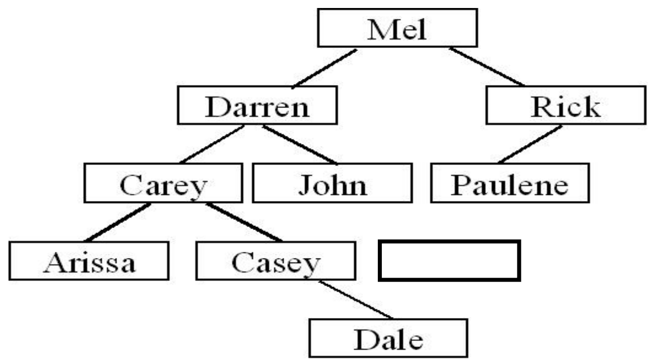
Lecture #13

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

Binary Trees, Cont.

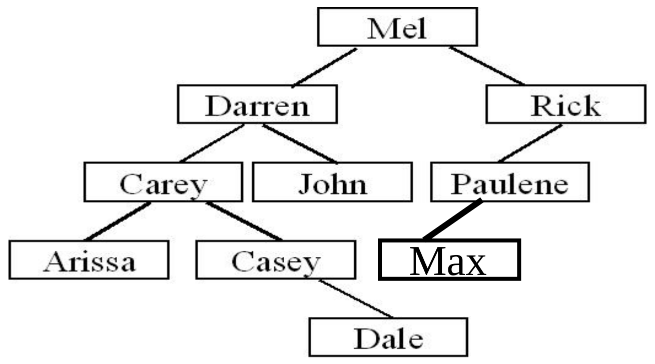
I'VE FINALLY FOUND IT... AFTER 15 YEARS
Robotatertotcomics
THE SCROLL OF TRUTH!
Writing documentation for your code helps you out in the long run
NYEHHH

Binary Tree Review



Question #1: Is the above tree a valid binary search tree?

Question #2: How about now?

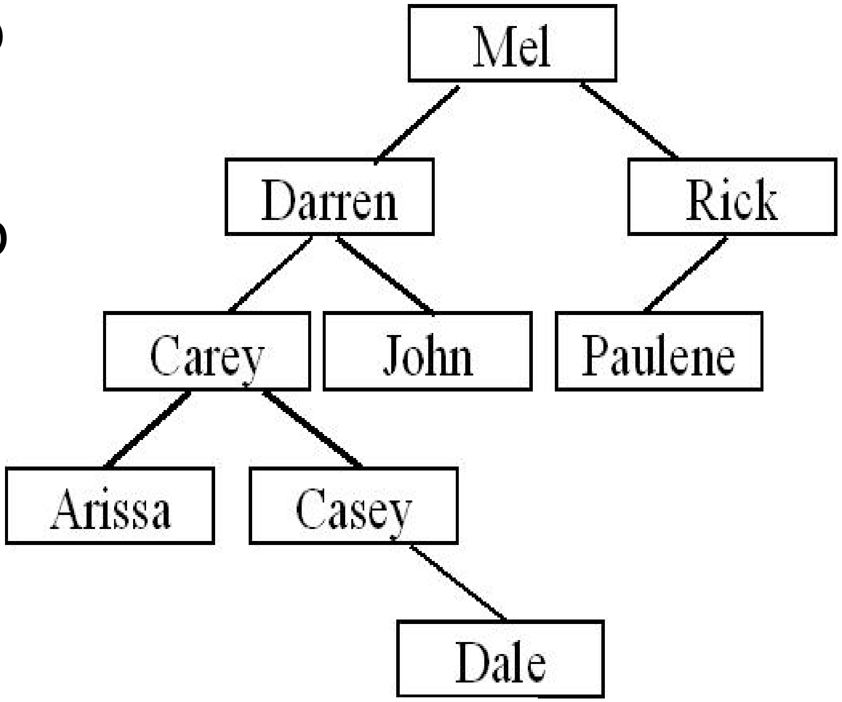


Answers:
1. Yes
2. No - Max is less than Mel (at the root) so it should be left of Mel

Binary Search Tree Insertion Review

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

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

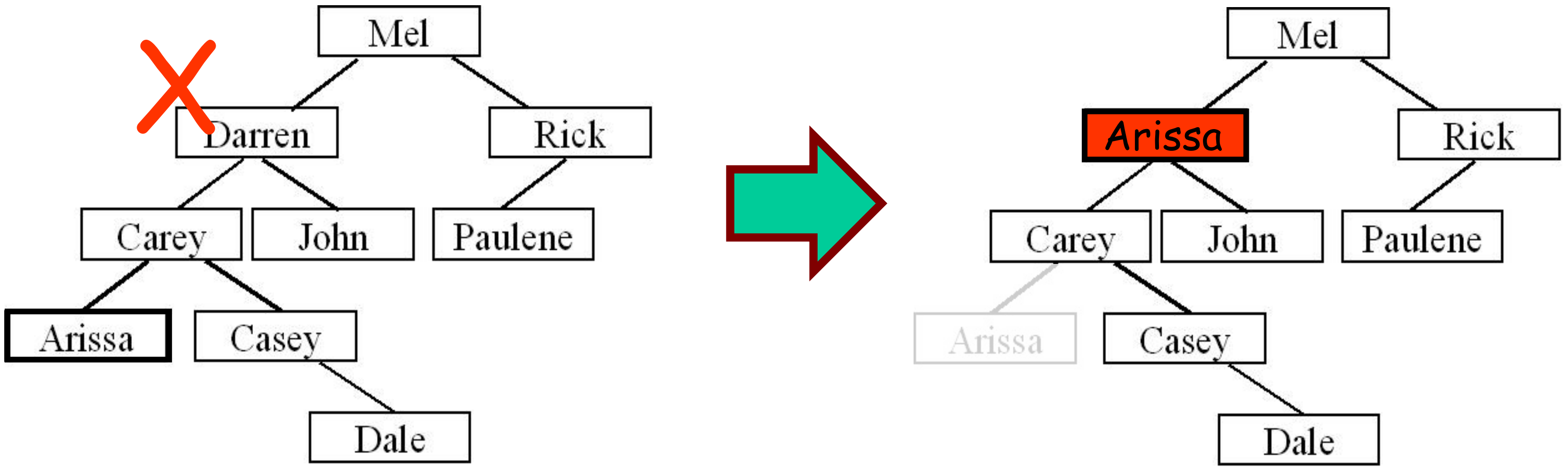


Answers:
1. Cathy would be added as the left child of Dale
2. Priyank would be added as the right child of Paulene

Deleting a Node from a Binary Search Tree

- Now, let's learn how to delete an item from a BST.
- Let's say we want to delete Darren from our tree…
- 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?
- No it's not!

- By simply moving an arbitrary node into Darren's slot, we violate our Binary Search Tree ordering requirement!
- Carey, Casey and Dale are NOT less than Arissa, so it breaks our tree!
- Next we'll see how to do this properly….



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!

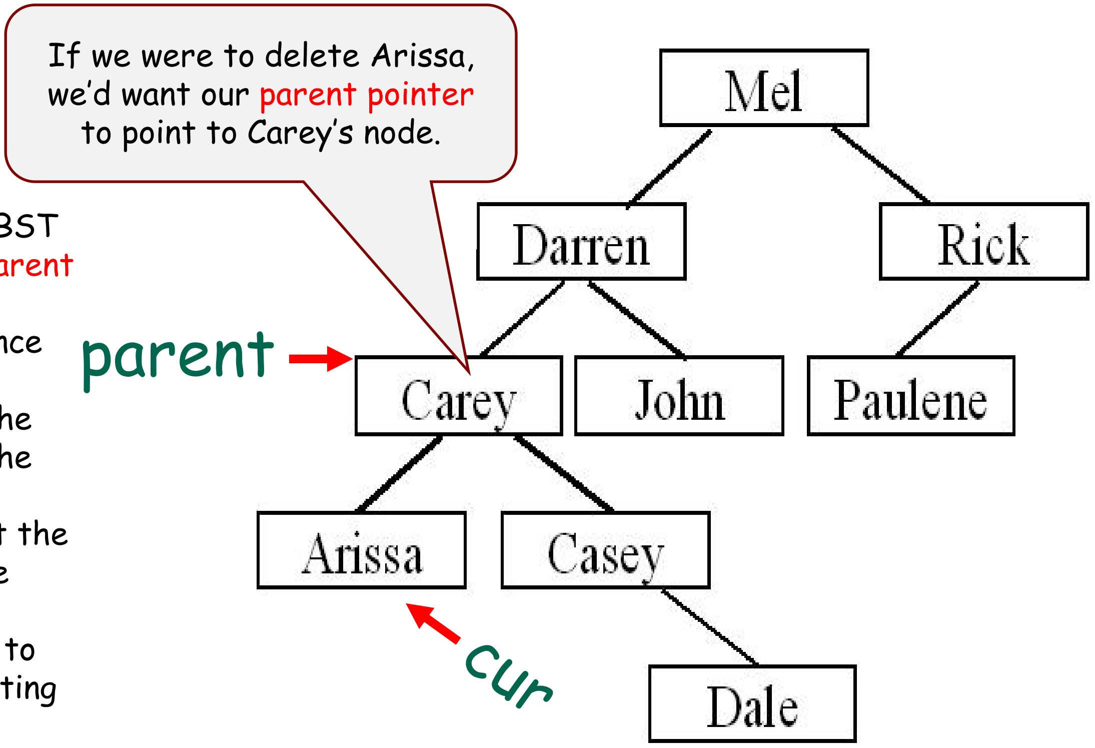
BST Deletion: Step #1

Here is the pseudocode for the first major step of deleting a node from a BST.

