Lecture #13

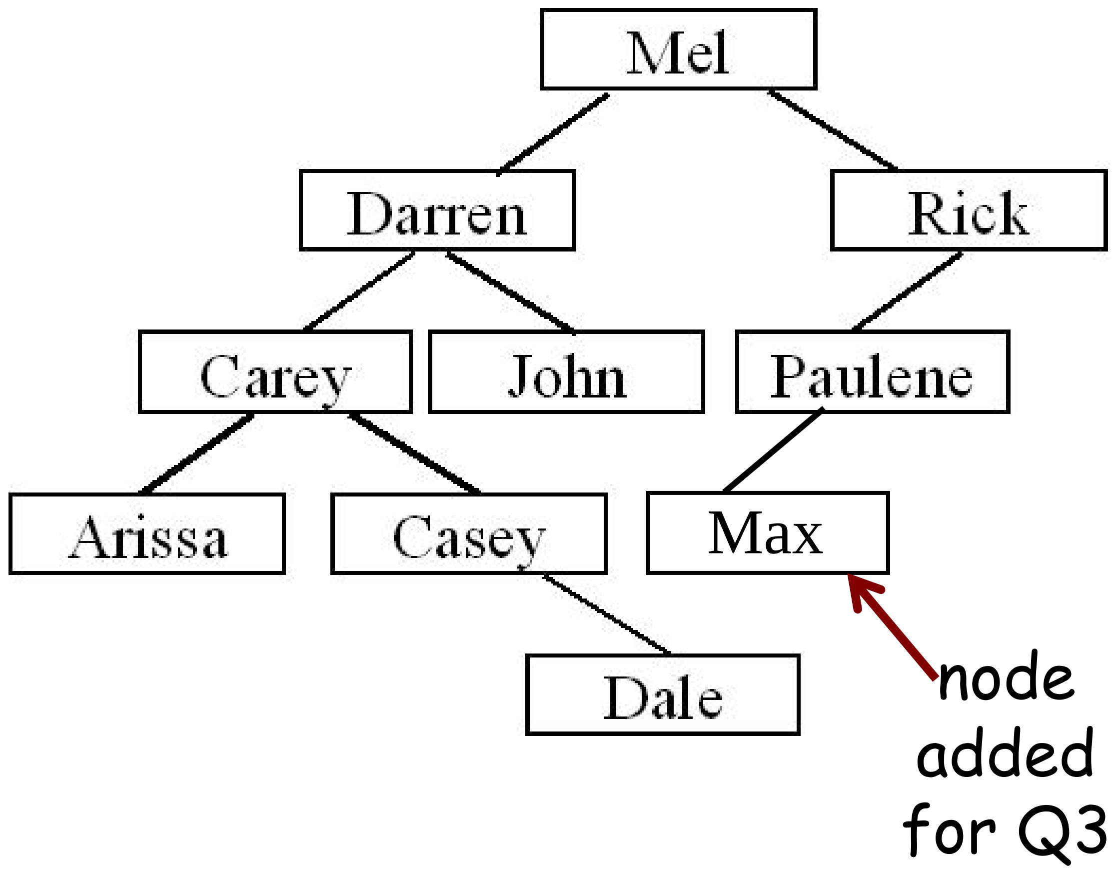
- Binary Tree Review
- Binary Search Tree *Node Deletion*
- Uses for Binary Search Trees
- Huffman Encoding
- Balanced Trees

Bina

```
void PostOrder(Node *cur)
{
   if (cur == NULL)   return;

   PostOrder(cur->left);      // Process nodes in left sub-tree.
   PostOrder(cur-> right);  // Process nodes in right sub-tree.
   cout << cur->value;          // Process the current node.
}
```

Question #1: What's the post-order traversal for the following tree?



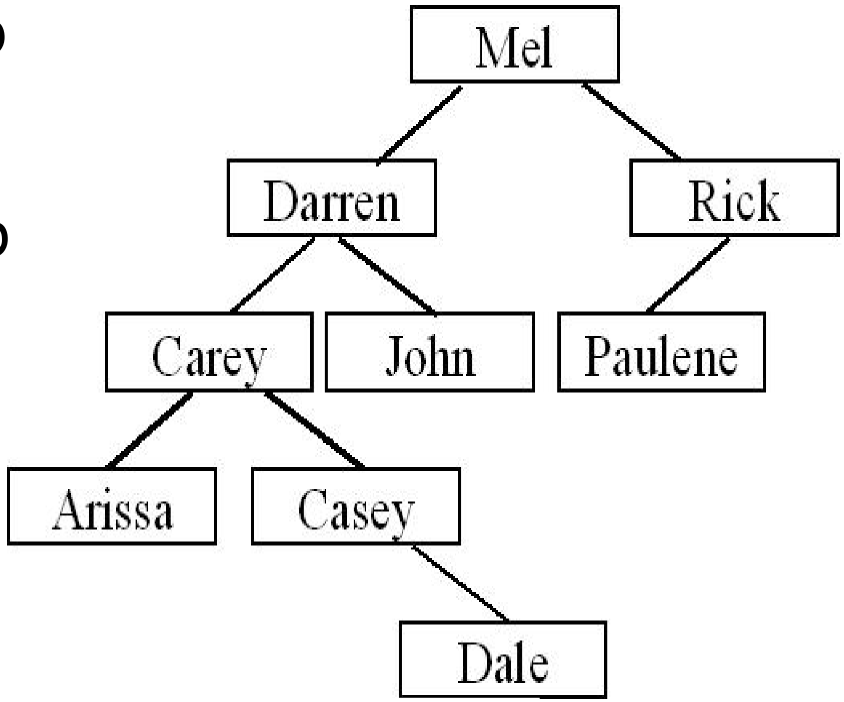
Question #2: Is the above tree a valid binary search tree?

Question #3: How about now?

Binary Search Tree Insertion Review

Question #1: How would you go about inserting "Cathy"

Question #2: How would you go about inserting "Priyank".



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!

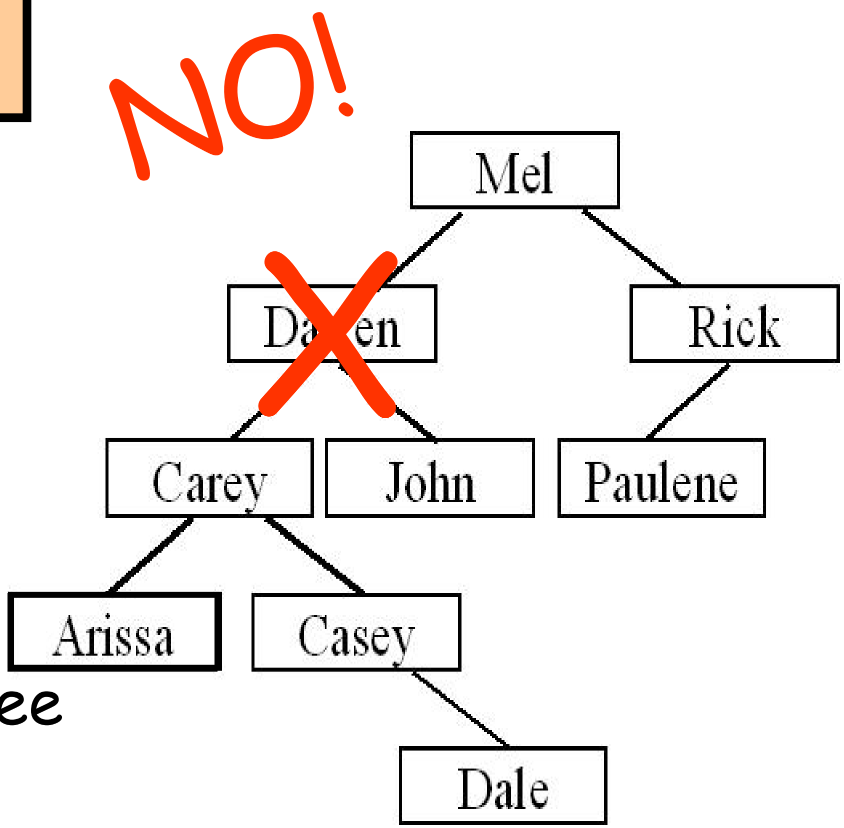
Carey is NOT less than Arissa!

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

It's not as easy as you might think!

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.

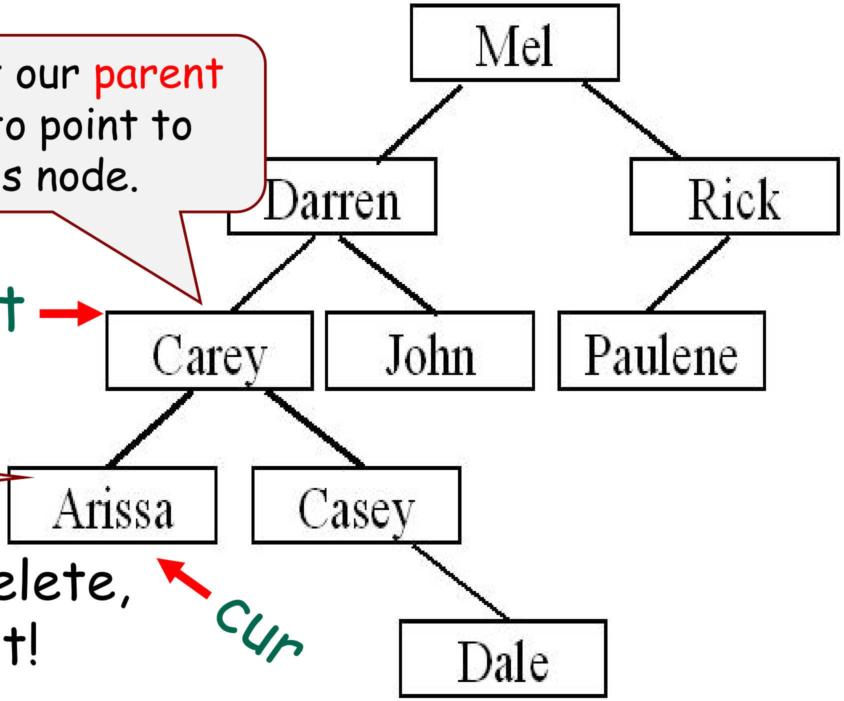
Step 1: Searching for value V

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;

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

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

parent



So if we were deleting Arissa…

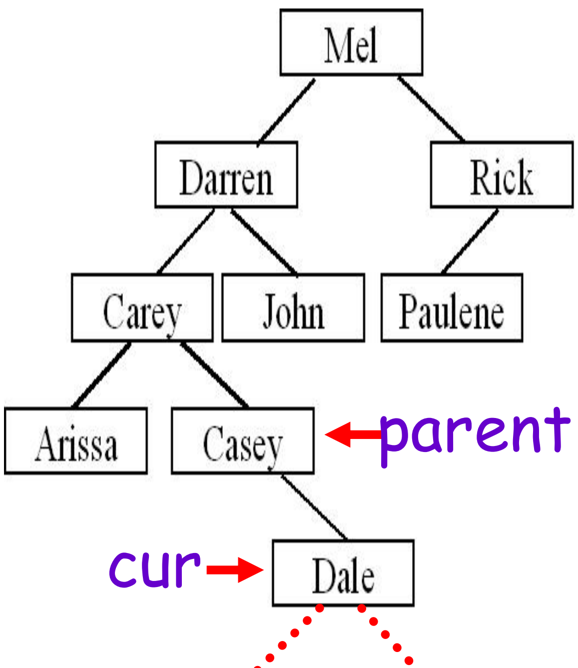
cur

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

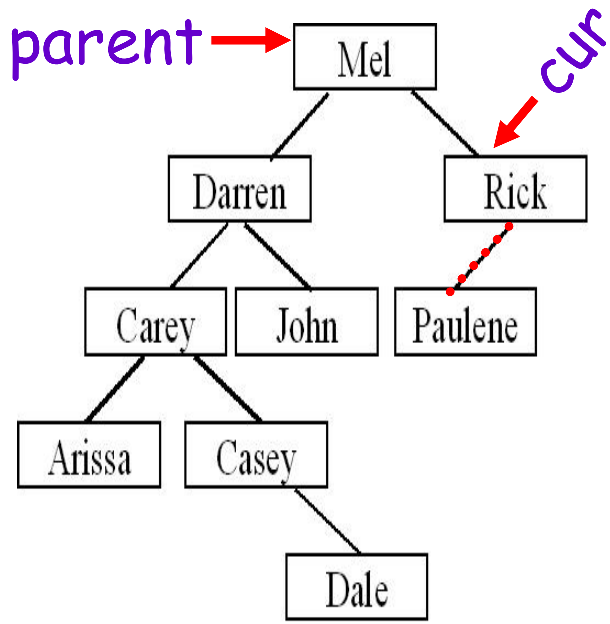
BST Deletion: Step #2

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

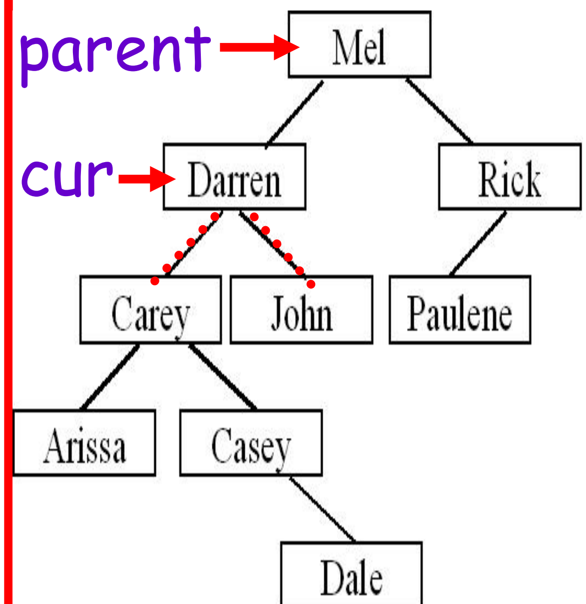
Case 1:
Our node is a leaf.



