Lecture #13

- Binary Trees, Cont.
  - Binary Search Tree *Node Deletion*
  - Uses for Binary Search Trees
  - Huffman Encoding
  - Balanced Trees

Deleting a Node from a Binary Search Tree

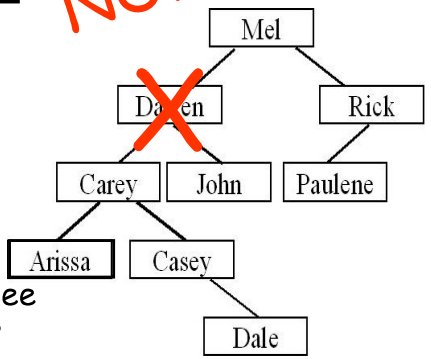
By simply moving an arbitrary node into Darren's slot, we violate our Binary Search Tree ordering requirement!

Carey is NOT less than Arissa!

Next we'll see how to do this properly….

It's not as easy as you might think!

NO!



Now how do I re-link the nodes back together?

Can I just move Arissa into Darren's old slot?

Hmm.. It seems OK, but is our tree still a valid binary search tree?

Deleting a Node from a Binary Search Tree

Here's a high-level algorithm to delete a node from a Binary Search Tree:

Given a value V to delete from the tree:

1. Find the value V in the tree, with a slightly-modified BST search.
   - Use two pointers: a cur pointer & a parent pointer
2. If the node was found, delete it from the tree, making sure to preserve its ordering!
   - There are three cases, so be careful!

BST Deletion: Step #1

This algorithm is very similar to our traditional BST searching algorithm… Except it also has a parent pointer.

When we're done with our loop below, we want the parent pointer to point to the node just above the target node we want to delete.

Every time we move down left or right, we advance the parent pointer as well!

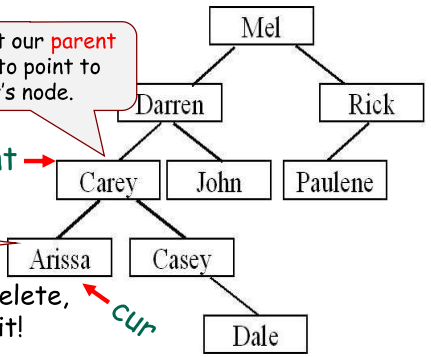
Step 1: Searching for value V

```
1. parent = NULL
2. cur = root
3. While (cur != NULL)
   A. If (V == cur->value) then we're done.
   B. If (V < cur->value)
        parent = cur;
        cur = cur->left;
   C. Else if (V > cur->value)
        parent = cur;
        cur = cur->right;
```

We'd want our parent pointer to point to Carey's node.

parent

So if we were deleting Arissa…



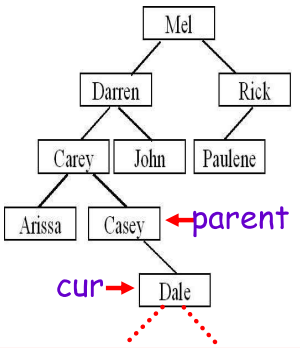
cur

Now cur points at the node we want to delete, and parent points to the node above it!

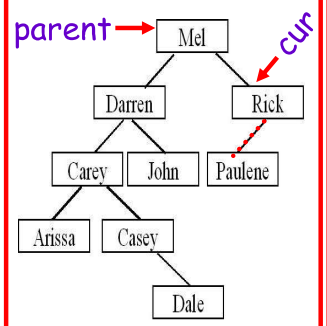
BST Deletion: Step #2

Once we've found our target node, we have to delete it.
There are 3 cases.

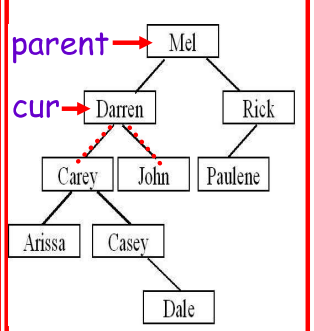
Case 1:
Our node is a leaf.



Case 2:
Our node has one child



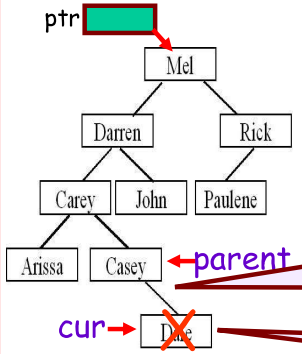
Case 3:
Our node has two children.



Step #2, Case #1 – Our Target Node is a Leaf

Let's look at case #1 – it has two sub-cases!

Case 1:
Our node is a leaf.



Case 1, Sub-case #1:
The target node is NOT the root node

1. Unlink the parent node from the target node (cur) by setting the parent's appropriate link to NULL.
2. Then delete the target (cur) node.

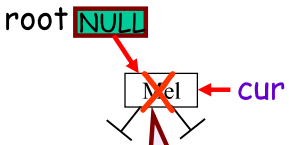
In this case, our target node (cur) is our parent node's right child…So we'll set parent->right to NULL to unlink the parent and cur.