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;
```

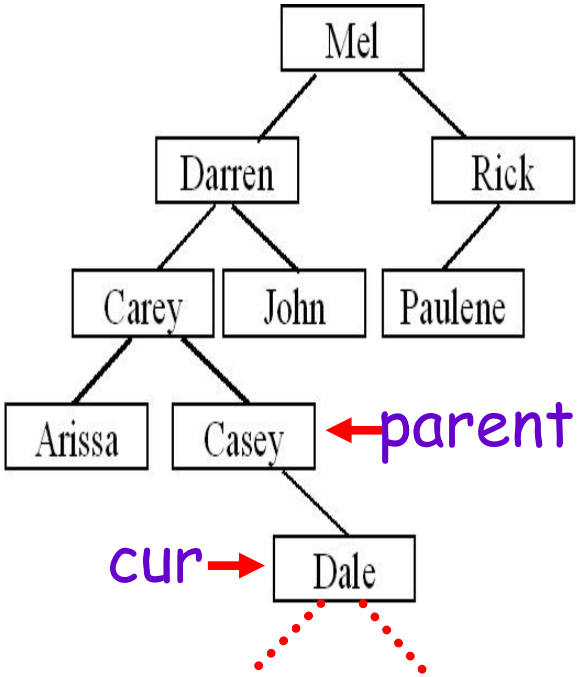
- This algorithm is very similar to our traditional BST searching algorithm except it also computes a parent pointer.
- Every time we move down left or right, we advance the parent pointer as well!
- When we're done with our loop above, we want the parent pointer to point to the node just above the target node we want to delete.
- After we complete the code above, cur points at the node we want to delete, and parent points to the node above it!
- If cur == nullptr, it means we didn't find a node to delete. If parent == nullptr, it means we're deleting the root node.


If we were to delete Arissa, we'd want our parent pointer to point to Carey's node.
Mel
Darren
Rick
parent
Carey
John
Paulene
Arissa
Casey
cur
Dale

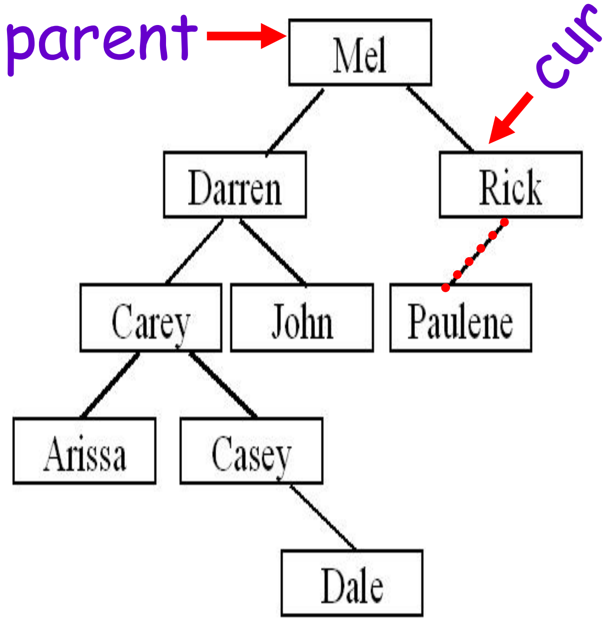
BST Deletion: Step #2

Once we've found our target node (and its parent node) in step #1 we have to delete the target. There are 3 cases.

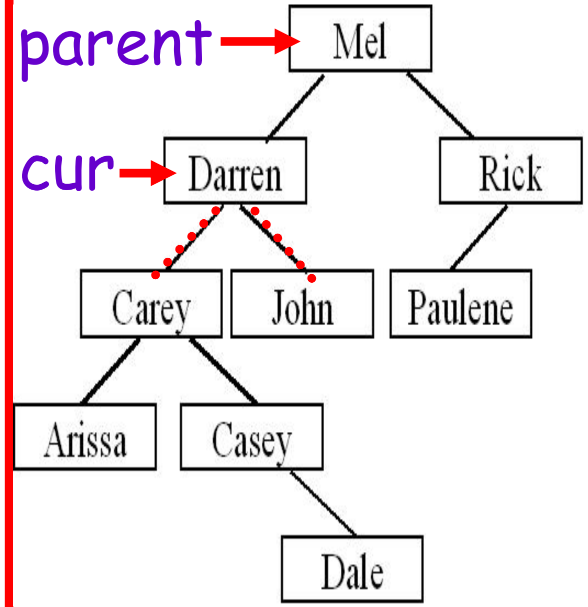
Case 1:
Our node is a leaf.



Case 2:
Our node has one child

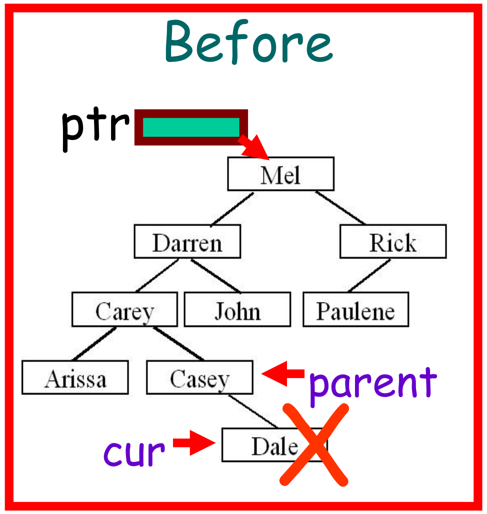


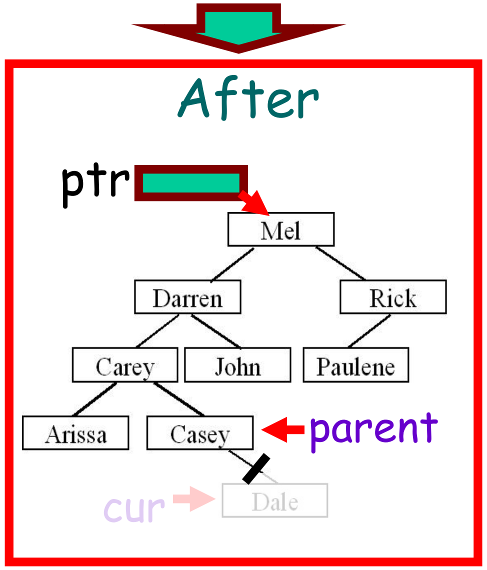
Case 3:
Our node has two children.



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

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





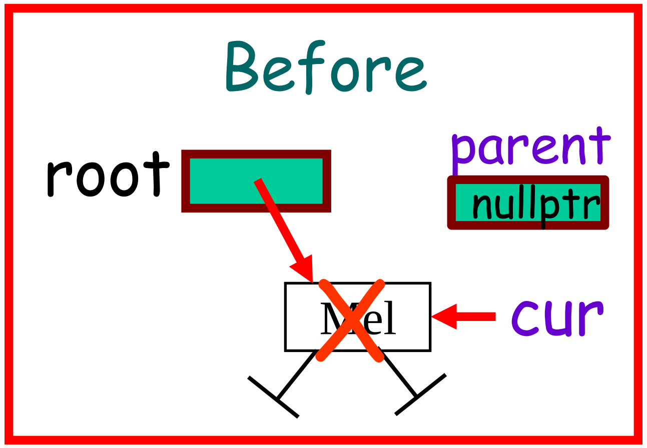
Case 1, Sub-case #1:
The target node is NOT the root node

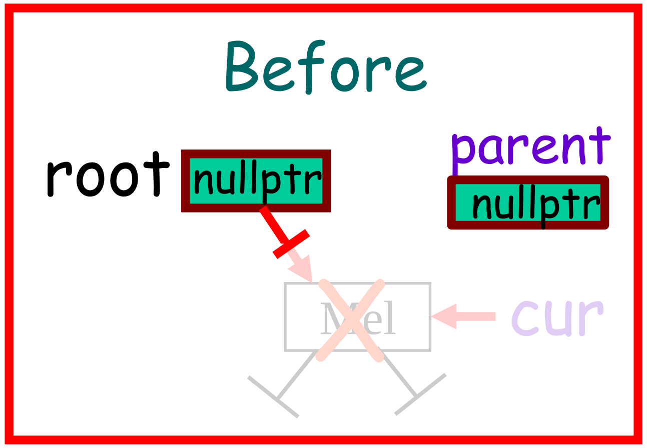
If the node we're deleting is a "leaf" node (with no children), that's case #1. The first sub-case is that this leaf node is NOT the root node (like with Dale - it's a leaf node, but not at the root of the tree). In this case the steps are as follows:

1. Unlink the parent node from the target node by setting the parent's appropriate left or right pointer to nullptr. In this case, our target node (Dale) is our parent node's (Casey's) right child…So we'll set parent->right to nullptr to unlink the parent from the target node.
2. Then we delete the target (cur) node using C++'s delete command.

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

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





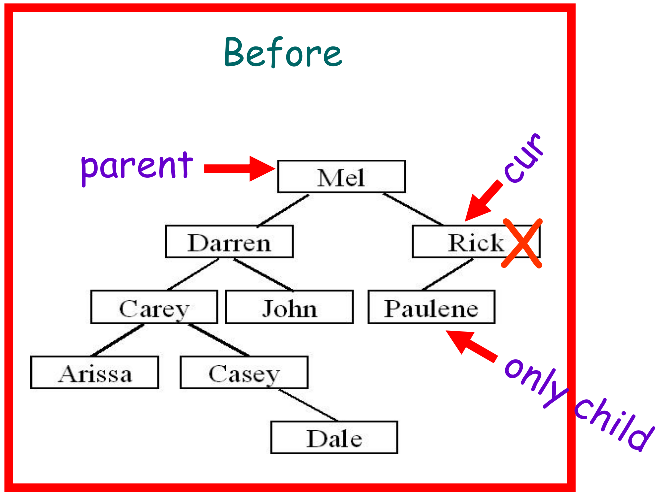
Case 1, Sub-case #2: The target node is the root node

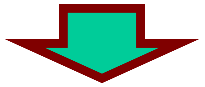
In sub-case #2 we're still deleting a leaf node, but now that leaf node is the only node in the entire tree - so it's at the root of the tree! We know it's this case because parent == nullptr, showing that we didn't move down the tree to find our node. So here's what we do:

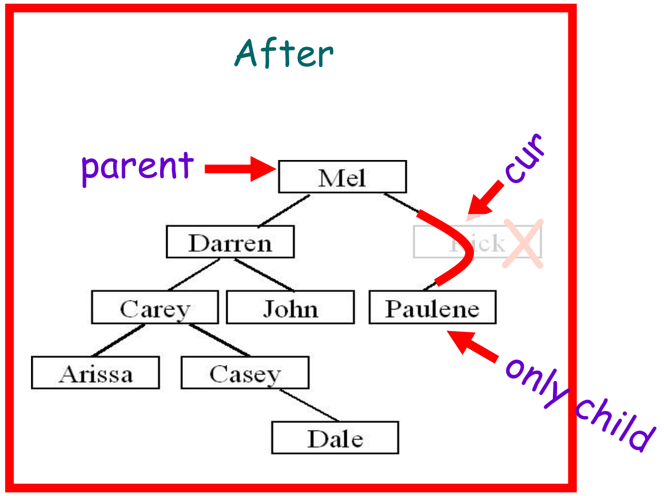
1. Set the root pointer to nullptr.
2. Then delete the target (cur) node.

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

Let's look at case #2 now… Case #2 is when the node we want to delete (Rick) has a SINGLE child of its own (Paulene). Case #2 also has two sub-cases!







Case 2, Sub-case #1:
The target node is NOT the root node

In this case, we want to link the parent node to the only child of the target node that we're deleting. Then we want to delete the target node with the C++ delete command.

So in this example, we'd want to link Mel's right pointer (which points to the target node, Rick) to Rick's left pointer (which points to Rick's only child, Paulene).
In this case that would be: parent->right = cur->left;
Then we'd delete Rick's node.

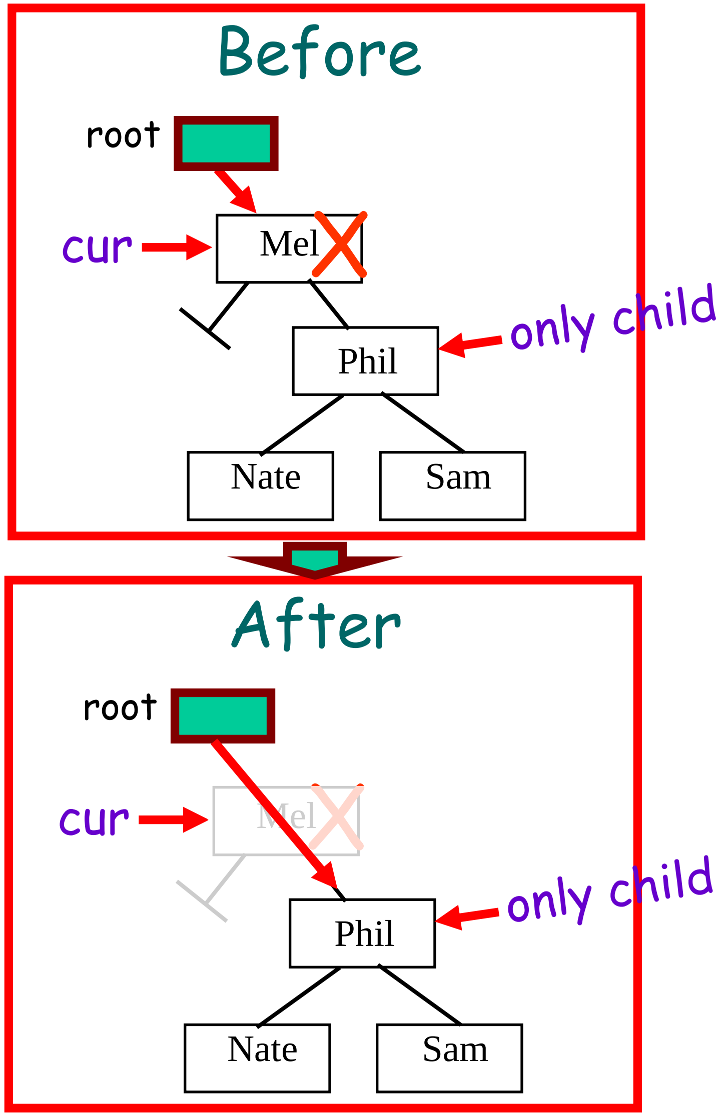
You can easily determine which child of the parent you're deleting, e.g.:

