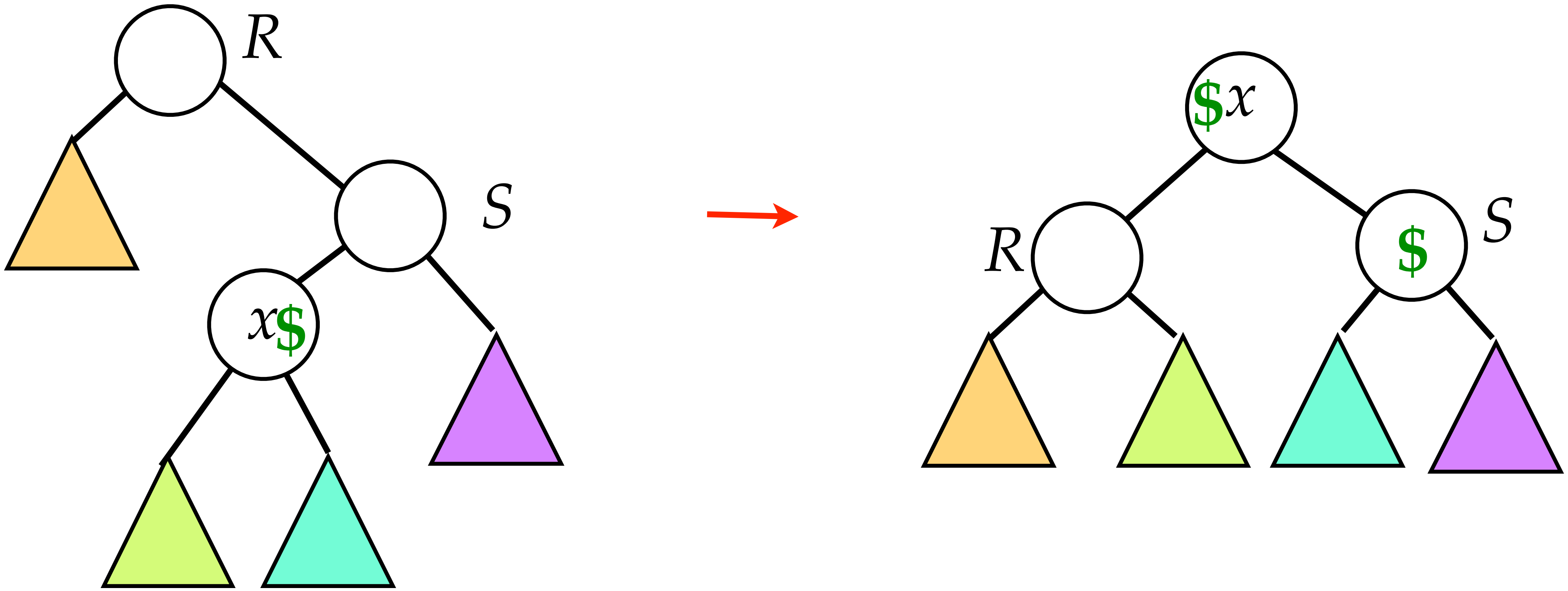
$\leq \; \mathrm{rank}^1(x) - \mathrm{rank}(x)$

But how do we pay for the rotations?

If $\mathrm{rank}^1(x) - \mathrm{rank}(x) > 0$, then
we need to add $\leq r^1(x) - r(x)$ but we have $3(r^1(x) - r(x)) > 1$
budgeted, so we have at least \$1 to pay for the rotations.

