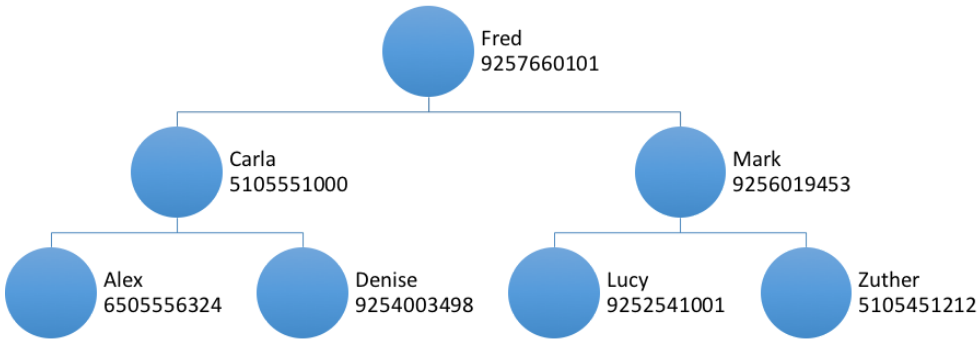
The Pop Operation, Reading

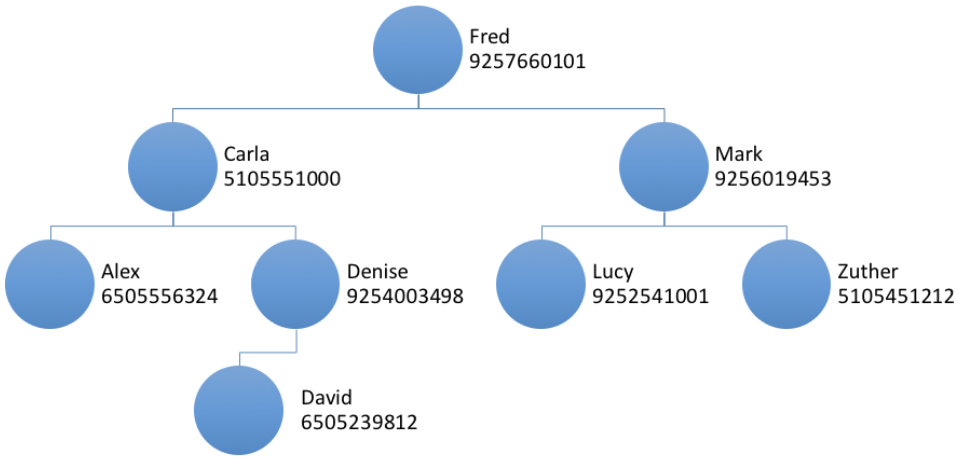
Removing and deallocating a node is more involved that it may appear at first. It's not just deallocating that node, because you'd lose its children too. And it's not linking over to a child (as you would when removing a node from a linked list) because there's the sibling node (the other child) who may not be able to be a child of the newly promoted parent because it may already have two children. Think about removing Fred, and how that would go:



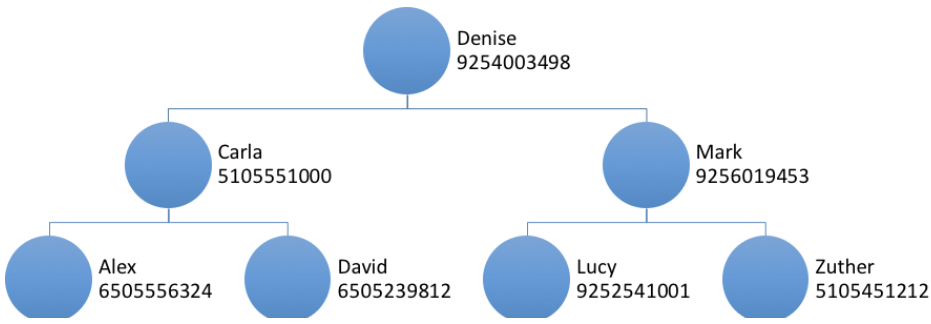
No, deallocating and relinking does not look too attractive right now. Recall that we want to maintain left-to-right order, ignoring the vertical offsets of the levels of the tree. That means the easiest thing to do may be to ***copy Denise or Lucy to the top spot***, and remove and deallocate *their* node.