Case 2:
Our node has one child



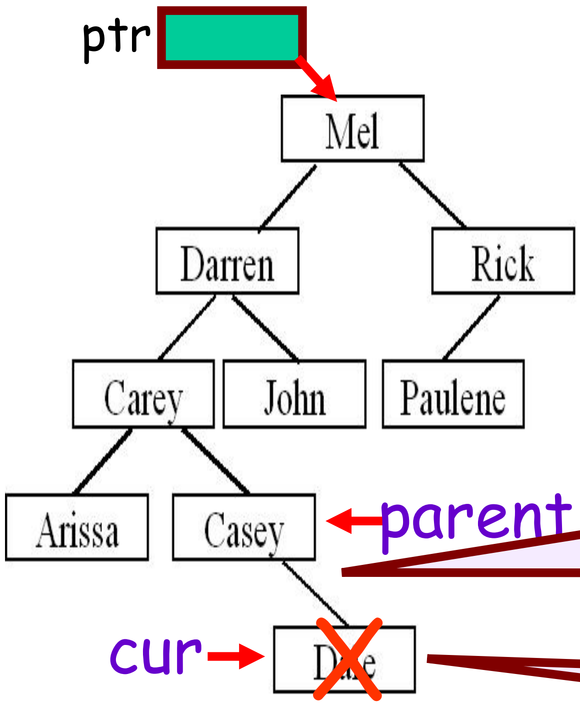
Case 3:
Our node has two children.



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.

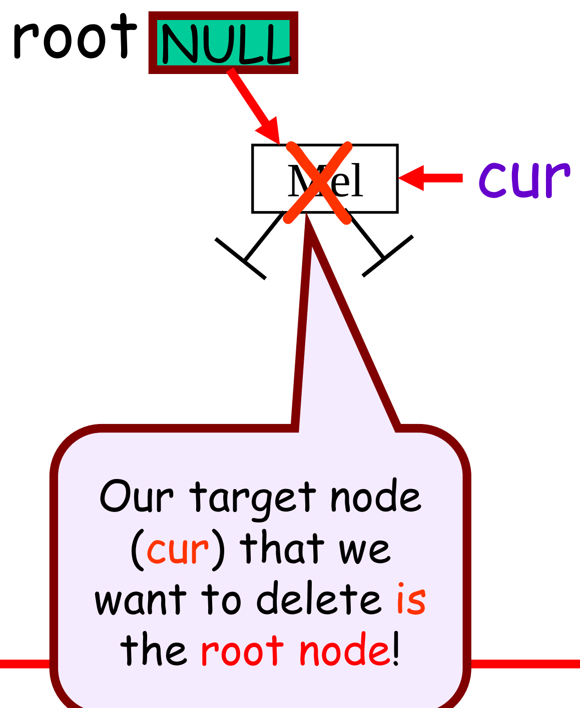
In this case, our target node (cur) is our parent node's right child…So we'll set parent->right to NULL to unlink the parent and cur.

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.



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.

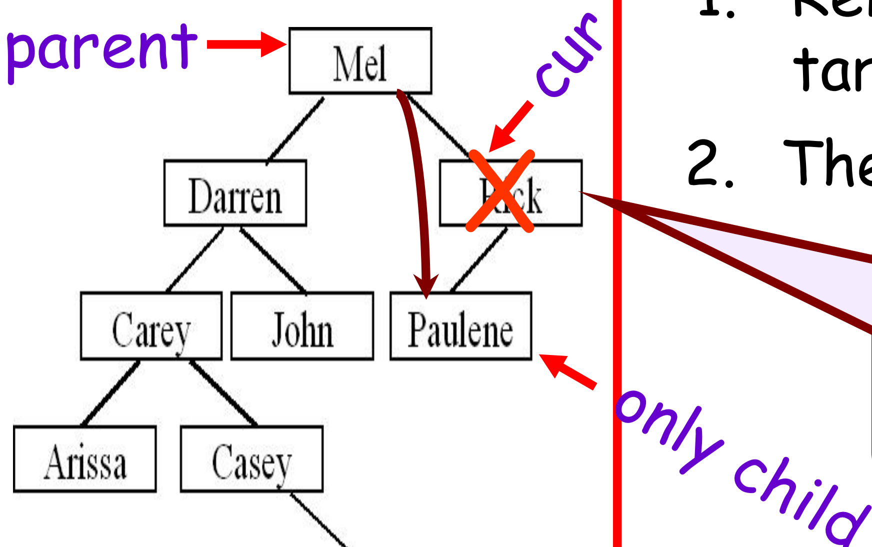
Case 1, Sub-case #2:
The target node is the root node

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.

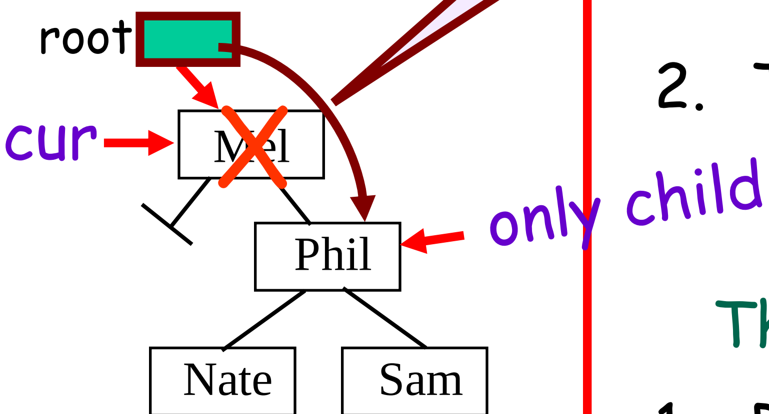
Our target node (cur) that we want to delete is NOT the root node!

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

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

Case 2:
Our node has one child

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



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

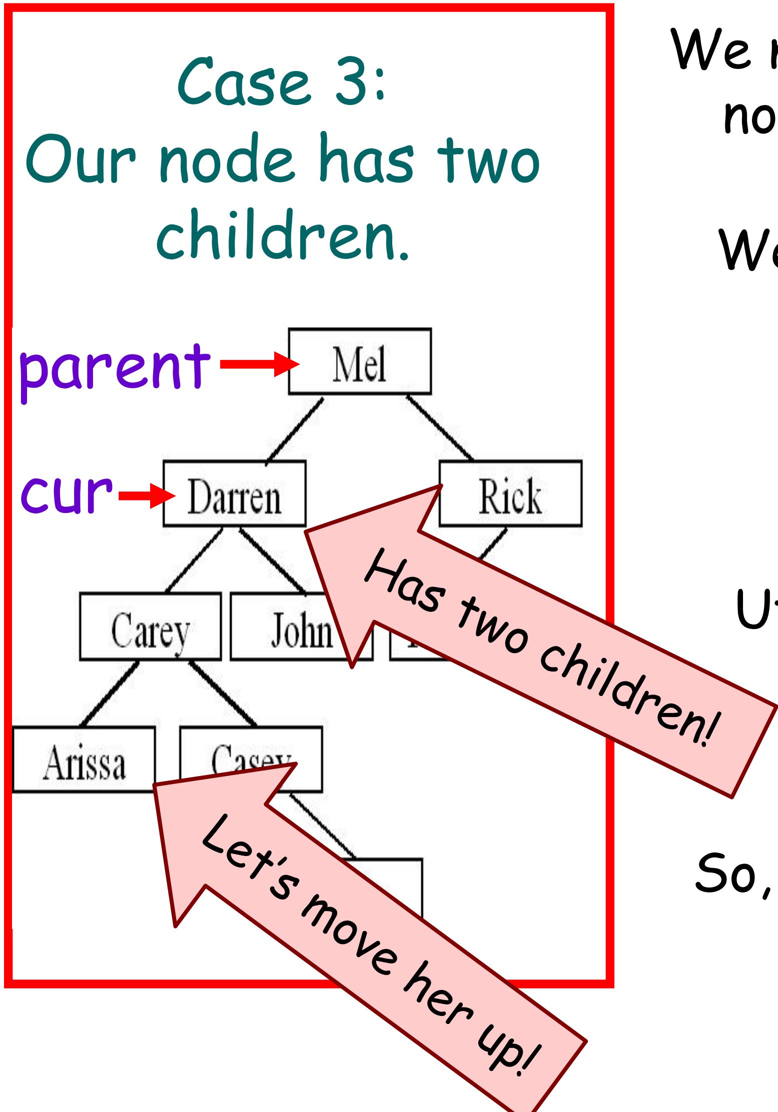
1. Relink the parent node to the target (cur) node's only child.
2. Then delete the target (cur) node.

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.

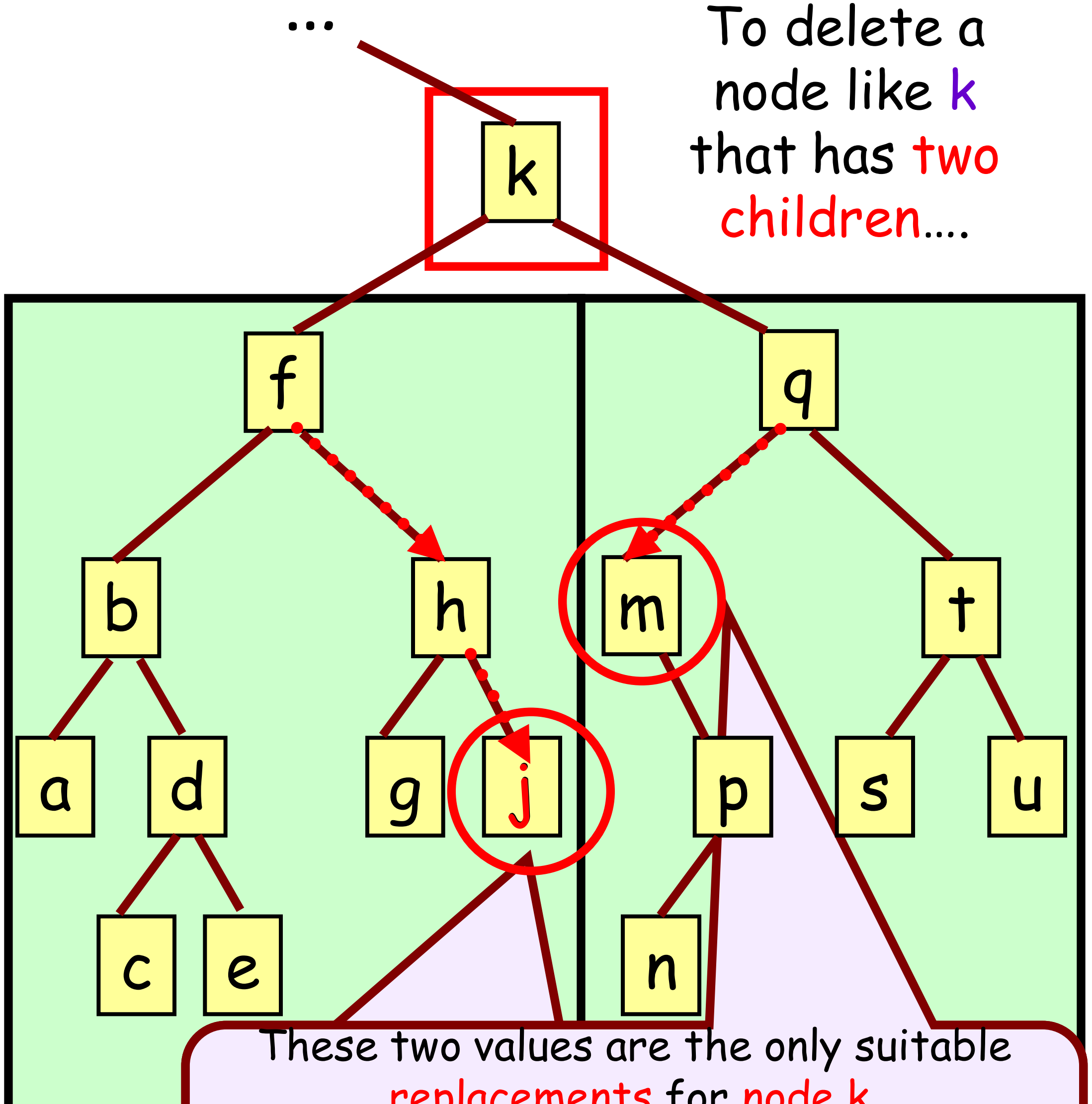
We can't just pick some arbitrary node and move it up into the vacated slot!

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



These two values are the only suitable replacements for node k.

Notice that both of them are either a leaf or have just one child!

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 an other node!