Otherwise see next slide.

case 2: $3(\mathit{rank}^1(x) - \mathit{rank}(x))$



$ needed to add $\leq$ $\text{rank}^1(R) - \text{rank}(x)$

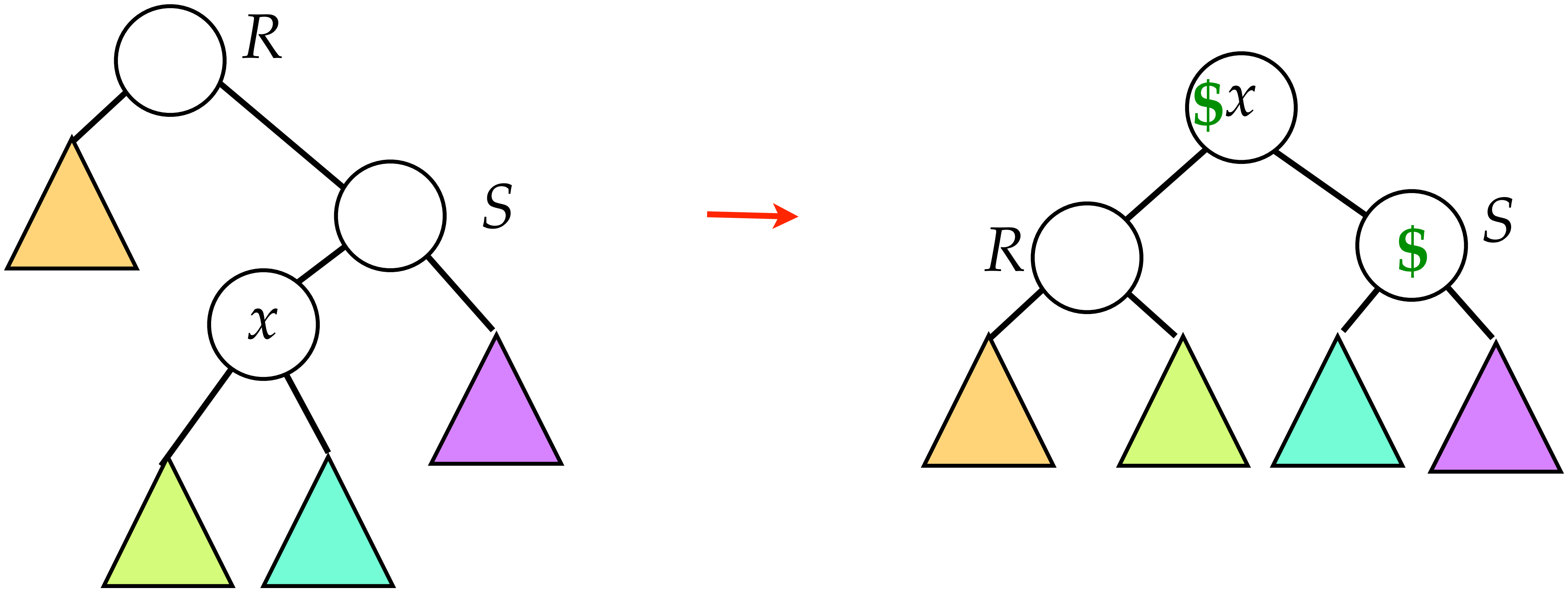
$\leq$ $\text{rank}^1(x) - \text{rank}(x)$

But how do we pay for the rotations?

If $\text{rank}^1(x) - \text{rank}(x) > 0$, then
we need to add $\leq r^1(x) - r(x)$ but we have $3(r^1(x) - r(x)) > 1$ budgeted, so we have at least $1 to pay for the rotations.

Otherwise see next slide.