```
 if (parent->left == cur)
    // deleting the left child of the parent
else
   // deleting the right child of the parent
```

And in the same way, we can determine whether the target node as an only child to the right or the left.

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

Ok let's look at Case #2, Sub-case #2 where we're deleting the root node of the tree, and that root node has exactly one child. Like in this case we're deleting Mel, which has just one immediate child, Phil.



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

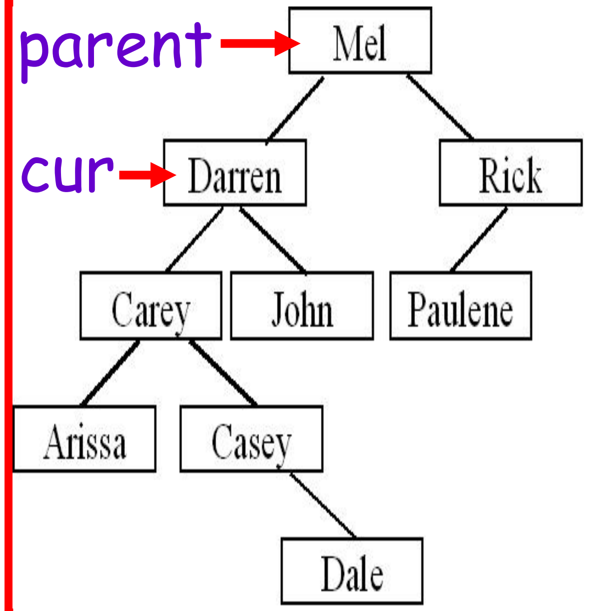
In this case, the node we want to delete is the root node, and it has a single child. To delete the root node, we have to change the root pointer to point around the target to the target node's only child (Phil).

Once we've done that, we then use the C++ delete command to delete the target node.

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

Let's look at case #3 now. In this case, we are trying to delete a node that currently has two children. There's only one subcase here to deal with.

Case 3:
Our node has two children.



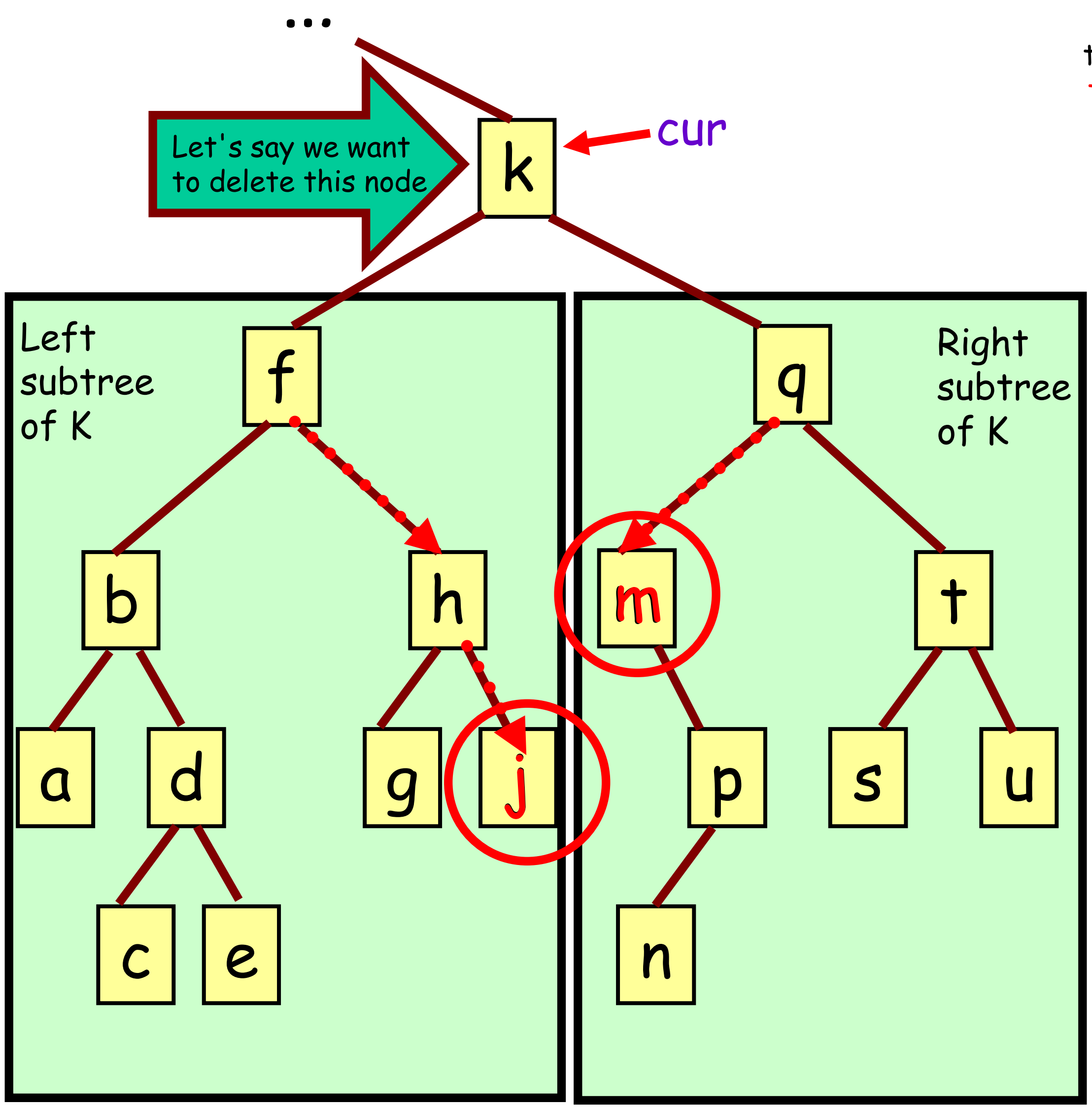
In this case, let's assume we're trying to delete Darren from our tree. Darren has two children, Carey and John, making this Case #3.

So how do we 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 because this might violate our BST's ordering properties!

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

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



Well there's a trick! We don't actually delete the target node itself! Instead, we replace the value in the target node with a value from another node... and then we delete that other node instead!

Let's say our goal is to delete some target node k (see example to the left). Node k could be the root node, or some node in the middle of the tree. It doesn't matter.

The replacement value to be copied into node k MUST come from ONLY one of two places:

1. K's left subtree's largest-valued node
2. K's right subtree's smallest-valued node

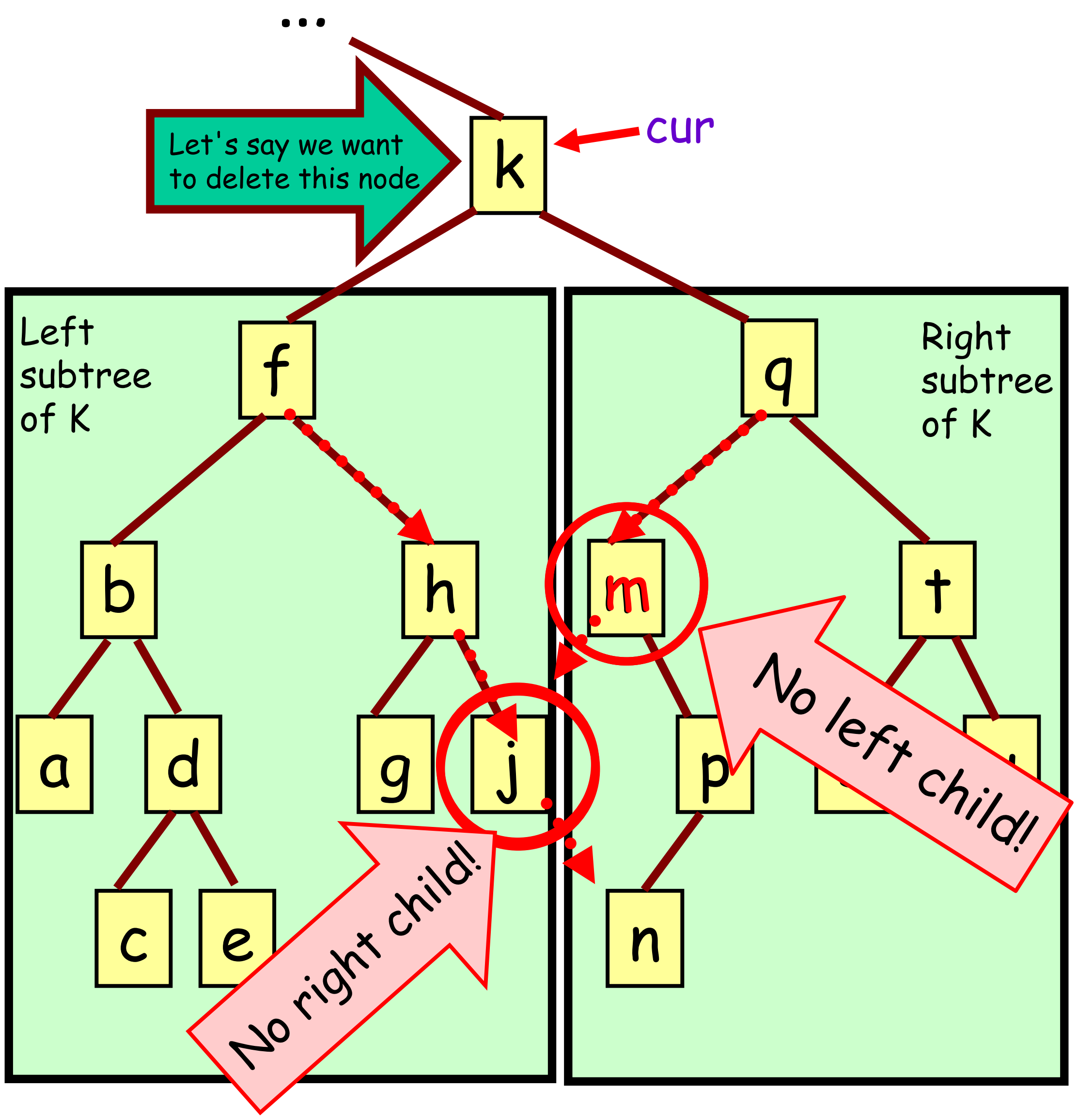
To find the largest valued node in k's left subtree, we basically set a pointer to k's left child then keep following the right pointer of each node (zero or more times) until we can't go any further:

```
ptr = cur->left;
while (ptr->right != nullptr)
  ptr = ptr->right;
// ptr points at the largest valued node on the right
```

We could do use a similar algorithm with k's right child if we wanted to (going all the way to the left to find the smallest value in the right subtree). Either choice is valid, and you can pick one way to implement your code. There's no right or wrong way.

Once we find the new target node (either j or m in this case), then we just copy that node's value up into our root node and then delete that other target node instead!

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



Notice that the biggest node in the left subtree, by definition, can't have a right child (because if it did, that would be the biggest value in the left subtree and we would have traversed until we reached it!).

So that means that the biggest node in the left subtree has either a left child, or no children at all.

In the example to the left, node j is the biggest value in the left subtree of k, and it's a leaf node with no children.

Similarly, the smallest node in the right subtree, by definition, can't have a left child (because if it did, that would be the smallest value in the right subtree and we would have traversed until we reached it).

So that means that the smallest node in the right subtree has either a right child, or no children at all.

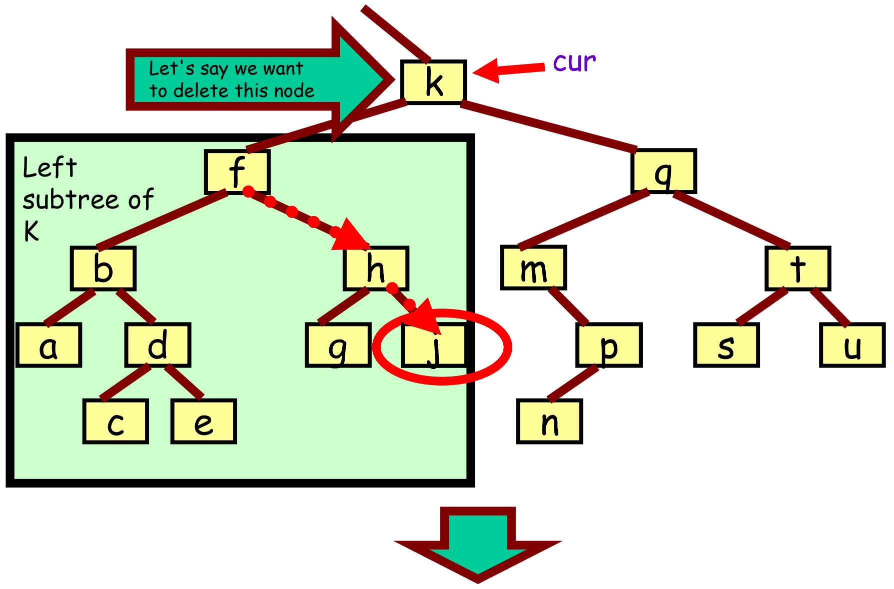
In the example to the left, node m is the smallest value in the right subtree of k, and it has a single right child (p).

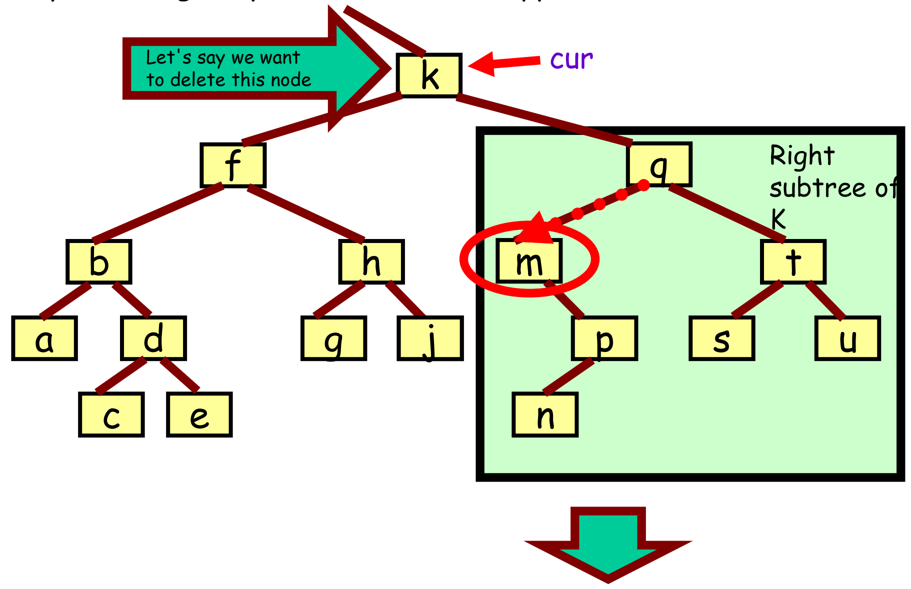
Once we find this new target node, we do the following:

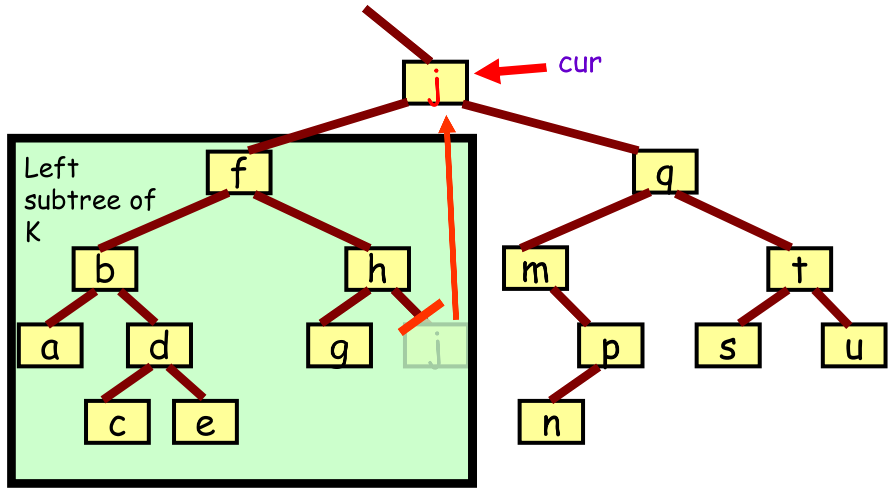
1. Copy its value (either j or p) into the original target node: cur->val = new_target->val;
2. Delete the new target node, which is easy, because it's either Case #1 or Case #2 from a few slides ago.

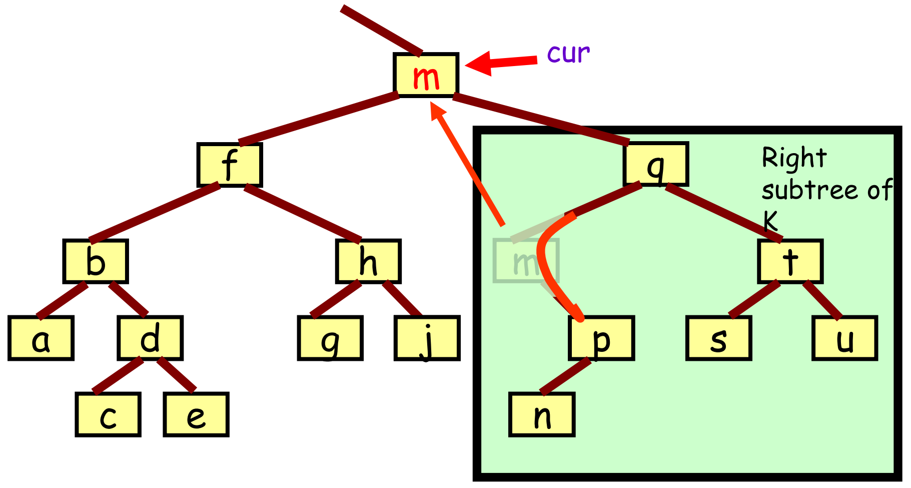
Step #2, Case #3 - Our Target Node has Two Children

Ok, let's see the before and after for our two possible options – again, you can use either approach.

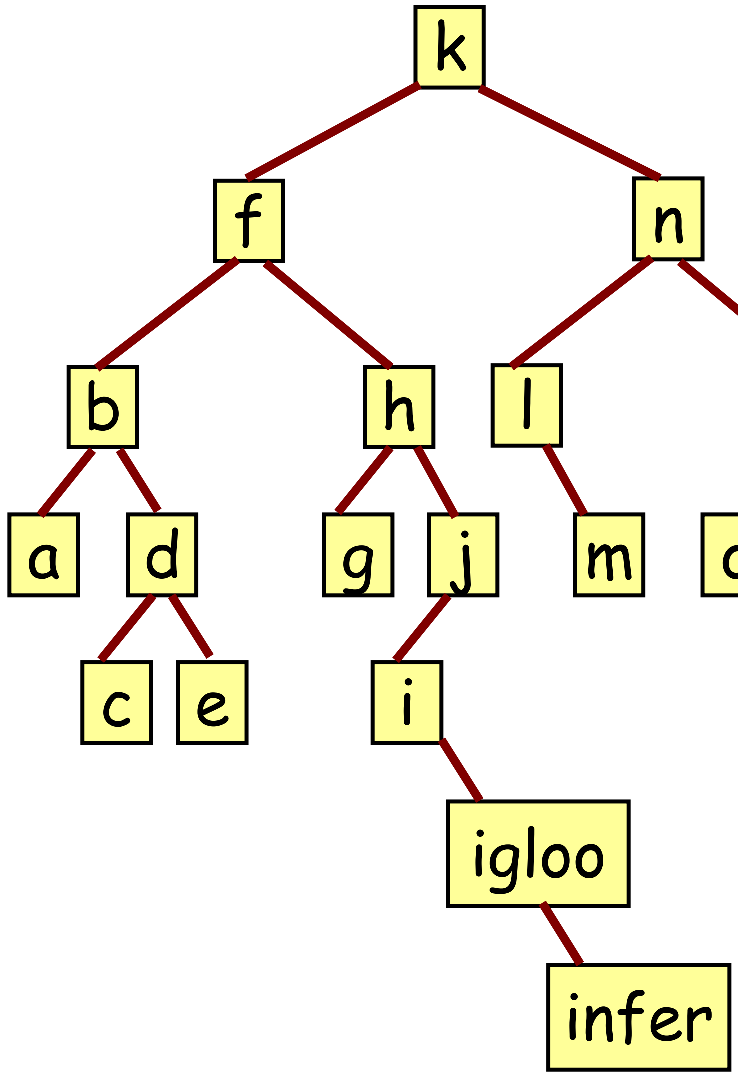








Deletion Exercise



1. Explain how you would go about deleting node k.
2. Explain how you would go about deleting node e.
3. Explain how you would go about deleting node i.

Answers:

1. To delete node k, you could pick a replacement node of j or l, copy that value up to node k, and then delete that node. To delete node j (Case #2), you'd link h's right pointer to node i. To delete node l (Case #2), you'd link n's left pointer to node m. Finally, you'd use the C++ delete command to delete either node j or l.
2. To delete node e (Case #1), you'd simply set d's right pointer to nullptr, then delete node e.
3. To delete node i, (Case #2), you'd simply link j's left pointer to igloo, then delete node i with C++'s delete command.

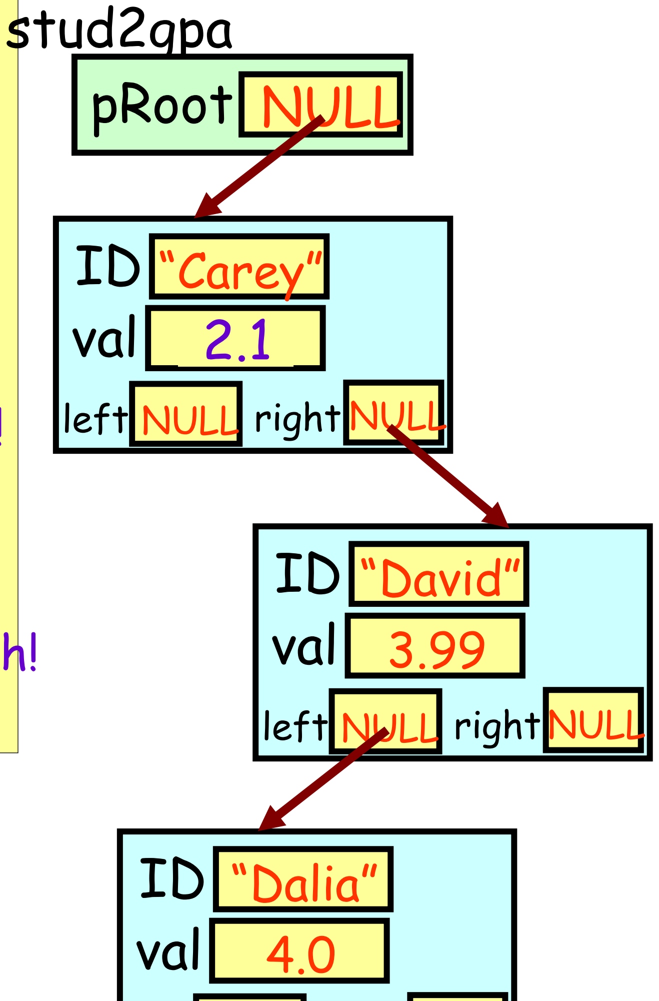
Where are Binary Search Trees Used?

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

}
```