How? We want to replace k with either:

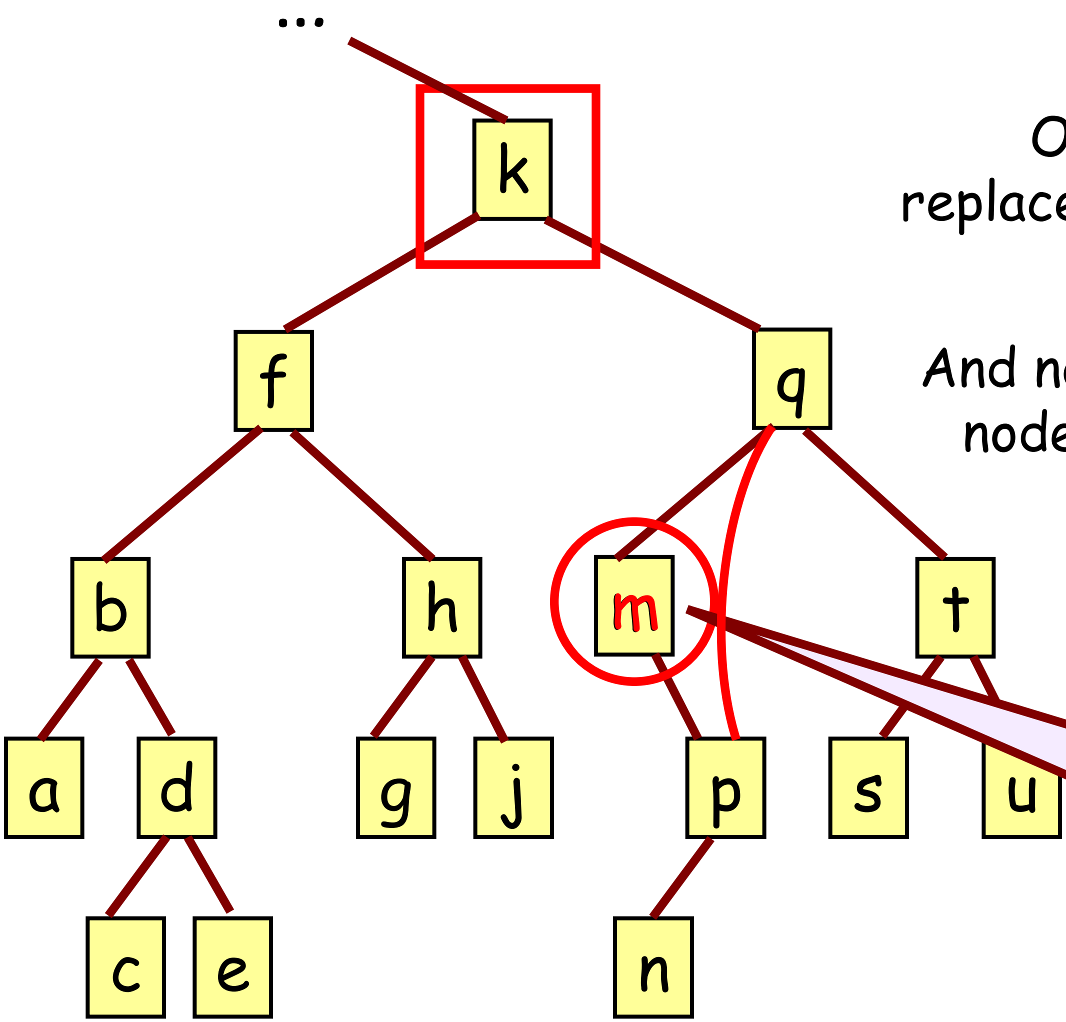
1. K's left subtree's largest-valued child

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

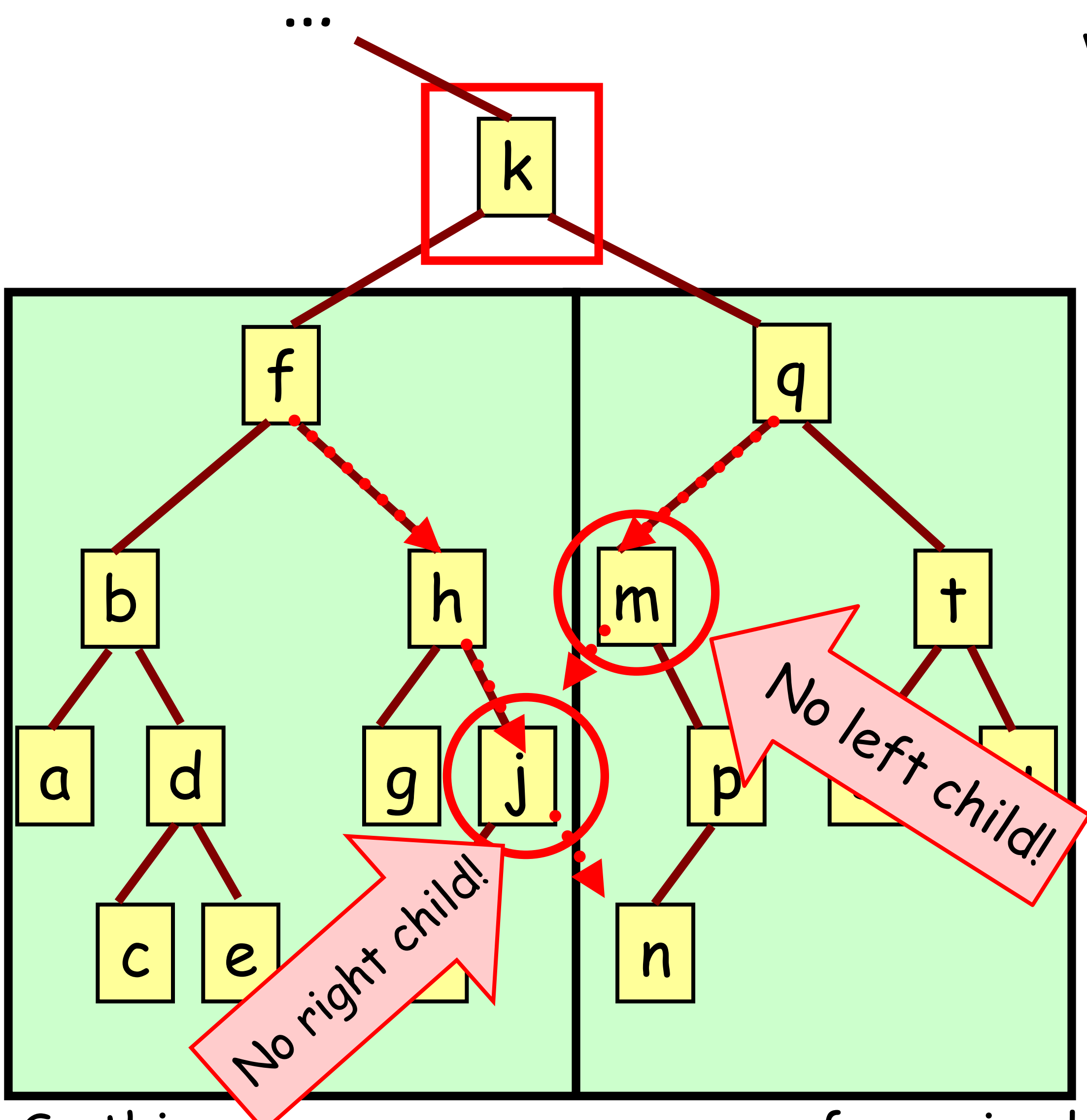


OK, now let's try the other replacement node and see if it works!

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

In this case, our node has one child, so we use Case 2.

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



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!

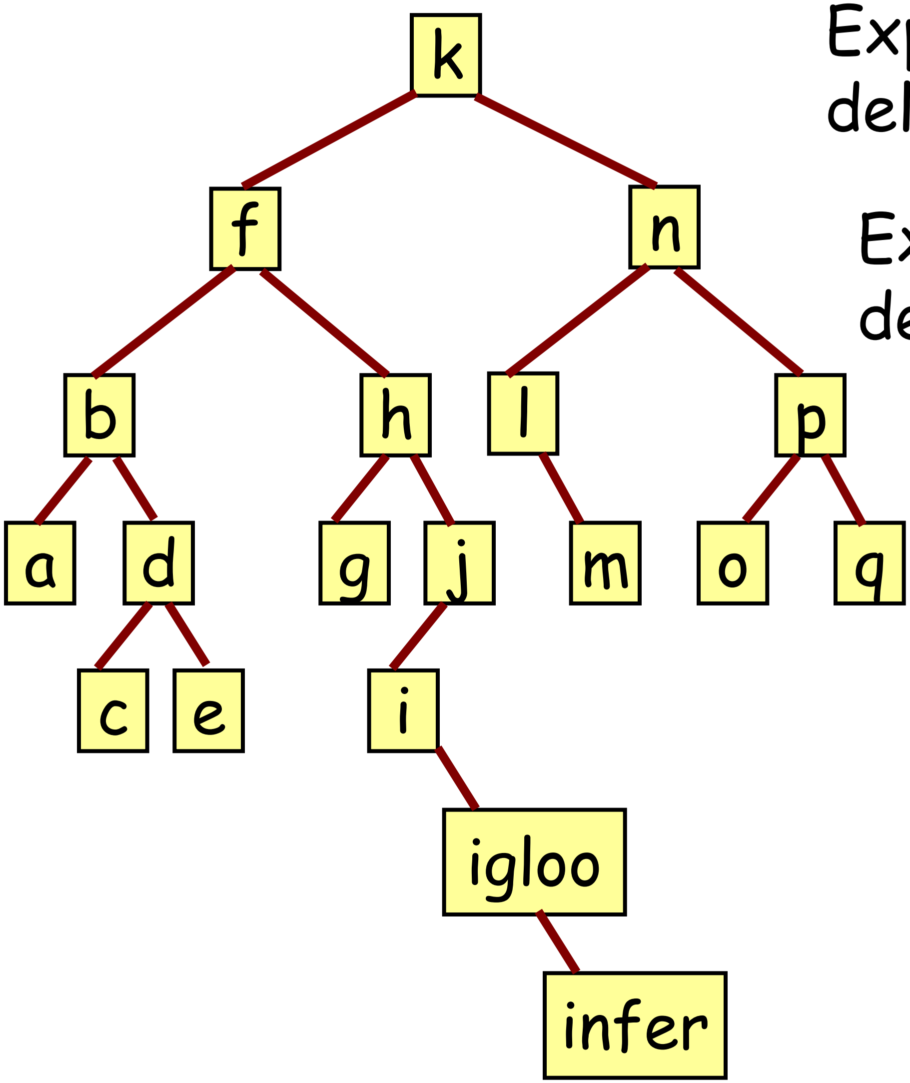
Either it has a left child or no children at all...

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

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

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

Deletion Exercise



Explain how you would *go* about deleting node k.

Explain how you would *go* about deleting node e.

Explain how you would *go* about deleting node i.

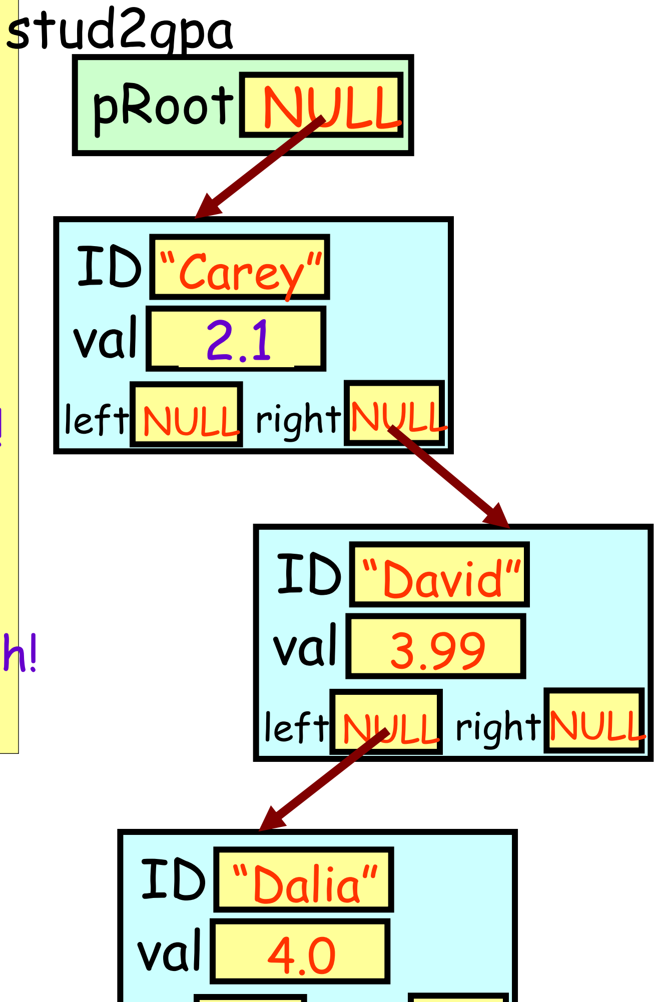
Where are Binary Search Trees Used?