case 2: $3(\mathit{rank}^1(x) - \mathit{rank}(x))$



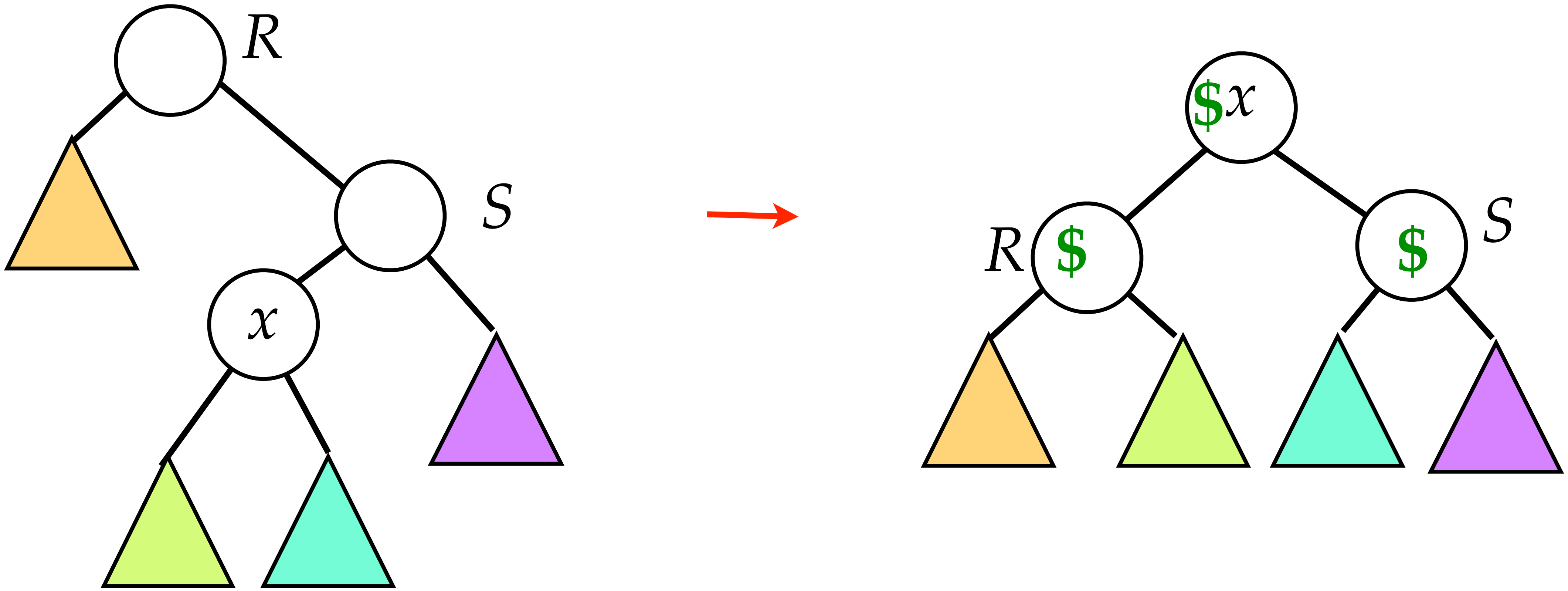
$$\textit{\$ needed to add} \leq \mathrm{rank}^1(R) - \mathrm{rank}(x)$$
$$\leq \mathrm{rank}^1(x) - \mathrm{rank}(x)$$

But how do we pay for the rotations?

If $\mathrm{rank}^1(x) - \mathrm{rank}(x) > 0$, then
we need to add $\leq r^1(x) - r(x)$ but we have $3(r^1(x) - r(x)) > 1$
budgeted, so we have at least \$1 to pay for the rotations.

Otherwise see next slide.