Our target node (cur) that we want to delete is NOT the root node!

Step #2, Case #1 – Our Target Node is a Leaf

Let's look at case #1 – it has two sub-cases!

Case 1:
Our node is a leaf.



Our target node (cur) that we want to delete is the root node!

Case 1, Sub-case #1:
The target node is NOT the root node

1. Unlink the parent node from the target node (cur) by setting the parent's appropriate link to NULL.
2. Then delete the target (cur) node.

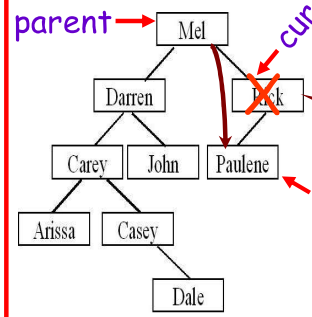
Case 1, Sub-case #2:
The target node is the root node

1. Set the root pointer to NULL.
2. Then delete the target (cur) node.

Step #2, Case #2 – Our Target Node has One Child

Let's look at case #2 now… It also has two sub-cases!

Case 2:
Our node has one child



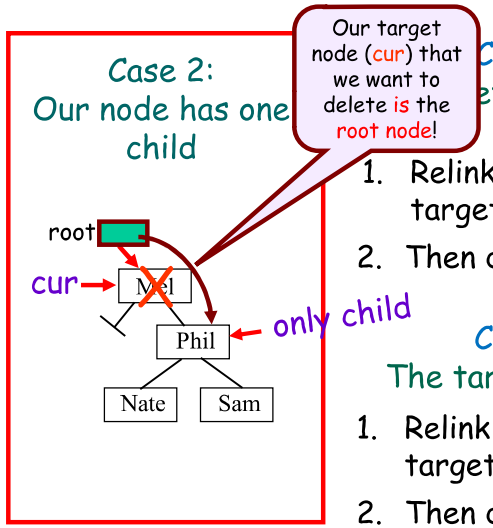
Case 1, Sub-case #1:
The target node is NOT the root node

1. Relink the parent node to the target (cur) node's only child.
2. Then delete the target (cur) node.

Our target node (cur) that we want to delete is NOT the root node!

Step #2, Case #2 – Our Target Node has One Child

Let's look at case #2 now… It also has two sub-cases!



Case 1, Sub-case #1:
The target node is NOT the root node

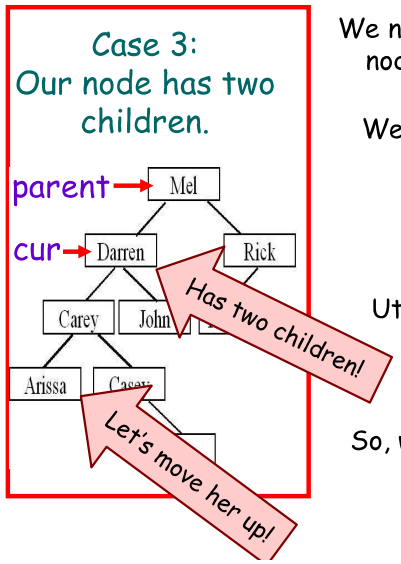
1. Relink the parent node to the target (cur) node's only child.
2. Then delete the target (cur) node.

Case 1, Sub-case #2:
The target node is the root node

1. Relink the root pointer to the target (cur) node's only child.
2. Then delete the target (cur) node.

Step #2, Case #3 – Our Target Node has Two Children

Let's look at case #3 now. The hard one!



We need to find a replacement for our target node that still leaves the BST consistent.

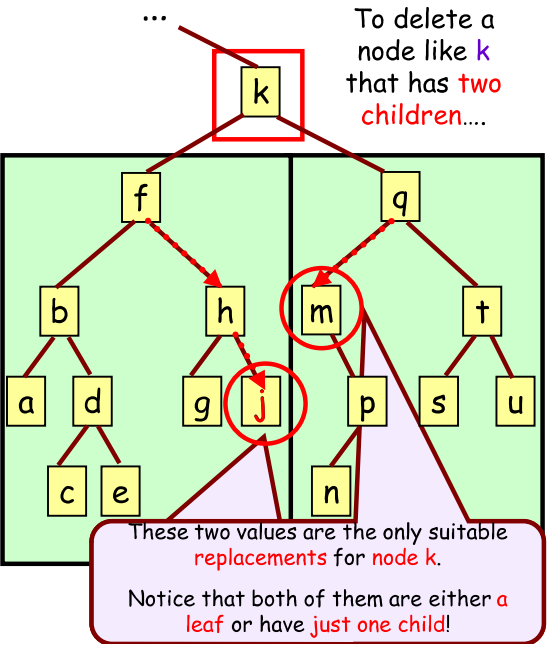
We can't just pick some arbitrary node and move it up into the vacated slot!

For instance, what if we tried replacing Darren with Arissa?

Utoh! If we replace Darren with Arissa, our BST is no longer consistent!

So, when deleting a node with two children, we have to be very careful!