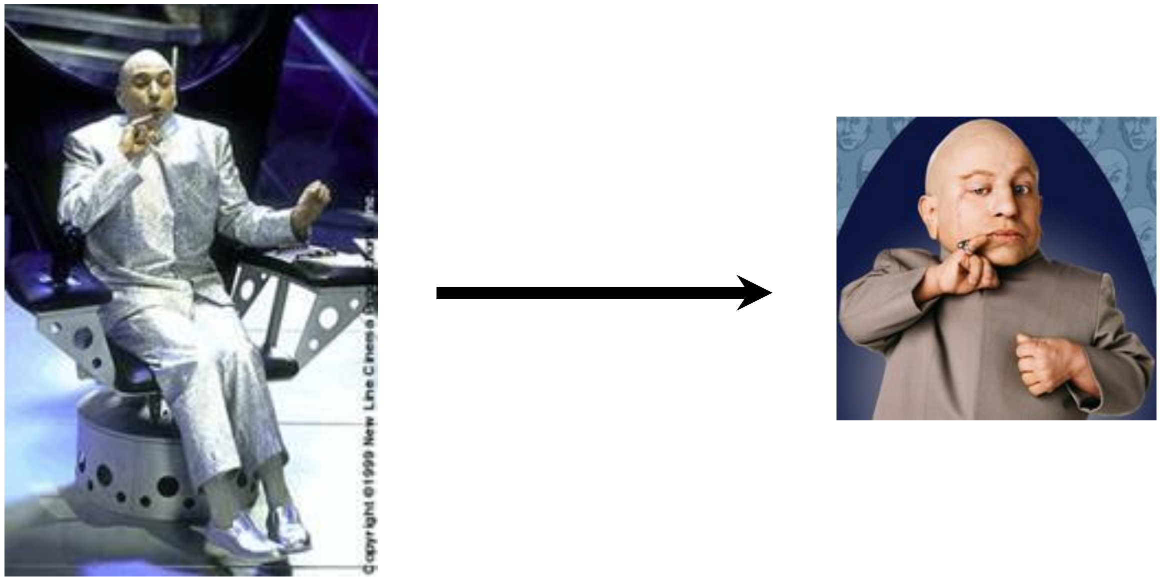
The STL set and map use binary search trees (a special balanced kind) to enable fast searching.