case 2: $3(rank^1(x) - rank(x))$



$\$$ *needed to add* $\leq \text{rank}^1(R) - \text{rank}(x)$

$\leq \text{rank}^1(x) - \text{rank}(x)$

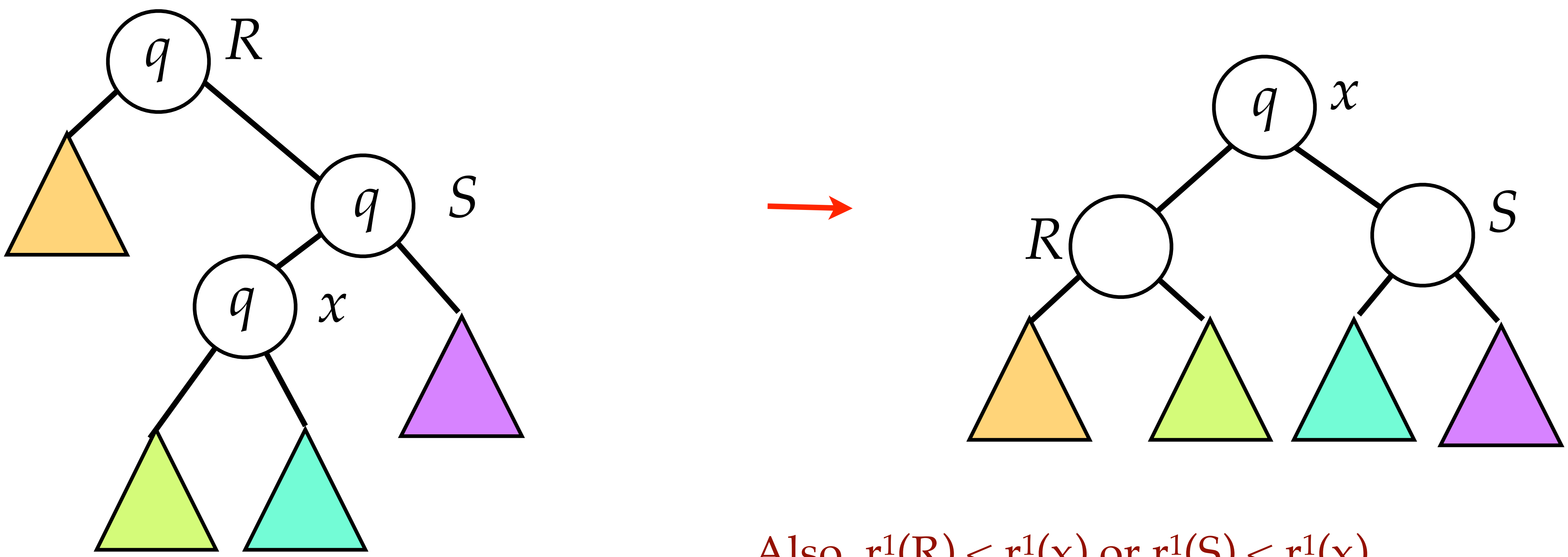
But how do we pay for the rotations?

If $\text{rank}^1(x) - \text{rank}(x) > 0$, then
we need to add $\leq r^1(x) - r(x)$ but we have $3(r^1(x) - r(x)) > 1$ budgeted, so we have at least \$1 to pay for the rotations.

Otherwise see next slide.

case 2: What if ($rank^1(x) - rank(x)$) is 0?

Since $r^1(x) = r(x)$, and $r^1(x) = r(R)$, we must have $r(x) = r(R) = r(S)$:



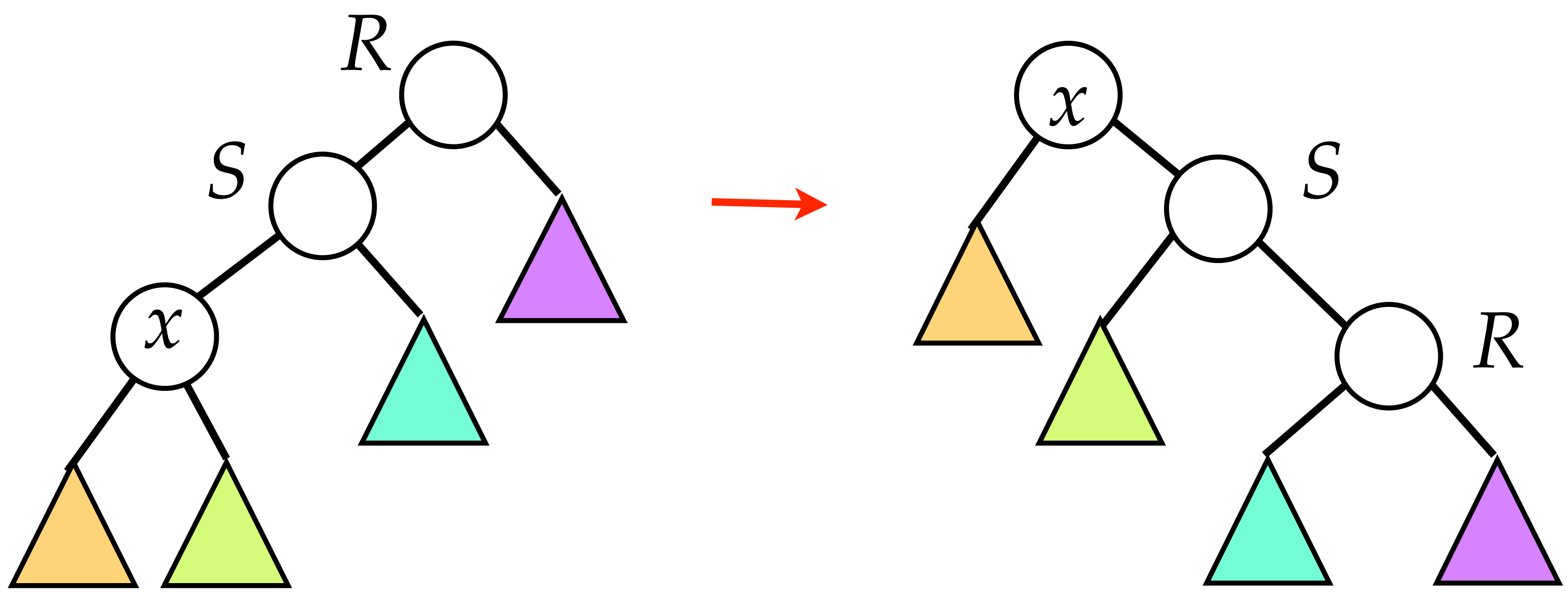
Also, $r^1(R) < r^1(x)$ or $r^1(S) < r^1(x)$

Suppose both $r(R) = r(S) = q$.

Then R would have at least 2^q nodes under it and S would have at least 2^q nodes under it. So x would have at least $2(2^q)$ nodes under it and x would then have rank $q+1$ instead of q.

Had $3q$ \$ on tree before, now need $< 3q$, so have 1 \$ left to pay for rotations.