To *find* Denise, or a right-descendent of Denise who is greater than Denise but still less than Fred and all of Fred's right-descendents, here's the ***dance step***: left, then right-right-right until there are no more rights. *That's* the node to copy and deallocate. To find Lucy, it's right, left-left-left until there are no more lefts.

Of course, we'll have to decide who gets preference -- the left child or the right in case there are both. We also have to have a following "prev" pointer, so when "p" finds Denise (or Lucy), it will be possible to cut them loose from Carla (or Mark). Also, if the deallocated node has a child, that child would need to get linked to "prev" where the deallocate node used to be. Got all that? It's like this:



Removing Fred, it's left (Carla), right (Denise), stop, landing us on Denise. Copy Denise to Fred's spot. Then link Carla's right (that used to be linked to the old Denise) to David instead. That's a relink, not a copy because David could have children, and copies and promotions will be way hard to do. This should be the result, with the relinked David moved up:



Finding The Key To Be Deleted: The deleteKey Setter

Start with the same search loop we've been using, but use the one with "prev" because if the key's node has no children, it's simply cut from its parent, "prev".

```
Node* p = rootNode;
Node* prev = 0;
while (p)
{
  if (p->key == key) break; // found it!
  prev = p; // save p's old position before advancing it
  if (p->key < key) p = p->right; else p = p->left;
}
```

Here are the possible outcomes:

1. "p" is zero: no match found -- nothing to do
2. p->left, p->right, and prev are all zero -- **deallocate the root** and set root to zero
3. p->left and p->right are both zero -- **deallocate p**, set prev's left or right to zero
4. p->left is zero, **seek a right descendent** to promote to "p"
5. **seek a left descendent** to promote to "p"

2. Deallocate The Root -- *there is no "1."*

In this case, the only node in the tree is the root, and it's key matches.

```
  delete p;
  rootNode = 0;
  --siz;
```

3. Deallocate p, Set prev's Left Or Right To Zero

In this case, the deleted key's node has no descendants. It's simply deallocated, but we have to figure out if it was prev's *left* or *right* child, because its corresponding pointer needs to be zeroed out.

```
  delete p;
  if (prev->left == p) prev->left = 0;
  else prev->right = 0;
  --siz;
```

Don't worry about deleting "p" before the **==p** in the next statement. All that **delete p;** does is deallocate the node pointed to by "p". "p" still has its same value after deallocation, it's just that it's now a "wild" pointer value.

4. Seek A Right Descendent To Promote To "p"

We've chosen to favor a left descendent for "promotion" (not that it matters...) so we only promote from the right side if there is no left. The dance step is right-left-left-left until there are no more lefts. We don't need the old "prev" anymore, so we *re-purpose* it. But we do need to remember the old "p" so we have a place to promote into. We'll copy the old "p" to "pSave" and re-purpose "p" too.

```
  Node* pSave = p; // promote to this node
```

```
prev = p;
p = p->right; // the "right" step
while (p->left) // left-left-left
{
  prev = p; // follow p around...
  p = p->left;
}
```

When this loop ends:

- pSave points to the node to receive the promoted node's contents
- p points to the node whose contents are to be promoted, and is to be deallocated
- prev points to the parent of p -- set prev's left to p's right *even if it's zero*

```
pSave->key = p->key;
pSave->value = p->value;
if (prev->left == p) prev->left = p->right;
else prev->right = p->right // in case there were no left-left-left steps
delete p;
--siz;
```

5. Seek A Left Descendent To Promote To "p"

This is just a mirror image of the previous:

```
Node* pSave = p; // promote to this node
prev = p;
p = p->left; // the "left" step
while (p->right) // right-right-right
{
  prev = p; // follow p around...
  p = p->right;
}

pSave->key = p->key;
pSave->value = p->value;
if (prev->right == p) prev->right = p->left;
else prev->left = p->left // in case there were no right-right-right steps
delete p;
--siz;
```