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

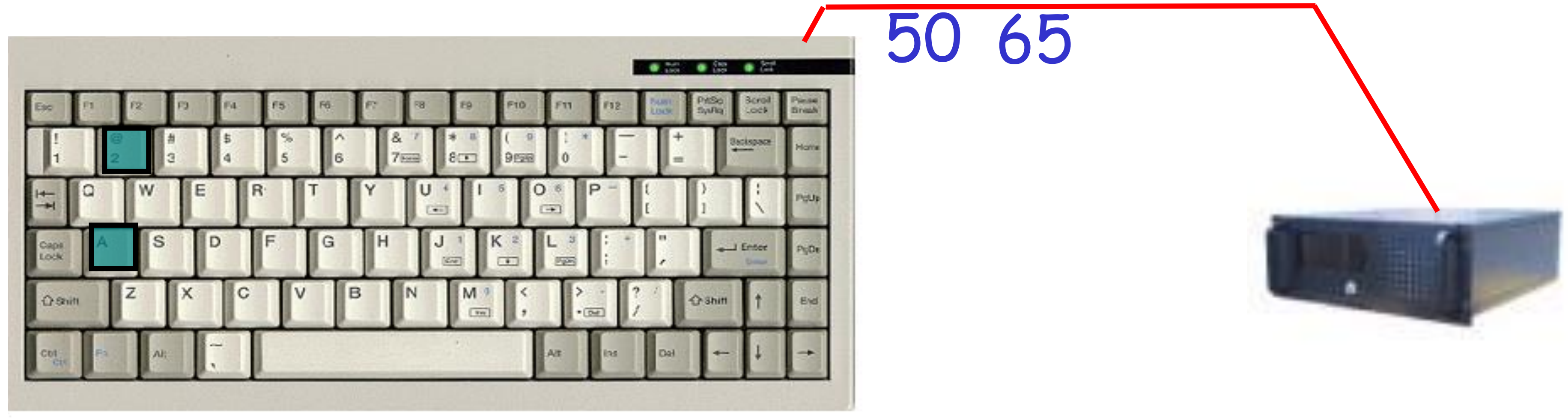
Before we actually cover Huffman Encoding, we need to learn a few things…

Remember the ASCII code?