case 3: $3(rank^1(x) - rank(x))$

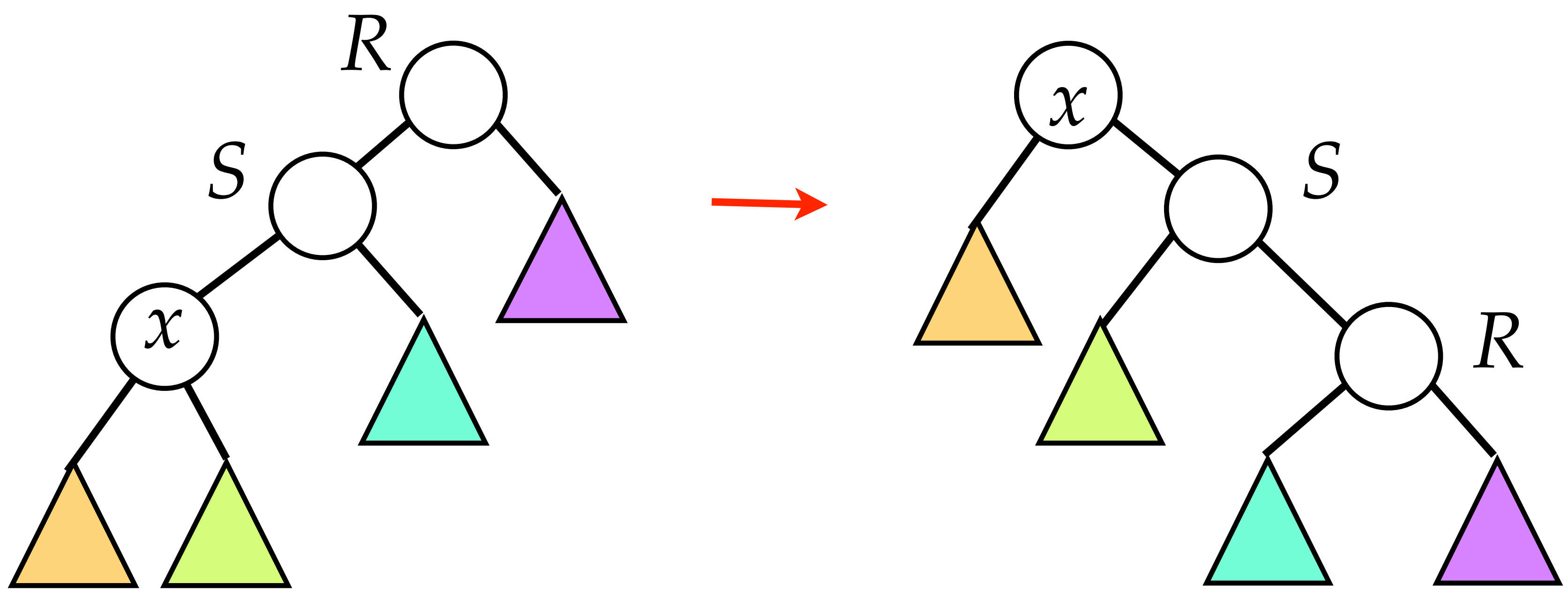


$ needed to add:

$$\boxed{r^1(x) + r^1(S) + r^1(R)} - \boxed{(r(x) + r(S) + r(R))}$$

$ needed for moved nodes $ already on moved nodes

case 3: $3(\mathit{rank}^1(x) - \mathit{rank}(x))$



$ needed to add:

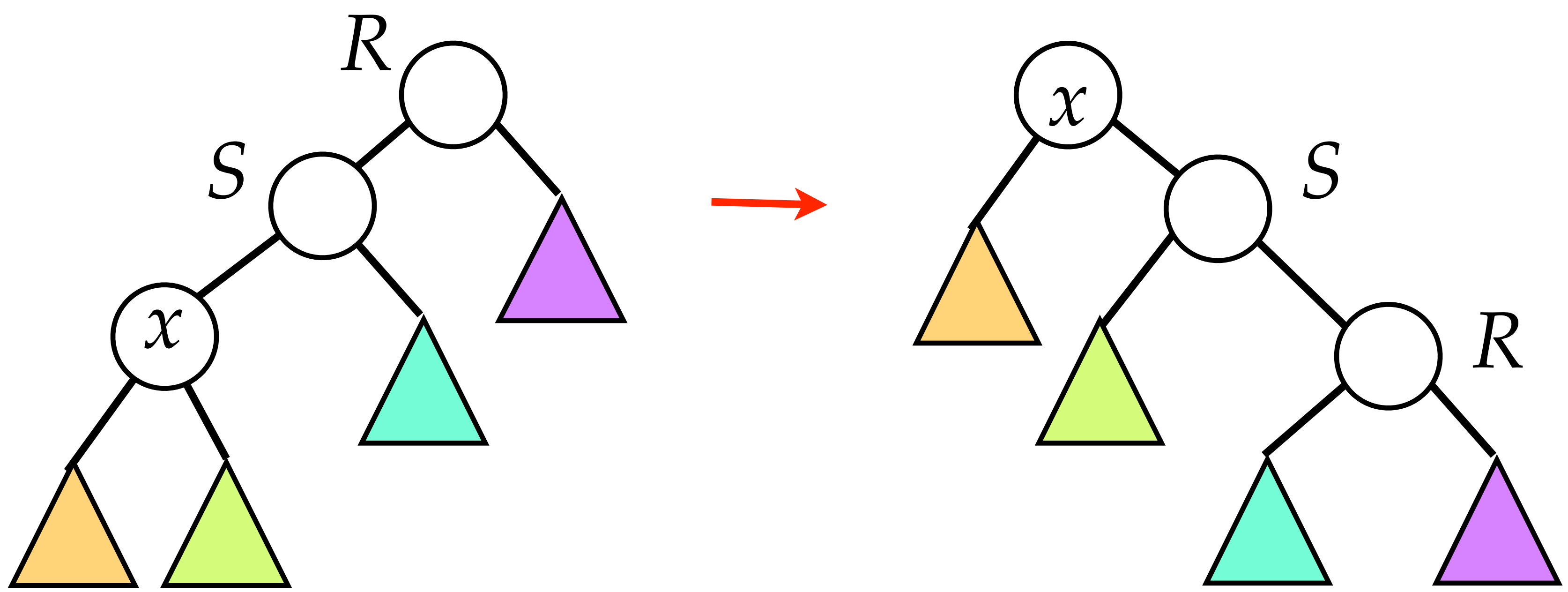
$$\boxed{r^1\cancel{(x)} + r^1(S) + r^1(R)} - \boxed{(r(x) + r(S) + r\cancel{(R)})}$$