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

}
```

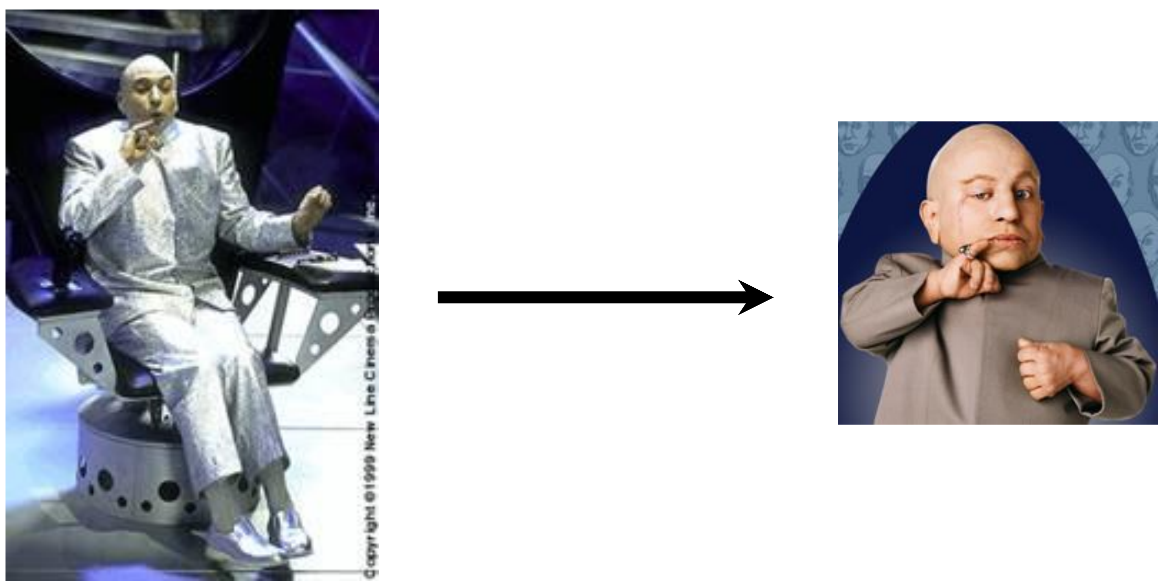
The STL set and map use binary search trees (a special balanced kind) to enable fast searching.

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

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

Huffman Encoding: Applying Trees to Real-World Problems

Huffman Encoding is a data compression technique that can be used to compress and decompress files (e.g. like creating ZIP files).



Background

Before we actually cover Huffman Encoding, we need to learn a few things…

Remember the ASCII code?

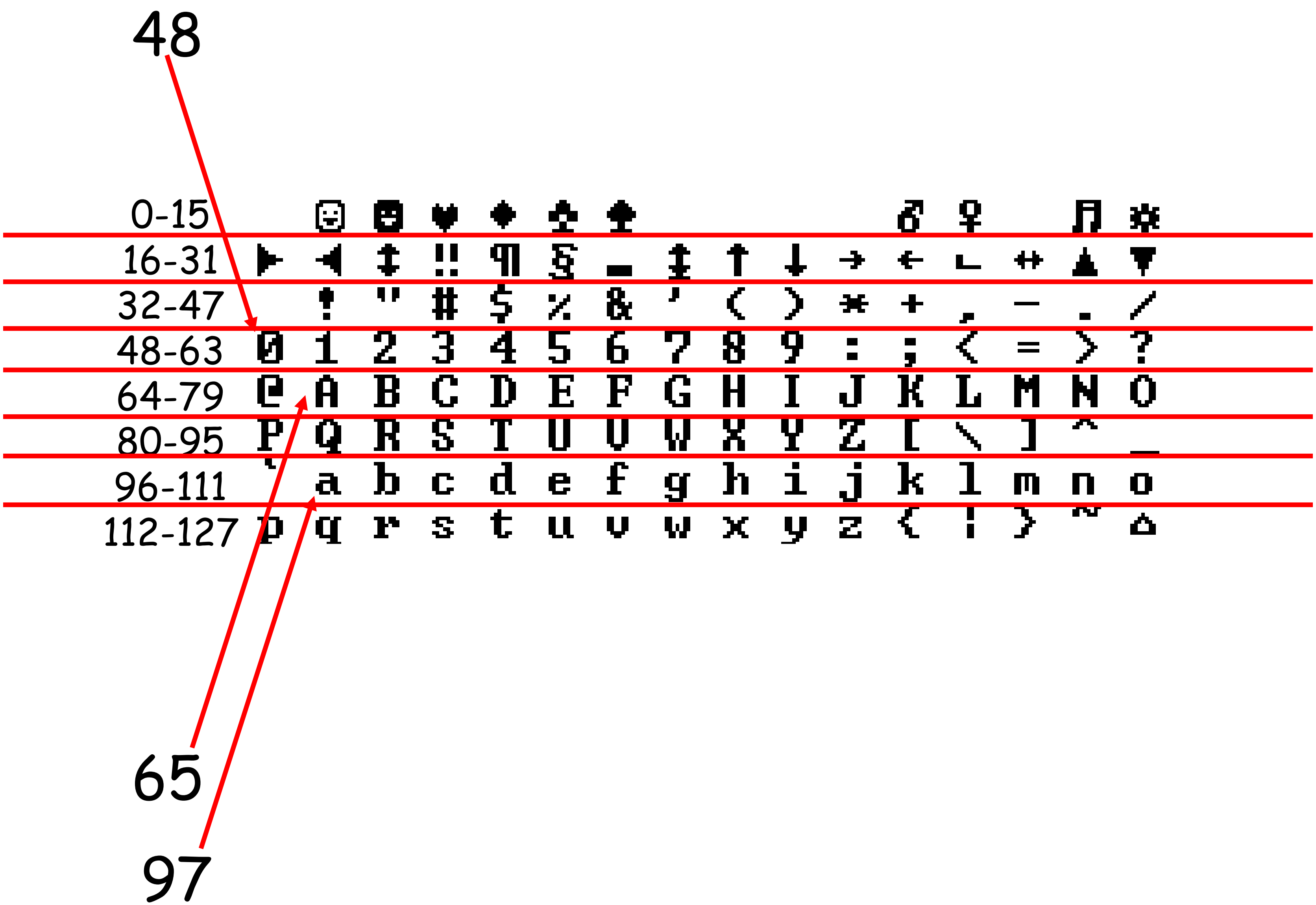
ASCII

Computers represent letters, punctuation and digit symbols using the ASCII code, storing each character as a number.

When you type a character on the keyboard, it's converted into a number and stored in the computer's memory!

50 65

The ASCII Chart



Computer Memory and Files

So basically, characters are stored in the computer's memory as numbers...

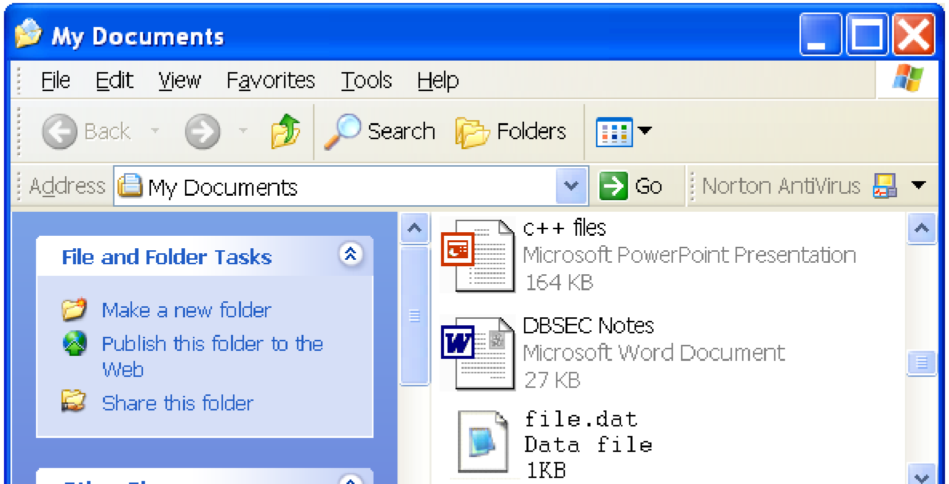
```
main()
{
    char data[7] = "Carey";

    ofstream out("file.dat");
    out << data;
    out.close();
}
```

data

67
97
114
101
121
0
0

Similarly, when you write data out to a file, it's stored as ASCII numbers too!



Bytes and Bits

Now, as you've probably heard, the computer actually stores all numbers as 1's and 0's (in binary) instead of decimal...

```
main()
{
   char data[7] = "Carey";