Step #2, Case #3 – Our Target Node has Two Children



To delete a node like k that has two children….

We don't actually delete the node itself!

Instead, we replace its value with one from another node!

How? We want to replace k with either:

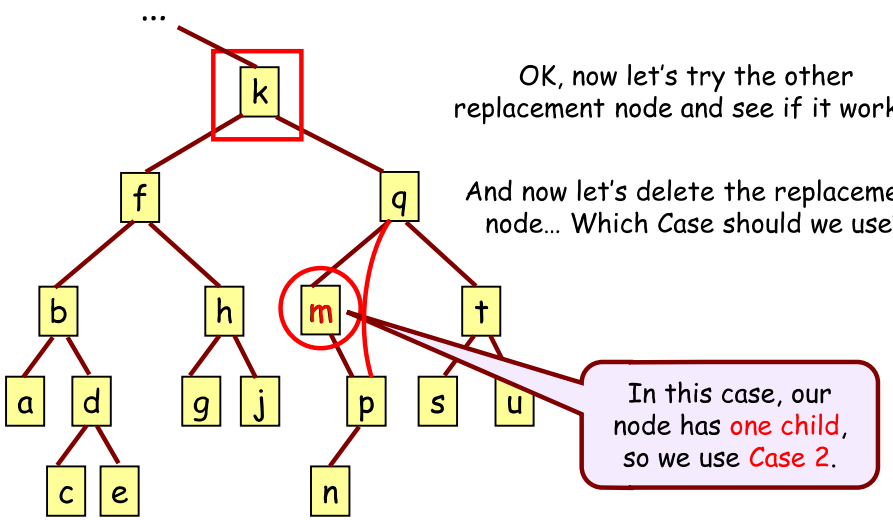
1. K's left subtree's largest-valued child

Or

2. K's right subtree's smallest-valued child

So we pick one, copy its value up, then delete that node!

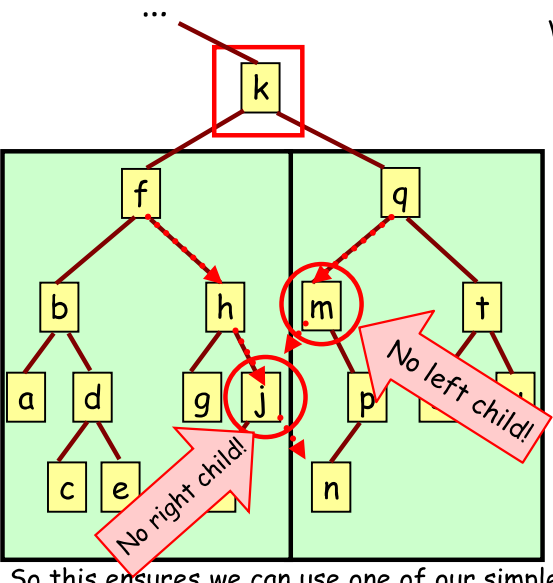
Step #2, Case #3 – Our Target Node has Two Children



OK, now let's try the other replacement node and see if it works!

And now let's delete the replacement node… Which Case should we use?

Step #2, Case #3 – Our Target Node has Two Children



So this ensures we can use one of our simpler deletion algorithms for the replacement!

Why is it guaranteed that our two replacement nodes have either zero or one child?

Well, we found the left subtree's maximum value by going all the way to the right…

So by definition, it can't have a right child!

Either it has a left child or no children at all…

The same holds true for the smallest value in our right subtree!

By definition, it can't have a left child!

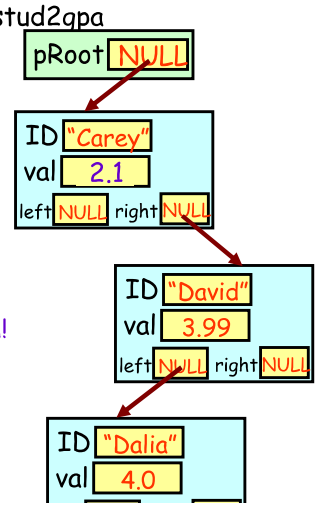
Where are Binary Search Trees Used?

Remember the STL map?

```
#include <map>
using namespace std;

main()
{
  map<string,float>  stud2gpa;

  stud2gpa["Carey"] = 3.62; // BST insert!
  stud2gpa["David"] = 3.99;
  stud2gpa["Dalia"] = 4.0;
  stud2gpa["Carey"] = 2.1;
  cout << stud2gpa["David"]; // BST search!
}
```



It uses a type of binary search tree to store the items!

Where are Binary Search Trees Used?

The STL set also uses a type of BSTs!

```
#include <set>
using namespace std;

main()
{
  set<int>      a;  // construct BST
  a.insert(2);      // insert into BST
  a.insert(3);
  a.insert(4);
  a.insert(2);

  int n;
  n = a.size();
  a.erase(2);       // delete from BST

}
```

The STL set and map use binary search trees (a special balanced kind) to enable fast searching.

Other STL containers like multiset and multimap also use binary search trees.

These containers can have duplicate mappings. (Unlike set and map)