$ needed for moved nodes $ already on moved nodes

$$= r^1(S) + r^1(R) - r(x) - r(S)$$

$r^1(x) = r(R)$

case 3: $3(rank^1(x) - rank(x))$



$ needed to add:

$\boxed{r^1\cancel{(x)} + r^1(S) + r^1(R)} - \boxed{(r(x) + r(S) + r\cancel{(R)})}$

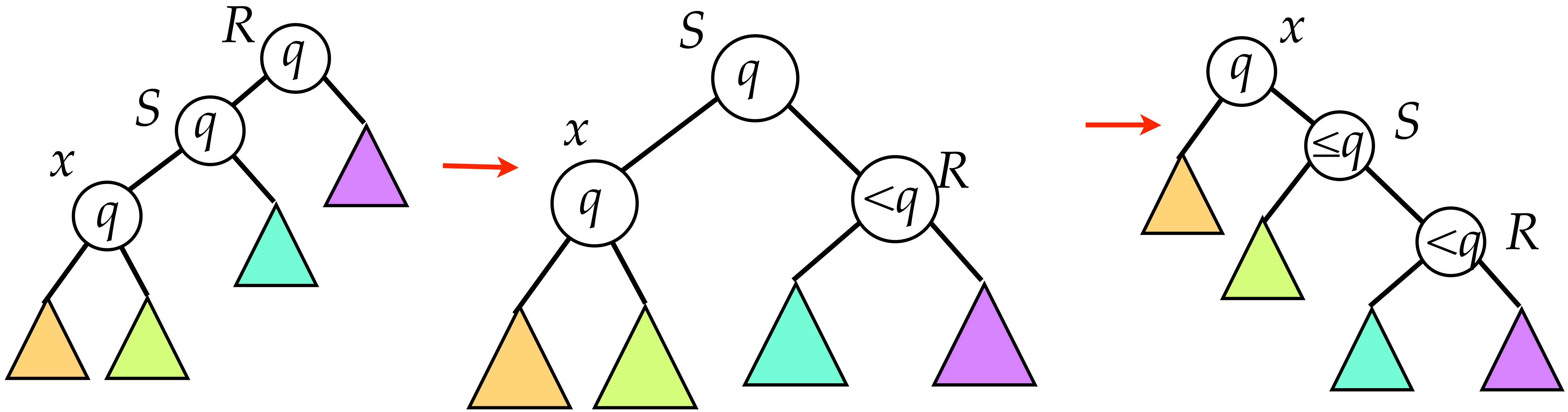
$ needed for moved nodes $ already on moved nodes

$= r^1(S) + r^1(R) - r(x) - r(S)$ $r^1(x) = r(R)$

$\leq r^1(x) + r^1(x) - r(x) - r(x)$ $r^1(R) \leq r^1(S) \leq r^1(x)$

$\leq 2(r^1(x) - r(x))$ $r(x) \leq r(S)$

case 3: What if $rank^1(x) - rank(x) = 0$?