   ofstream out("file.dat");
   out << data;
   out.close();
}
```

data
01000011
01100001
01110010
01100101
01111001
00000000
00000000

Each character is represented by 8 bits.

Each bit can have a value of either 0 or 1
(i.e. 1 = high voltage and 0 = low voltage)

Binary and Decimal

Every decimal number has an equivalent binary representation
(they're just two ways of representing the same thing)

Decimal Number	Binary Equivalent
0	00000000
1	00000001
2	00000010
3	00000011
4	00000100
…	…
255	11111111

So that's binary…

Consider a Data File

Now lets consider a simple data file containing the data:

"I AM SAM MAM."

As we've learned, this is actually stored as 13 numbers in our data file:

73 32 65 77 32 83 65 77 32 77 65 77 46

And in reality, its *really* stored in the computer as a set of 104 binary digits (bits):

01001001 00100000 01000001 01001101 00100000 01010011 01000001 01001101 00100000 01001101 01000001 01001101 00101110

(13 characters * 8 bits/character = 104 bits)

Data Compresion

So our original string "I AM SAM MAM." requires 104 bits to store on our computer... OK.

01001001 00100000 01000001 01001101 00100000 01010011 01000001 01001101 00100000 01001101 01000001 01001101 00101110

The question is:

Can we somehow reduce the number of bits required to store our data?

And of course, the answer is YES!

Huffman Encoding

To compress a file "file.dat" with Huffman encoding, we use the following steps:

1. Compute the frequency of each character in file.dat.
2. Build a Huffman tree (a binary tree) based on these frequencies.
3. Use this binary tree to convert the original file's contents to a more compressed form.
4. Save the converted (compressed) data to a file.

Huffman Encoding: Step #1

Step #1: Compute the frequency of each character in file.dat. (i.e. compute a *histogram*)

FILE.DAT

I AM SAM_MAM.

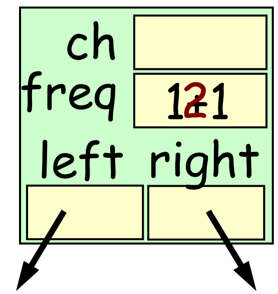
'A'	**3**
'I'	**1**
'M'	**4**
'S'	**1**
Space	**3**
Period	**1**

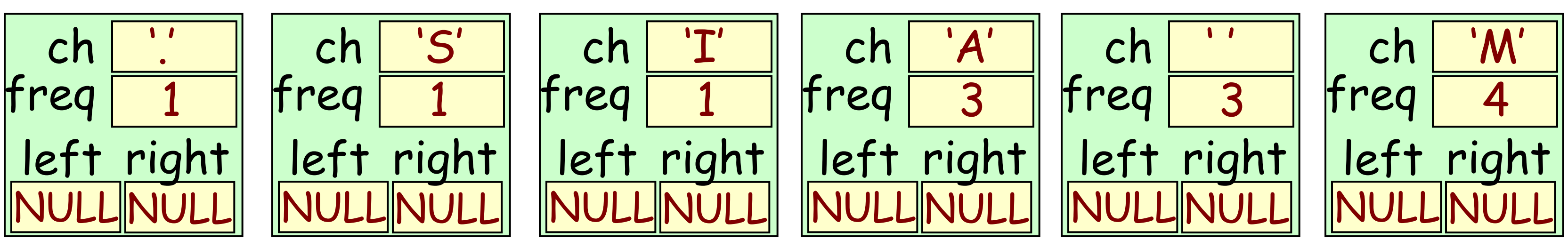
Huffman Encoding: Step #2

Step #2: Build a Huffman tree (a binary tree) based on these frequencies:

A. Create a binary tree leaf node for each entry in our table, but don't insert any of these into a tree!

B. While we have more than one node left:
   1. Find the two nodes with lowest freqs.
   2. Create a new parent node.
   3. Link the parent to each of the children.
   4. Set the parent's total frequency equal to the sum of its children's frequencies.