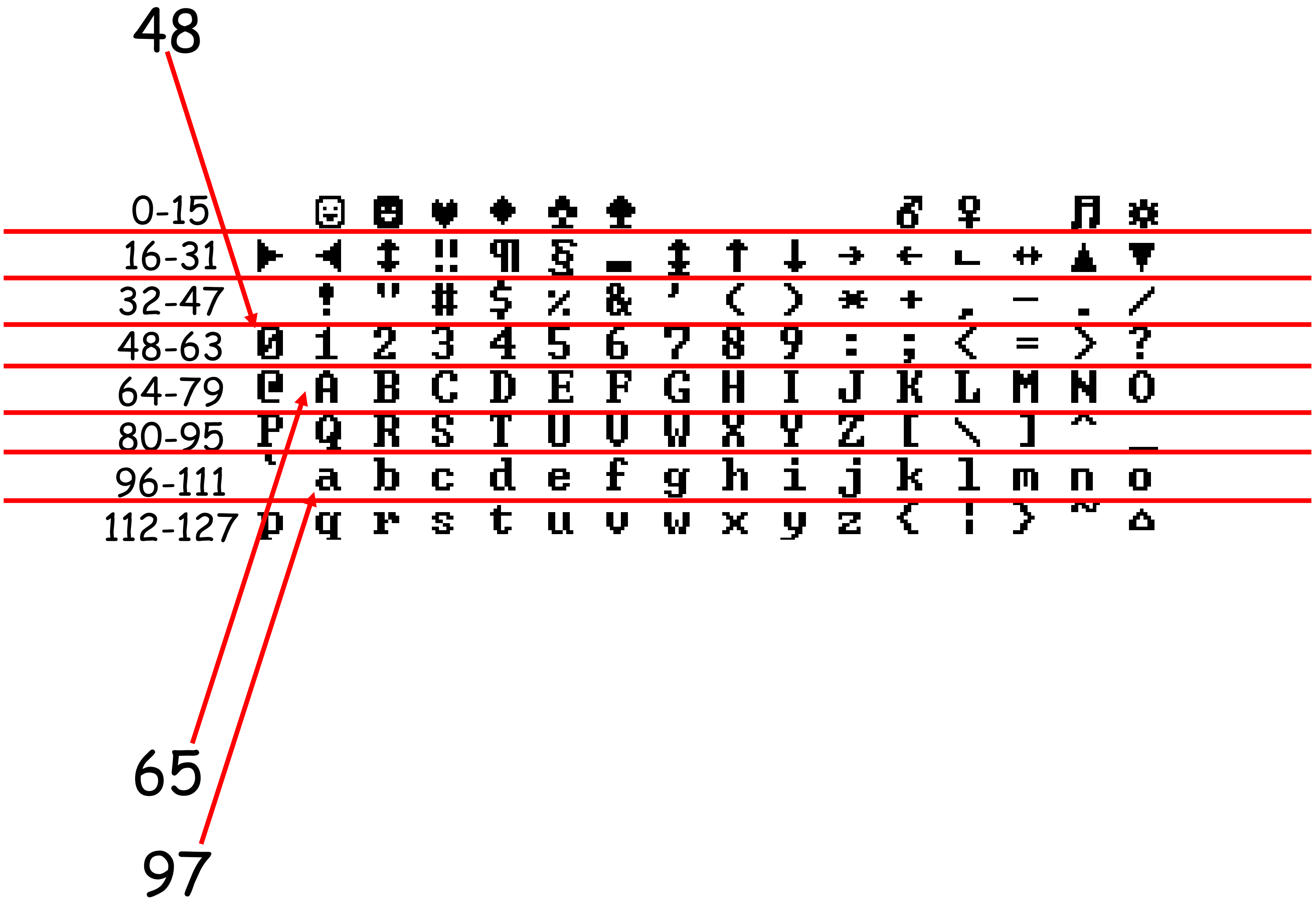
ASCII

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!



The ASCII Chart



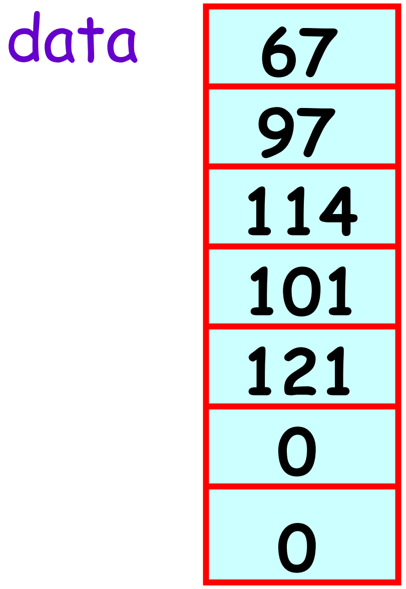
0-15		☺	☻	♥	♦	♣	♠					♂	♀		♫	☼
16-31	►	◄	↕	‼	¶	§	▬	↨	↑	↓	→	←	∟	↔	▲	▼
32-47		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
48-63	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64-79	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80-95	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
96-111	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112-127	p	q	r	s	t	u	v	w	x	y	z	{	¦	}	~	⌂

Computer Memory and Files

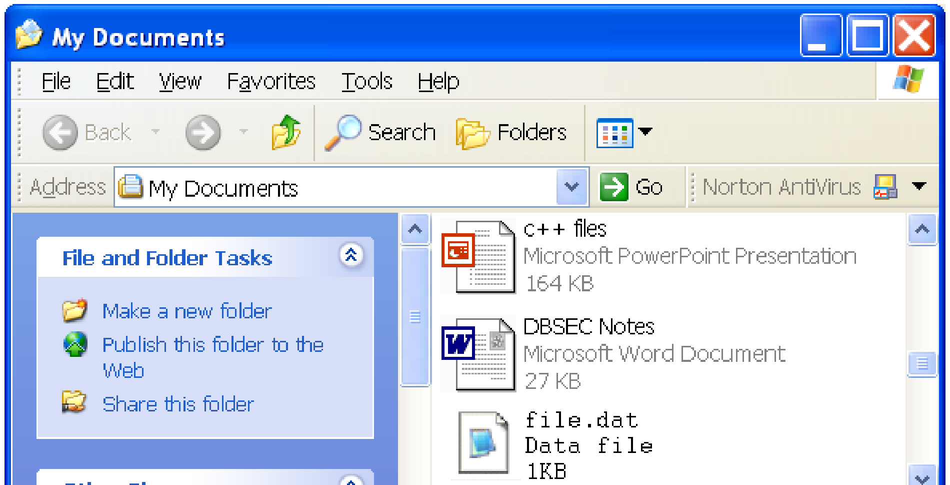
So basically, characters are stored in the computer's memory as numbers…