Why must R have rank $< q$?

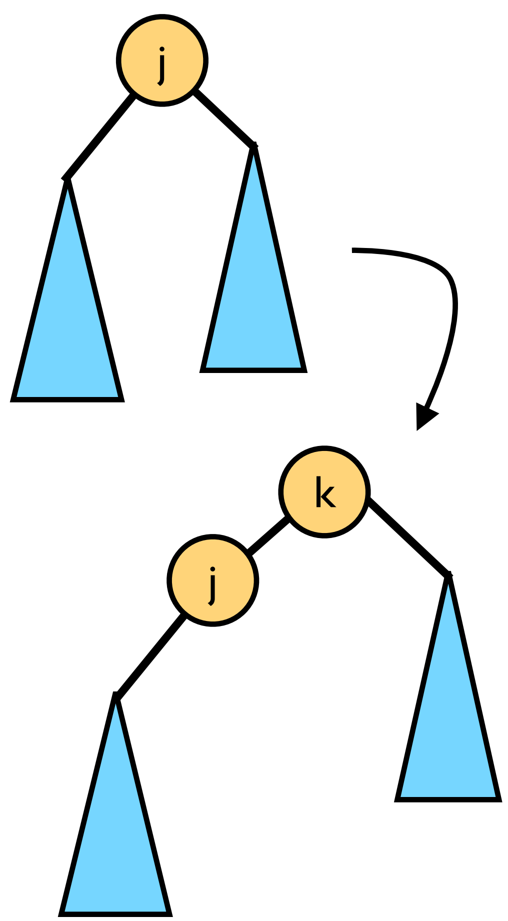
Suppose not. Then x has at least 2^q nodes under it, and R has at least 2^q nodes under it.

So S has at least $2(2^q) = 2^{q+1}$ nodes under it,
so it should have rank Q+1, but it has rank Q, which is a contradiction.

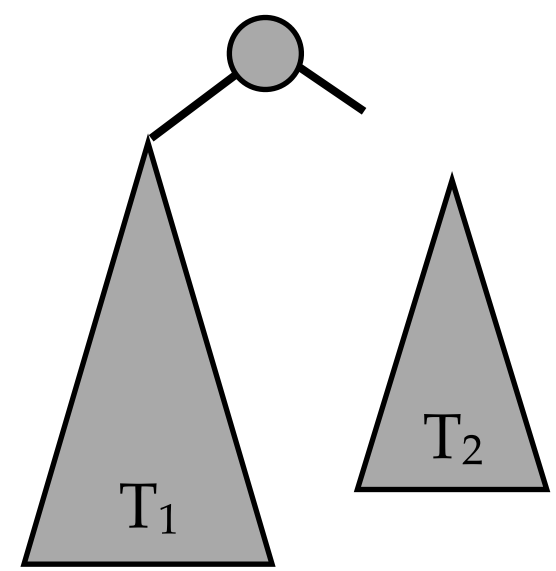
So before we had $3q$ \$ on the nodes, now we need only $\leq 2q + q - 1$

Additional Cost of Insert & Concat

- Cost of *insert* & *concat* more than the cost of a splay because may have to add $s to root to maintain invariant:



insert(T, k): k has *n* descendants, so need to put [log n] $ on k



concat(T_1, T_2): root gets at most *n* new descendants from T_2, so need to put [log n] dollars on root.