   5. Place the new parent node in our grouping.

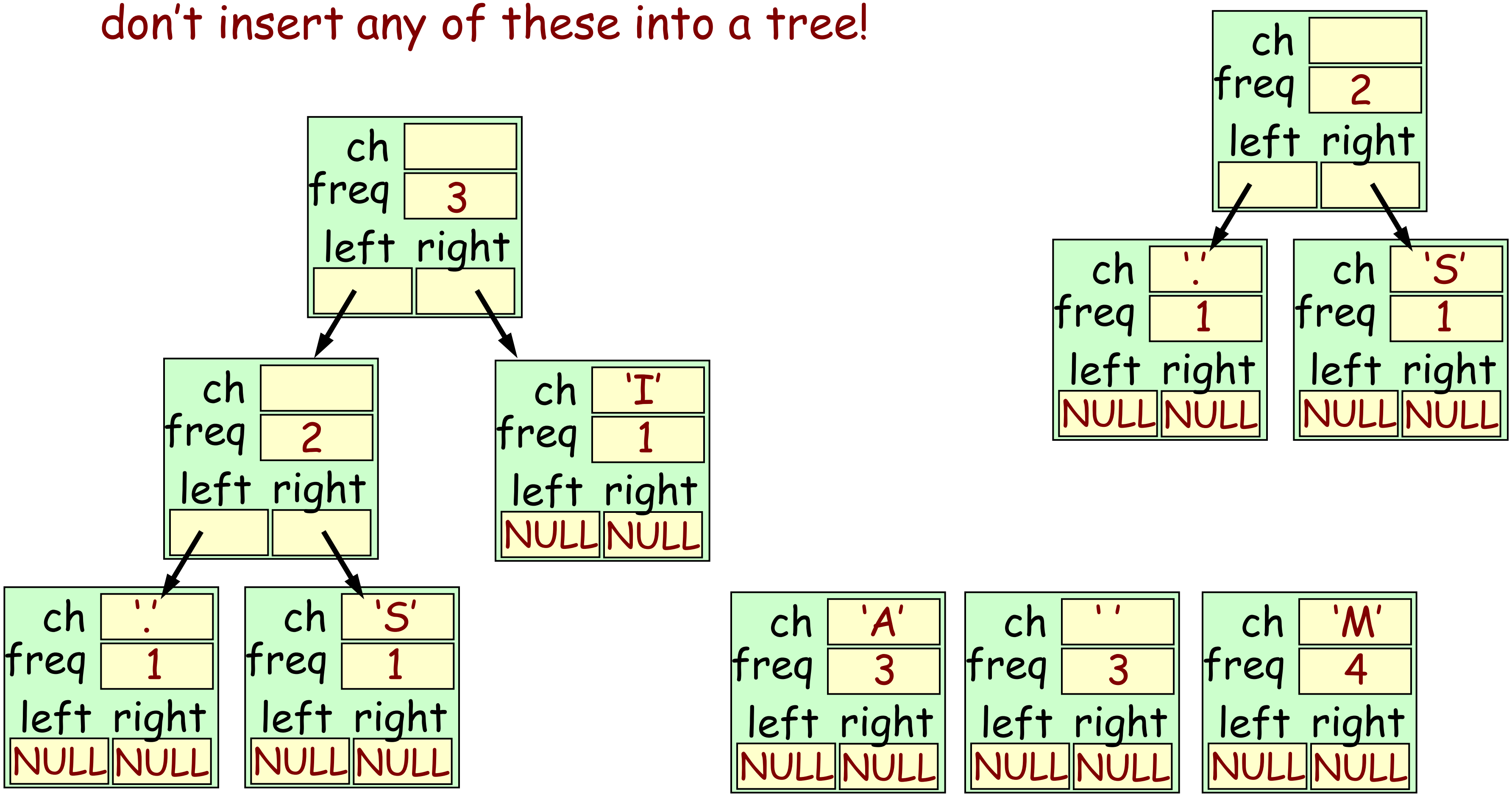




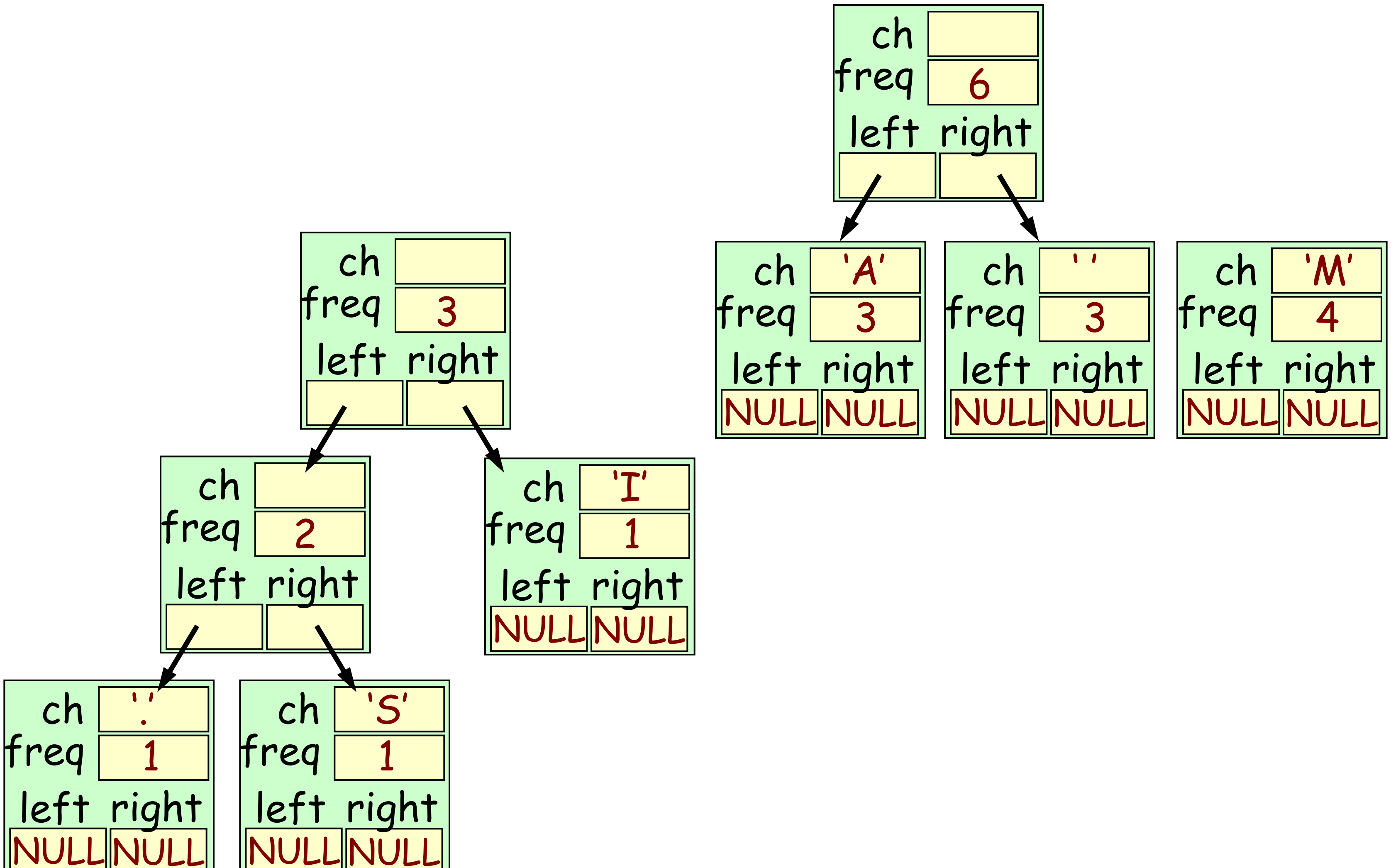
Huffman Encoding: Step #2

Step #2: Build a Huffman tree (a binary tree) based on these frequencies:

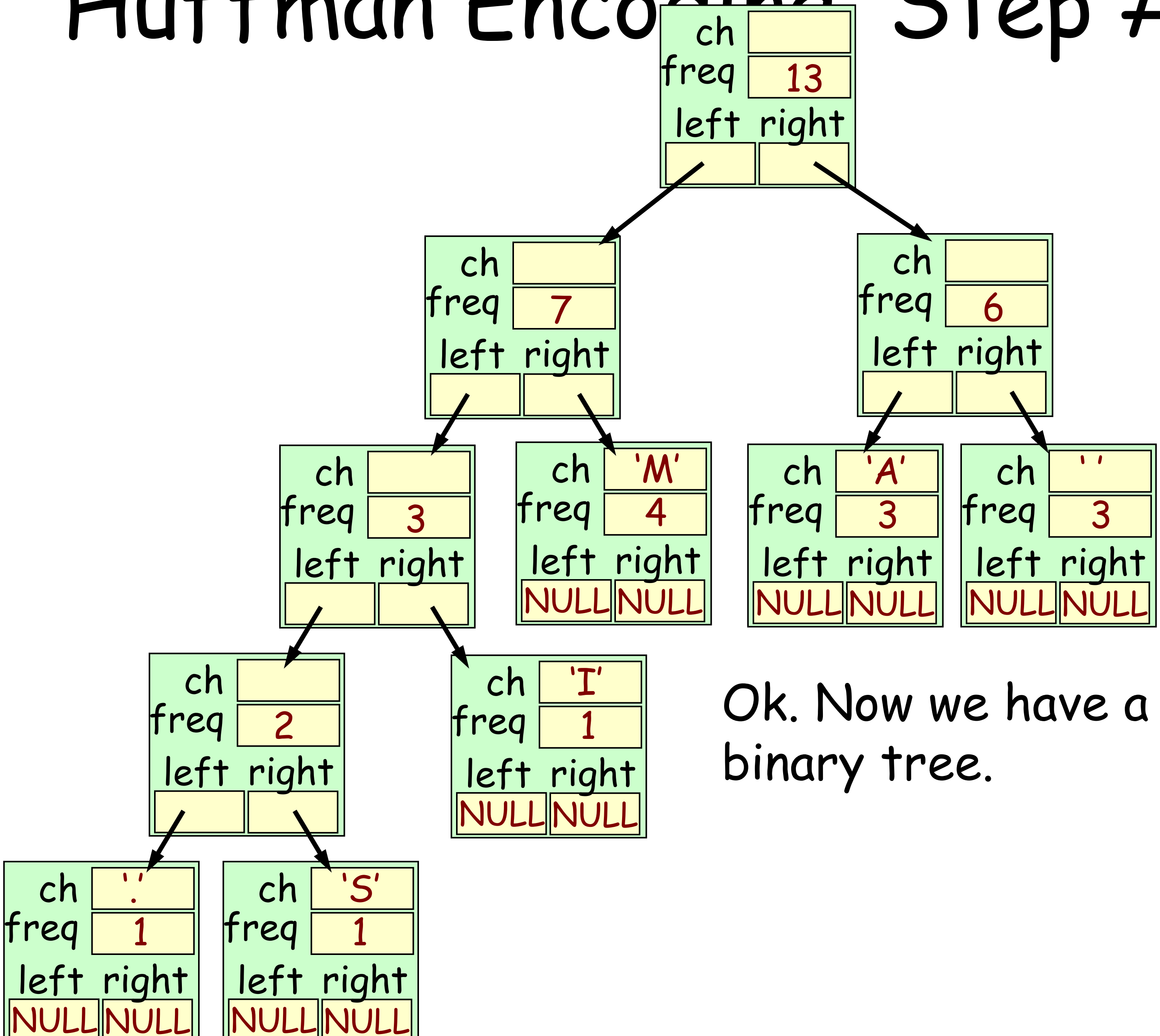
A. Create a binary tree leaf node for each entry in our table, but don't insert any of these into a tree!



Huffman Encoding: Step #2



Huffman Encoding: Step #2



Ok. Now we have a single binary tree.

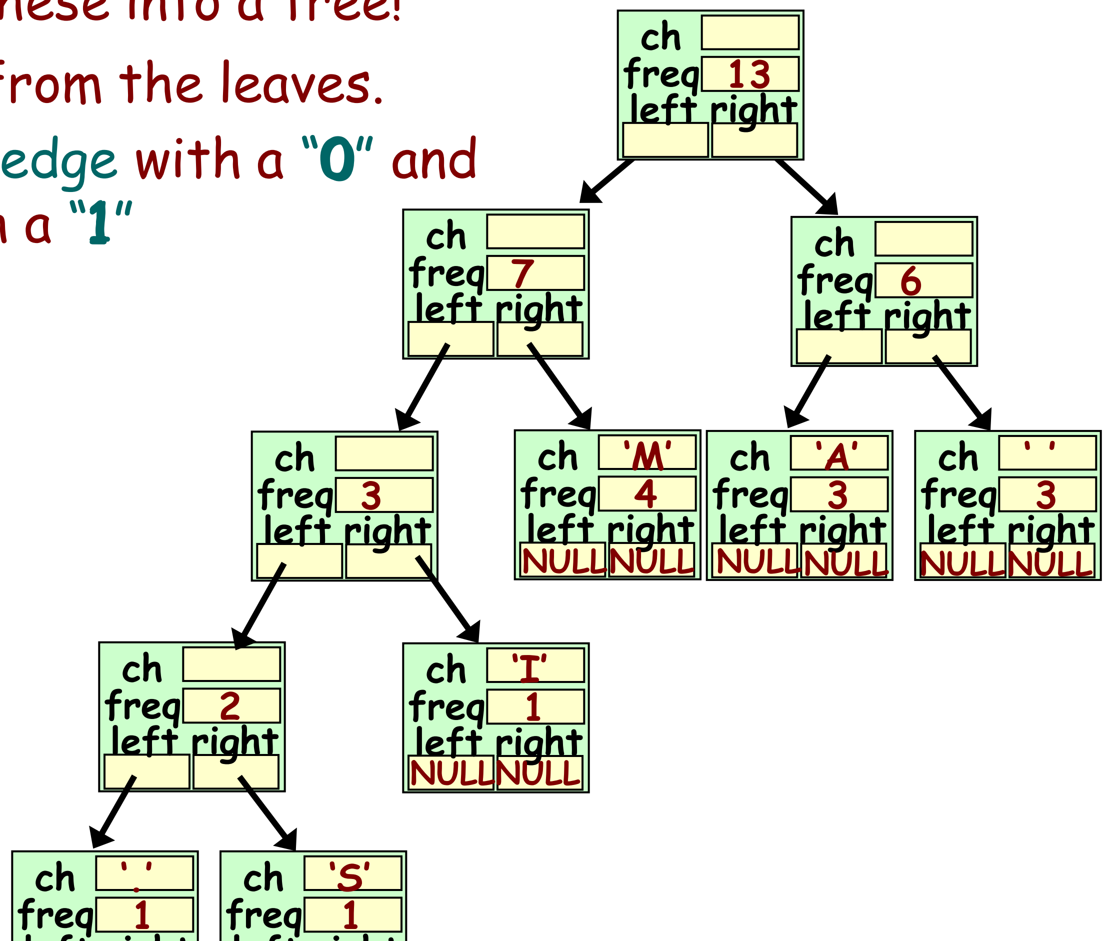
Huffman Encoding: Step #2

Step #2: Build a Huffman tree (a binary tree) based on these frequencies:

A. Create a binary tree leaf node for each entry in our table, but don't insert any of these into a tree!

B. Build a binary tree from the leaves.

C. Now label each left edge with a "**0**" and each right edge with a "**1**"



Huffman Encoding: Step #2

Now we can determine the new bit-encoding for each character.

The bit encoding for a character is the path of 0's and 1's that you take from the root of the tree to the character of interest.

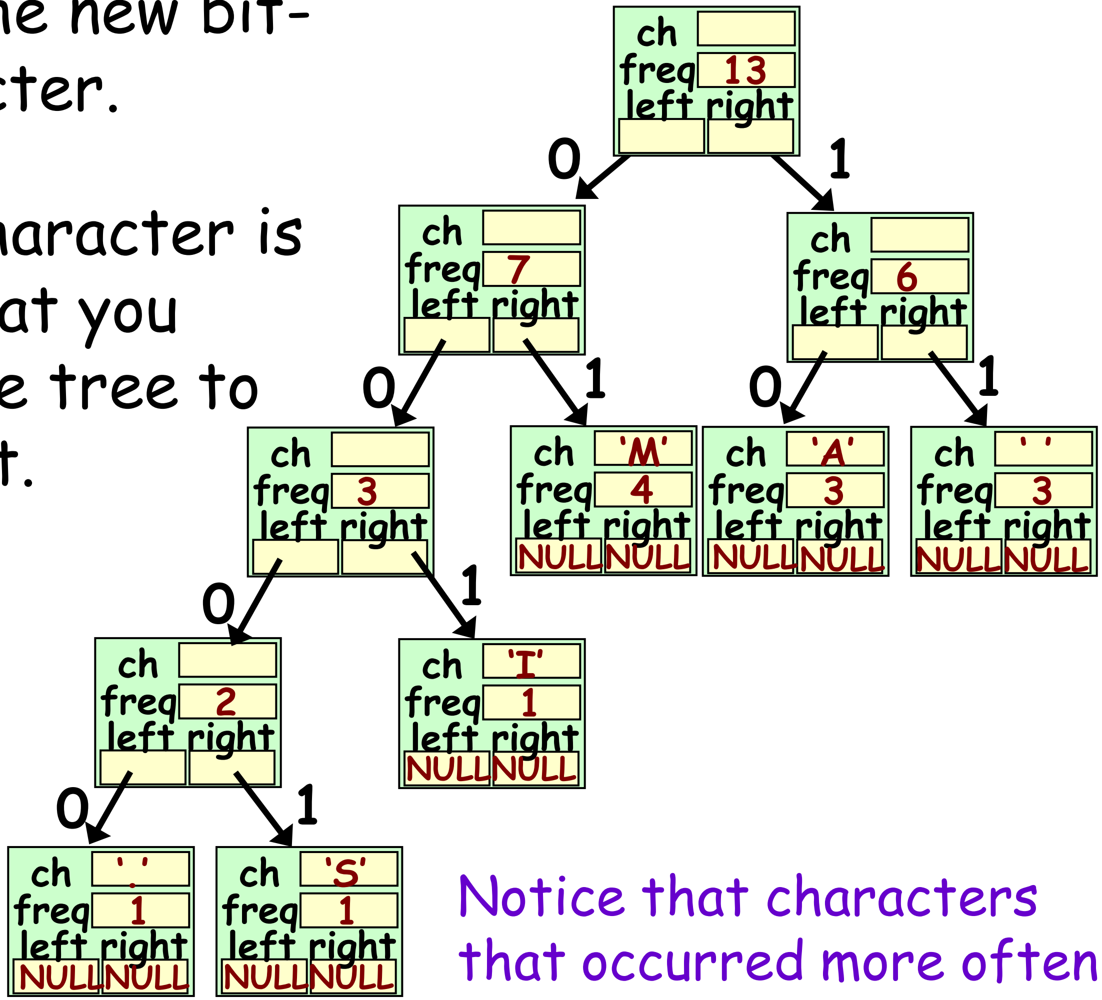
For example:

S is encoded as 0001

A is encoded as 10

M is encoded as 01

Etc…



Notice that characters that occurred more often in our message have shorter bit-encodings!

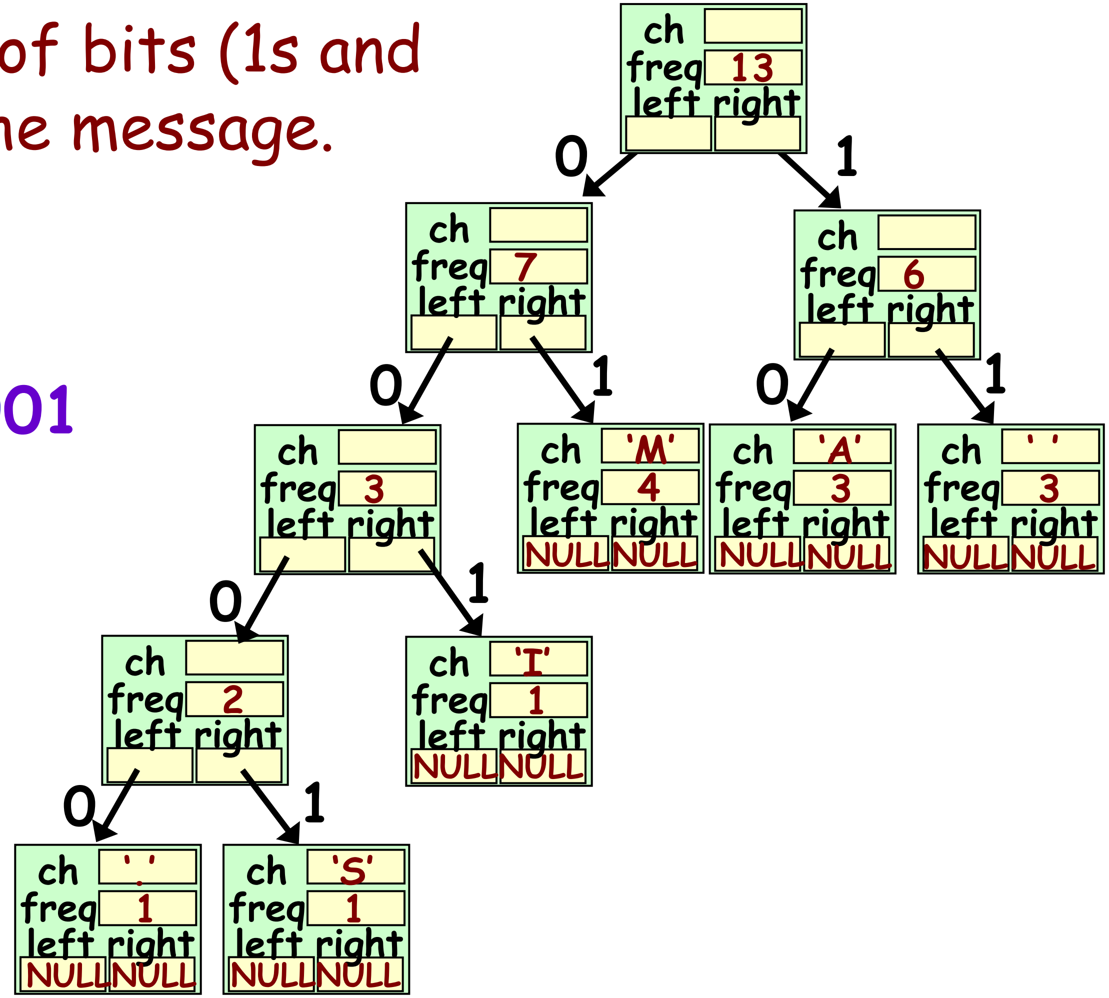
Huffman Encoding: Step #3

Step #3: Use this binary tree to convert the original file's contents to a more compressed form..

i.e. find the sequence of bits (1s and 0s) for each char in the message.

I AM SAM MAM.

0011110011100011001
110110010000



Huffman Encoding: Step #4

Step #4: Save the converted (compressed) data to a file.

001 1110011100011001110110010000

compressed.dat

Notice that our new file less than four bytes or 31 bits long!

Our original file is 13 bytes or 104 bits long!

We saved over 69%!

originalfile.dat

01001001 00100000 01000001
01001101 00100000 01010011
01000001 01001101 00100000
01001101 01000001 01001101
00101110

Ok… So I cheated a bit…

compressed.dat

Encoding:
'A' = "10"
' ' = "11"
'M' = "01"
'I' = "001"
'.' = "0000"
'S' = "0001"
Encoded Data:
001 1110
01110001
10011101
10010000

If all we have is our 31 bits of data… its impossible to interpret the file!

Did 000 equal "I" or did 000 equal "Q"? Or was it 00 equals "A"?

So, we must add some additional data to the top of our compressed file to specify the encoding we used…

