Step 3: swap \((30 + 35)\)

3 pointer changes:
- \(a\)'s parent (right ptr) \(\rightarrow a\)'s child \(20 \rightarrow 35\)
- \(a\)'s right child (left ptr) \(\rightarrow a\)
  \(35 \rightarrow 30\)
- \(a\)'s (right ptr) \(\rightarrow a\)'s successor \(30 \rightarrow 33\)

5) left/right switched for other side

Example 2: insert (4)

Swap \(a\) (10) + \(a\)'s child (5)

- \(a\)'s parent (left ptr) \(\rightarrow a\)'s child \(20 \rightarrow 5\)
- \(a\)'s left child (right ptr) \(\rightarrow a\)
  \(5 \rightarrow 10\)
- \(a\)'s new (left ptr) \(\rightarrow a\)'s successor \(10 \rightarrow \text{null}\)
AVL? ✓

insert (34)

AVL? No a? 20

Which type of insert? inner.
-> left subtree of a's right child.

double swap - swap a's child and grandchild (on side of imbalance)
swap a and a's new child

After 1st swap:
swap 35 + 30

Possibly skip?

3 ptr changes: <from original tree>
a's (right) => a's right child's left child  20 => 30
a's grandchild (right ptr) => a's child  30 => 35
a's child (left ptr) => it's (prev) predecessor  35 => 33
After 2nd swap
Swap $20 \leftrightarrow 30$

3 ptr. changes

- $a$'s parent's (right ptr) $\rightarrow a$'s child, root $\rightarrow 30$
- $a$'s child's (left ptr) $\rightarrow a$
- $a$'s (right ptr) $\rightarrow a$'s (prev) successor, $20 \rightarrow 25$

Another example

Type of insert? inner
(right subtree of $a$'s left child).
Double swap: (1) swap α's child and grandchild.

\[
\begin{array}{c}
14 \\
\alpha \rightarrow 12 \rightarrow 15 \\
\downarrow \downarrow \downarrow \\
8 \quad 13 \quad 16 \\
\downarrow \downarrow \\
7 \quad 9 \\
\downarrow \\
5
\end{array}
\]

(2) swap α + α's new child.

\[
\begin{array}{c}
14 \\
\downarrow \downarrow \\
8 \quad 15 \\
\downarrow \downarrow \\
7 \quad 12 \quad 16 \\
\downarrow \downarrow \\
5 \quad 9 \quad 13
\end{array}
\]
Suppose space wasn't an issue.

**Lazy Deletion** - As we did for arrays.

Remove (2)

Problem? Setting the value to -1 destroys order of tree. Instead leave element in tree, but mark as deleted.

**Binary Node**
- element
- left
- right
- duplicates
- boolean deleted

<table>
<thead>
<tr>
<th>Operation</th>
<th>AVL</th>
<th>D-LL</th>
<th>ArrayList</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>contains</td>
<td>$O(\log n)$ worst+avg</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>insert</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>lazy + non lazy remove</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>get</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
When to use AVL Trees?

(1) - to find min/max value

```
Object findMax(BinaryNode t)?

if (t != null) { //make sure tree is not empty
    while (t.right != null)
        t = t.right
    return t.element
}
```

(2) - to find a range of values

```
all values here > a
```

```
all values here: > b and < a
```

(3) - to get values in sorted order
inorder traversal