Now clearly this adds some overhead to our file…

But usually there's a pretty big savings anyway!

Decoding…

1. Extract the encoding scheme from the compressed file.
2. Build a Huffman tree (a binary tree) based on the encodings.
3. Use this binary tree to convert the compressed file's contents back to the original characters.
4. Save the converted (uncompressed) data to a file.

compressed.dat

Encoding:

'A' = "10"
' ' = "11"
'M' = "01"
'I' = "001"
'.' = "0000"
'S' = "0001"

Encoded Data:

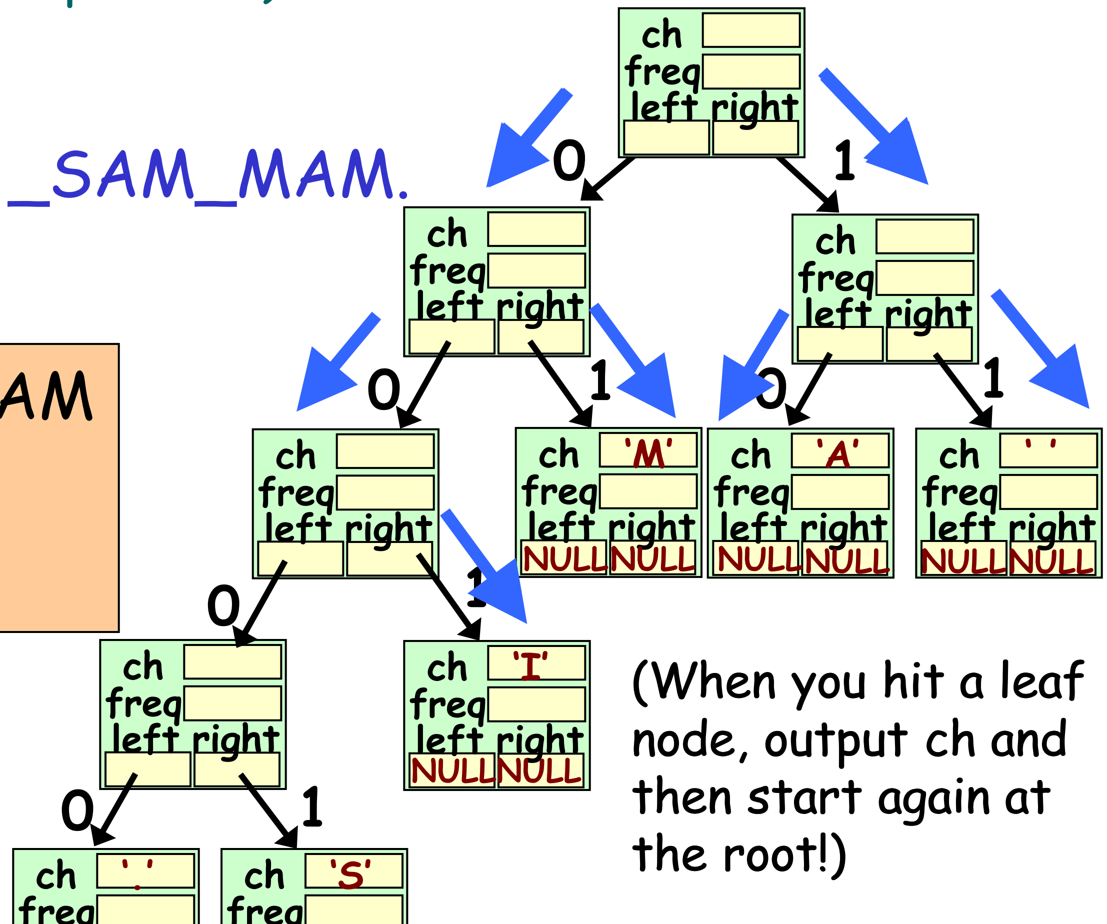
001 1110
01110001
10011101
10010000

Output:

I_AM_SAM_MAM.

output.dat

I AM SAM
MAM.



(When you hit a leaf node, output ch and then start again at the root!)

Balanced Search Trees

Question: What happens if we insert the following values into a binary search tree?

5, 10, 7, 9, 8, 20, 18, 17, 16, 15, 14, 13, 12, 11

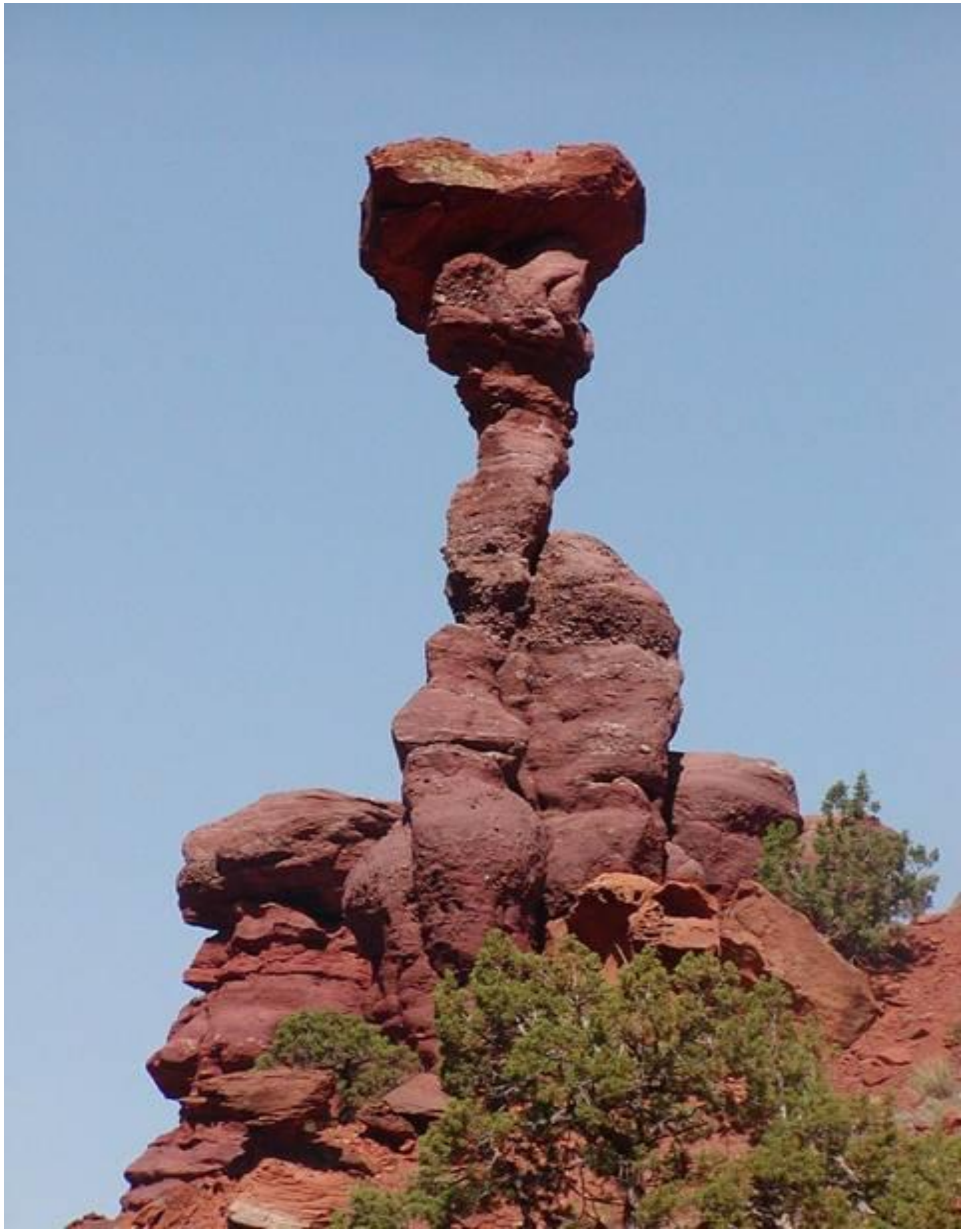
Question: What is the *approximate* big-oh cost of searching for a value in this tree?

Balanced Search Trees

In real life, trees often end up looking just like our example, especially after repeated insertions and deletions.

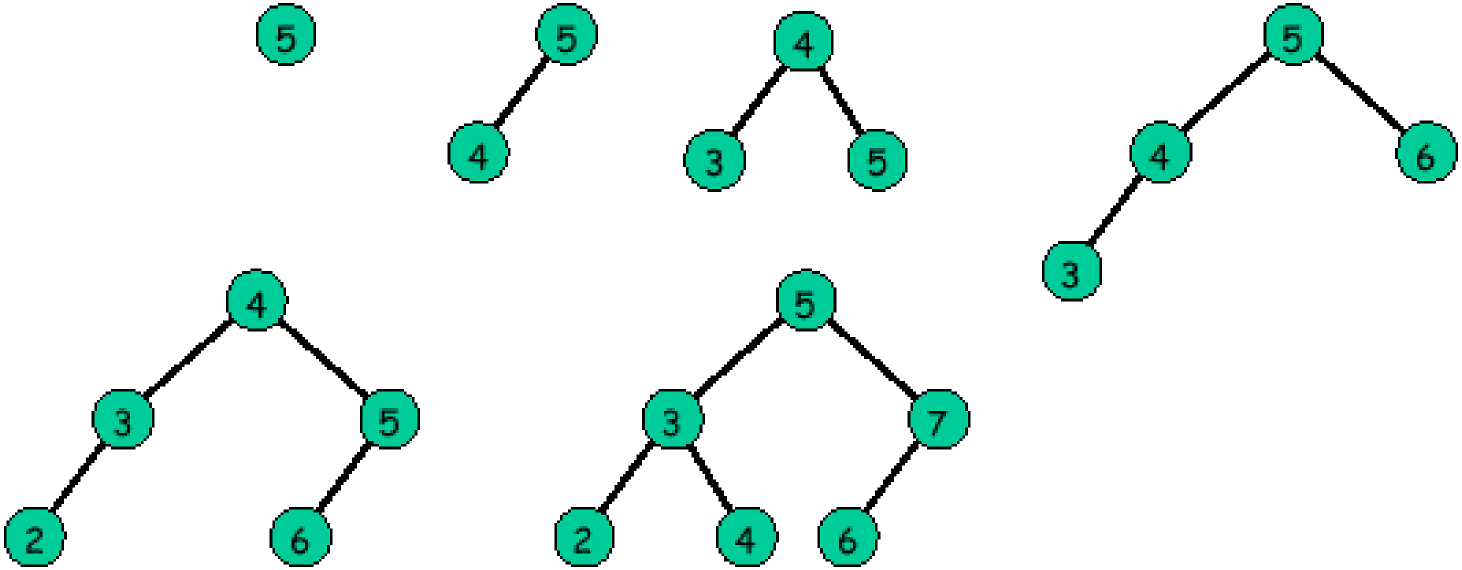
It would be nice if we could come up with a tree ADT that *always maintained an even height*.

This would ensure that all insertions, searches and deletions would be O(log n).



Balanced Search Trees

A binary tree is "perfectly balanced" if for each node, the number of nodes in its left and right subtrees differ by at most one.



Perfectly balanced search trees have a maximum height of log(n), but are difficult to maintain during insertion/deletion.

Balanced Search Trees

There are 3 popular approaches to building a balanced binary search tree :

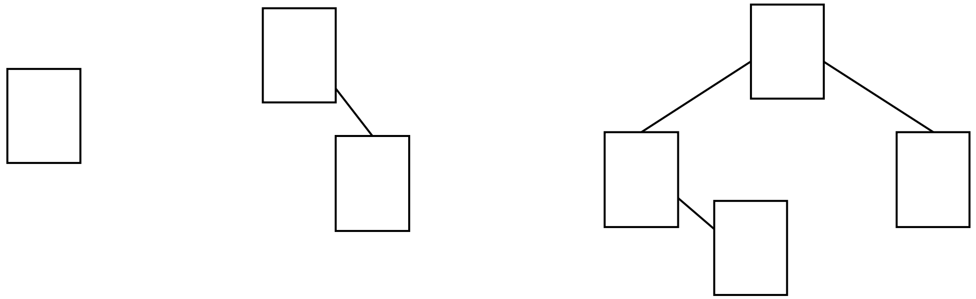
- AVL trees
- 2-3 trees
- Red-black trees

Let's learn about AVL trees!

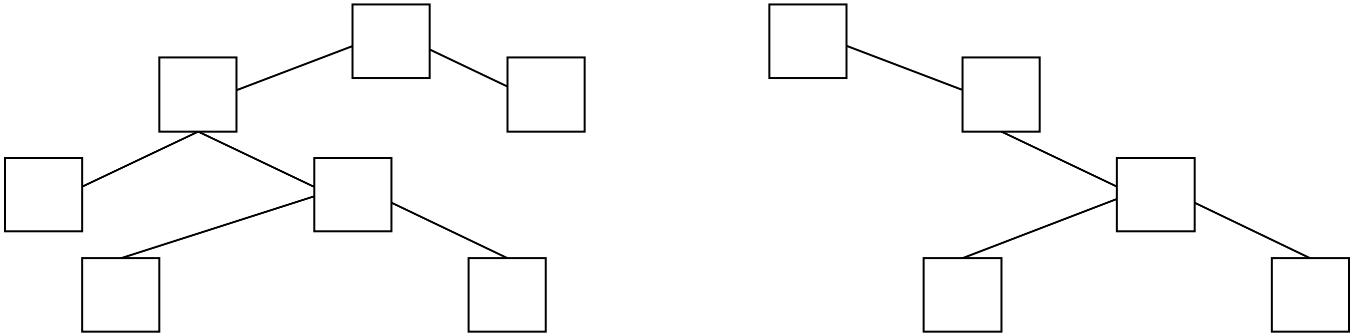
AVL Trees

AVL tree: a BST in which the heights of the left and right *sub-trees* of each node differ at most by 1.

AVL Trees



Non-Legal AVL Trees



Implementing AVL Trees

- To implement an AVL tree, *each* node has a new **balance value**:

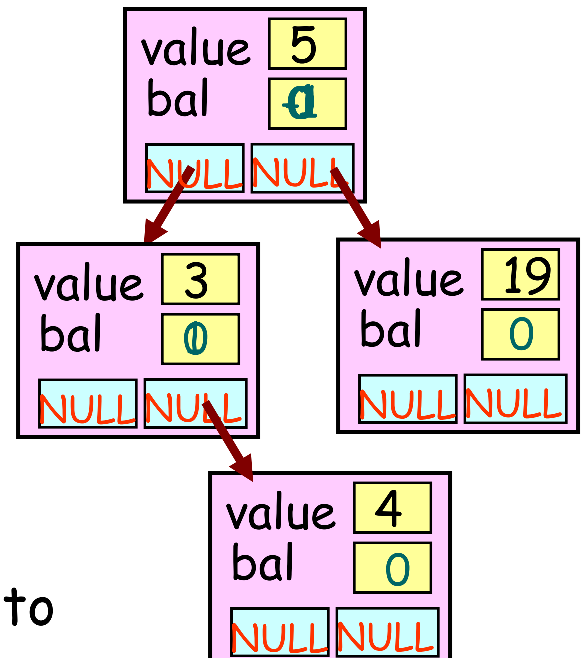
  0 if the node's left and right subtrees have the same height

  -1 if the node's left subtree is 1 higher than the right

  1 if the node's right subtree is 1 higher than the left

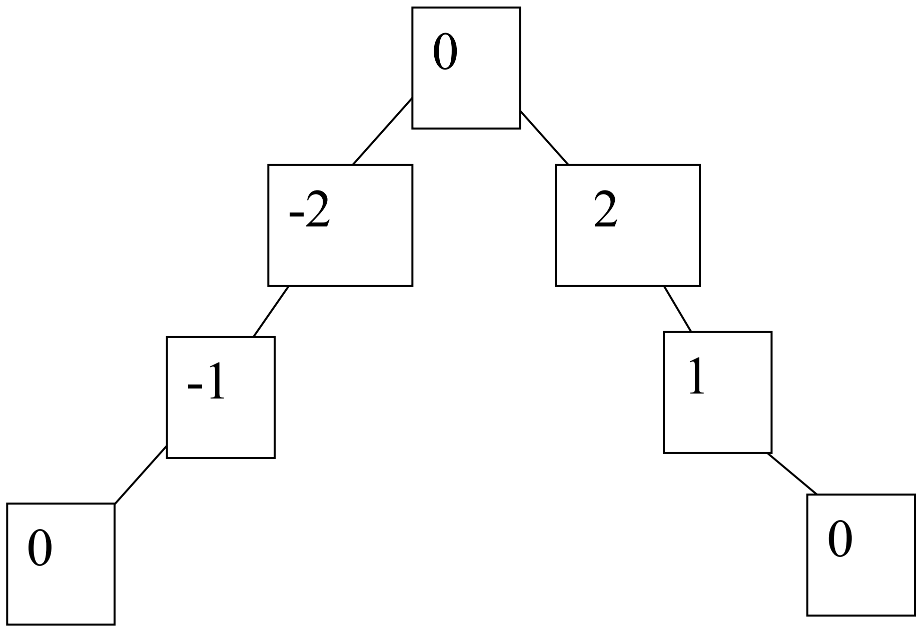
```
struct AVLNode
{
    int          value;

    int          balance;
    AVLNode      *left, *right;
};
```



When we insert a node we have to update all balance values from the new node to the root of the tree…

A Non-AVL Tree



Notice: Even though the root node has a balance of zero, its sub-trees aren't balanced!

Insertion into an AVL Tree

(the simple case)

Case 1: After you insert the new node, all nodes in the tree *still* have balances of 0, -1 or 1.

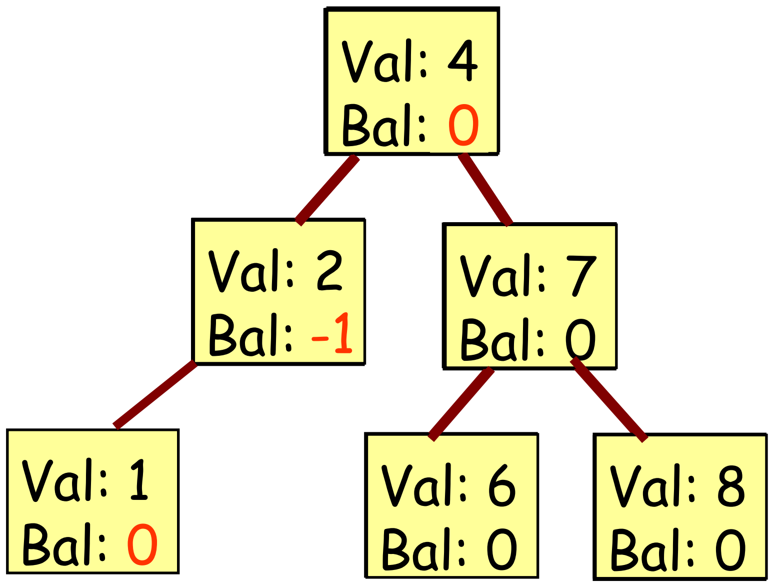
1. Insert the new node normally (just like in a BST)
2. Then update the balances of all parent nodes, from the new leaf to the root.
3. If all of the balances are still 0, -1 or 1, DONE!

Balance values:
-1 = left higher
1 = right higher
0 = even height

Insertion into an AVL Tree

(the simple case)

Insert the following values into an AVL tree, showing the balance factor of each node as you go: 4, 7, 2, 8, 6, 1



Balance values:
- -1 = left higher
- 1 = right higher
- 0 = even height

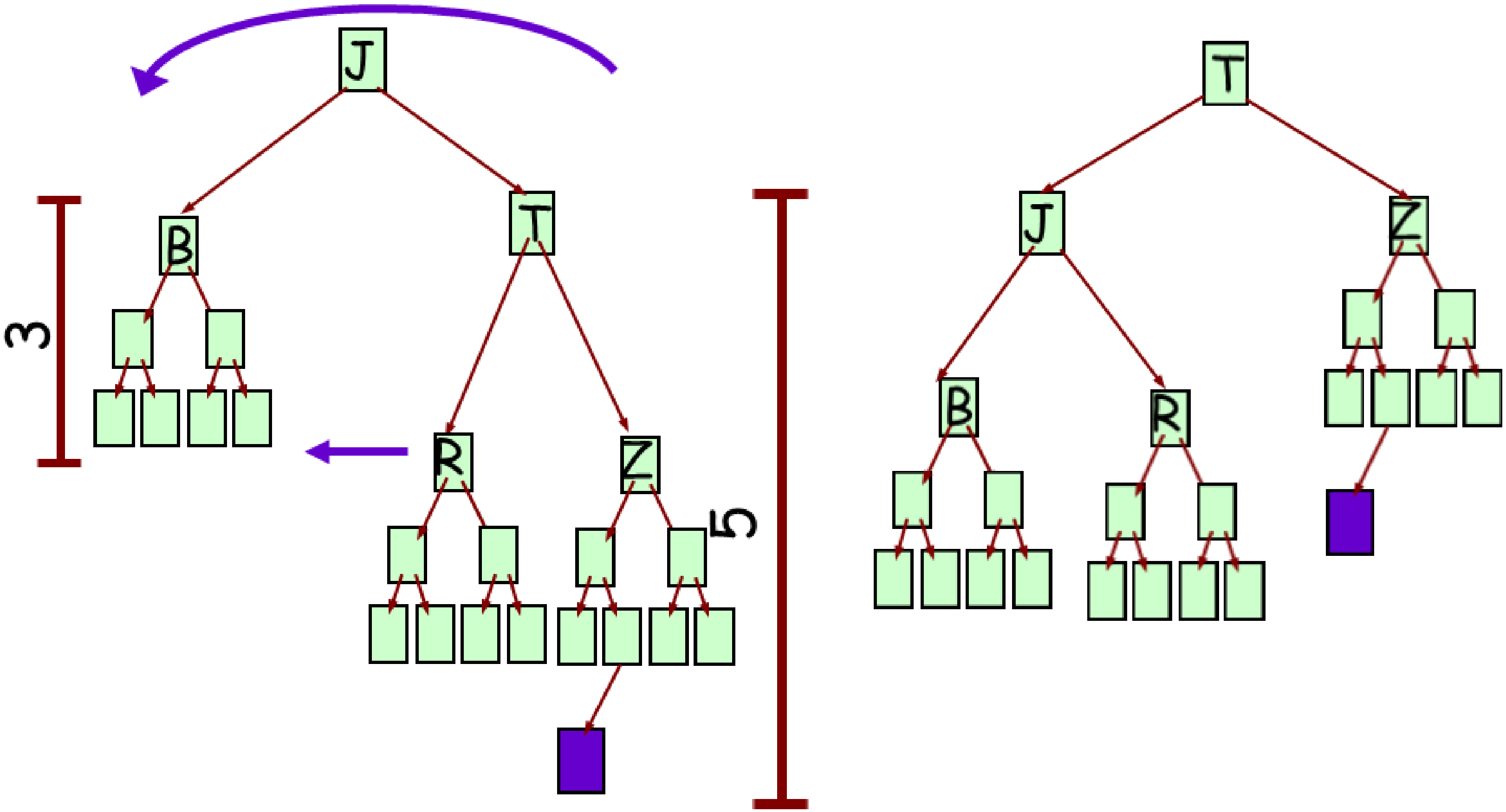
Insertion into an AVL Tree
(more complex case)

Case 2: When you insert a new node, it causes one or more balances to become less than -1 or greater than 1.

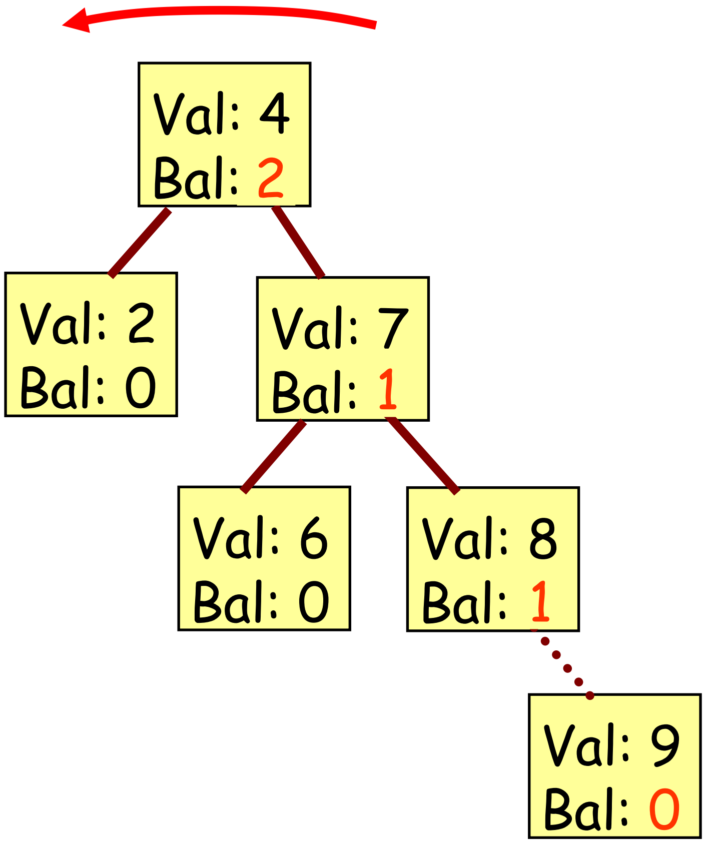
- We must now “rebalance” the tree so all nodes still have a balance of –1, 0 or 1.
- Let’s consider the case of inserting into the right subtree (left is similar)
- There are two sub-cases to consider:
  - Single rotation
  - Double rotation

Case 2.1: Single Rotation

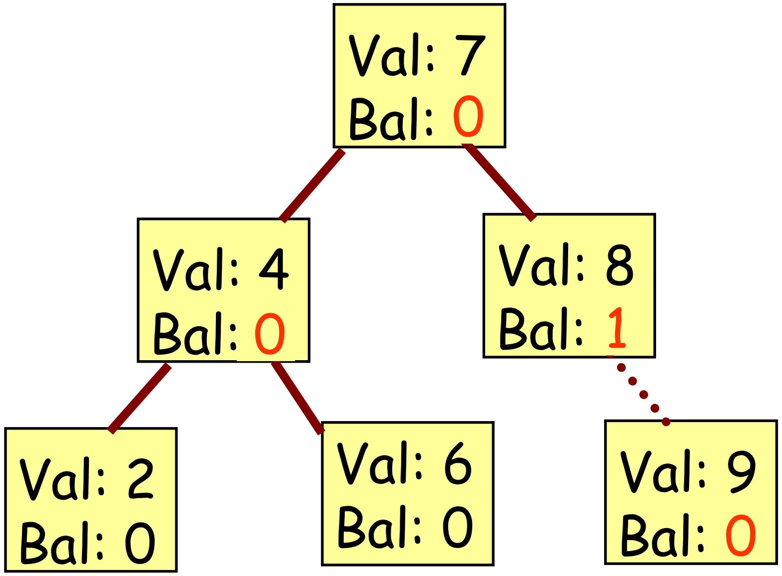
If we add a node to a subtree that causes any node in the AVL tree to go out of balance, then we rotate:



Case 2.1: Single Rotation



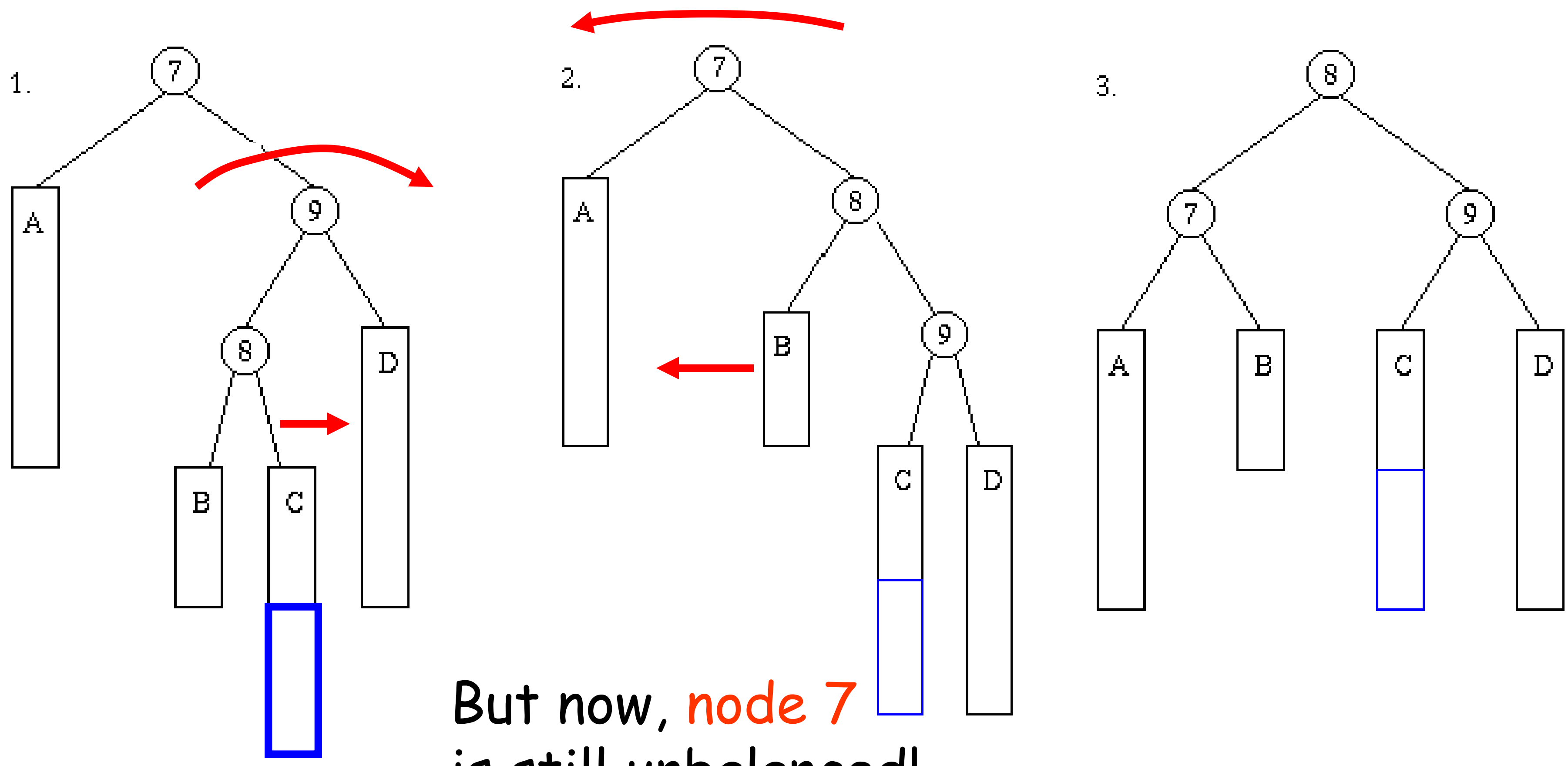
So, what would our new tree look like if we add a value of 9?



The nodes 2 4 7 8 9, that comprise the 'outside edge' of the tree, have been rotated one node to the left.

Case 2.1: Double Rotation

In some cases, when we add a node, it requires two rebalances.



But now, node 7 is still unbalanced!