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

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



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



c++ files
Microsoft PowerPoint Presentation
164 KB

DBSEC Notes
Microsoft Word Document
27 KB

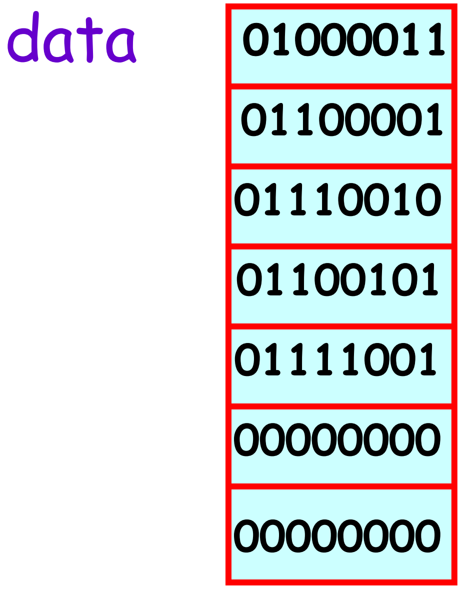
file.dat
Data file
1KB

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();
}
```



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**

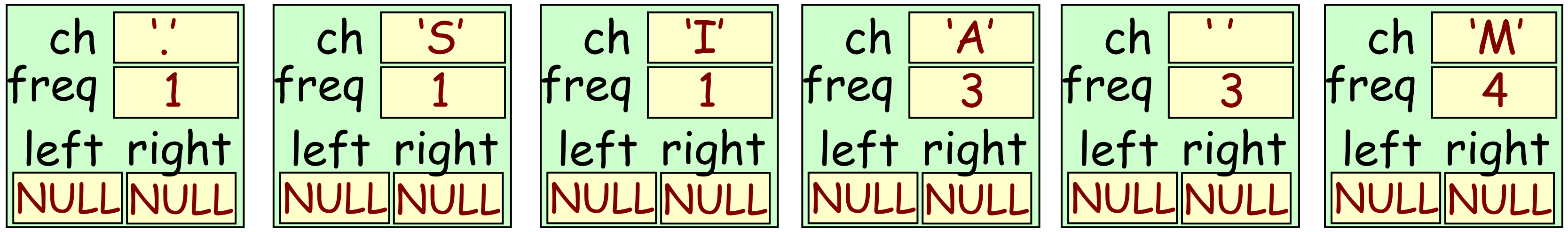
You can do this by simply counting how many time each letter or punctuation mark occurs in the original file, as you see to the left.

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!

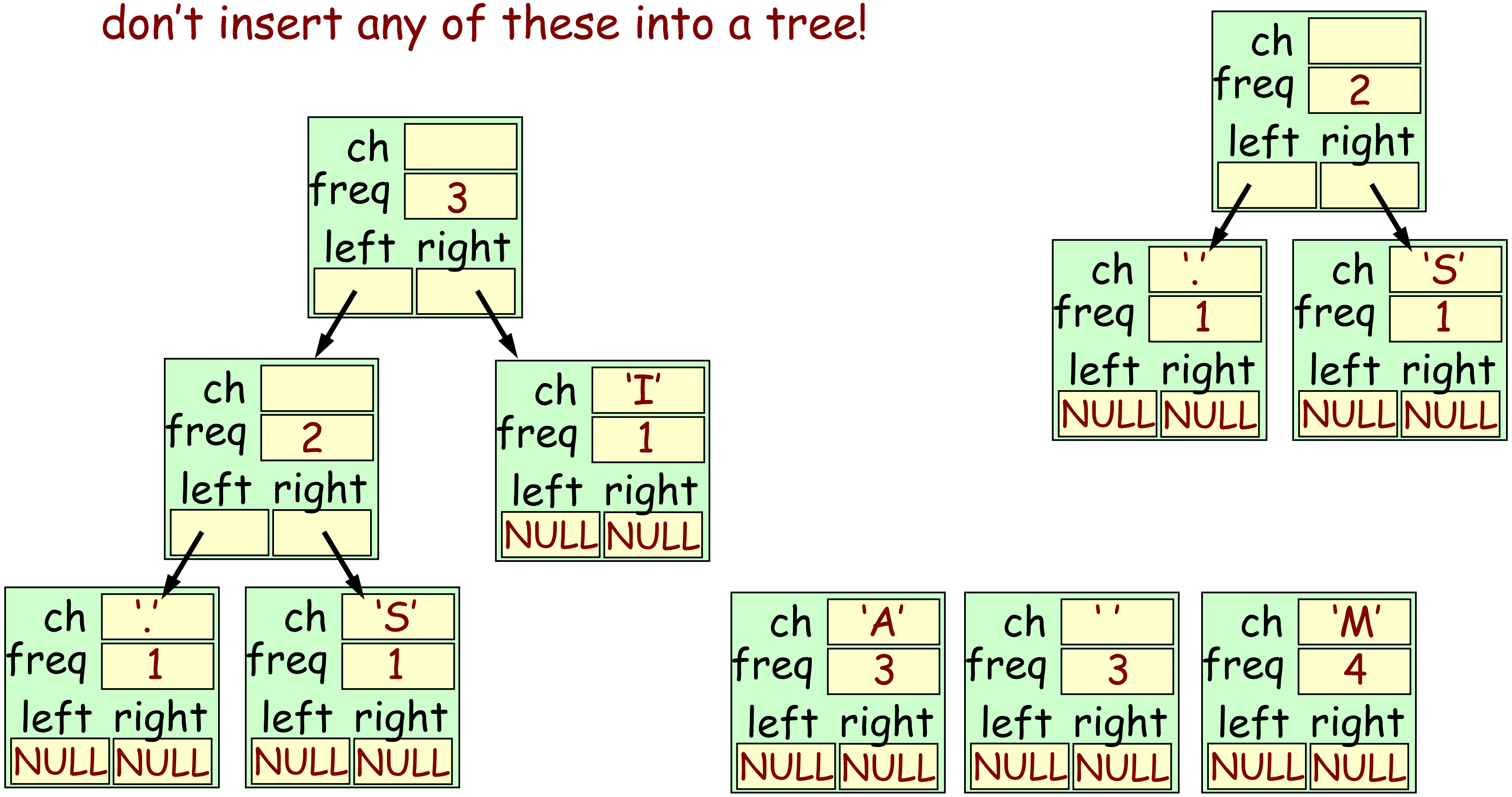
B. While we have more than one node left:
   1. Find the two nodes with lowest freqs.
   2. Create a new parent node.
   3. Link the parent to each of the children.
   4. Set the parent's total frequency equal to the sum of its children's frequencies.
   5. Place the new parent node in our grouping.


ch '.' freq 1 left right NULL NULL
ch 'S' freq 1 left right NULL NULL
ch 'I' freq 1 left right NULL NULL
ch 'A' freq 3 left right NULL NULL
ch ' ' freq 3 left right NULL NULL
ch 'M' freq 4 left right NULL NULL

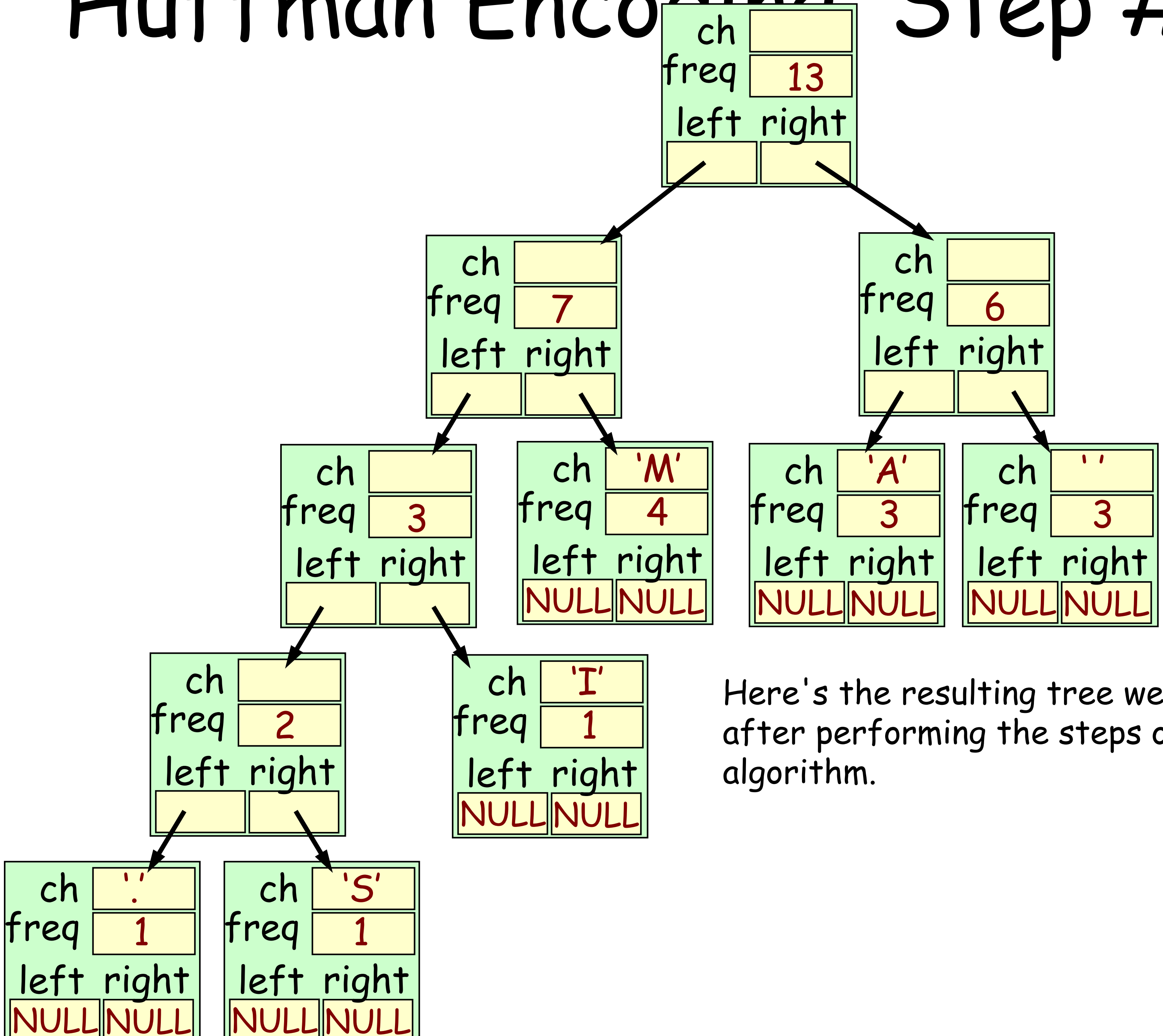
Huffman Encoding: Step #2

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



Here's the resulting tree we'd get after performing the steps of the algorithm.

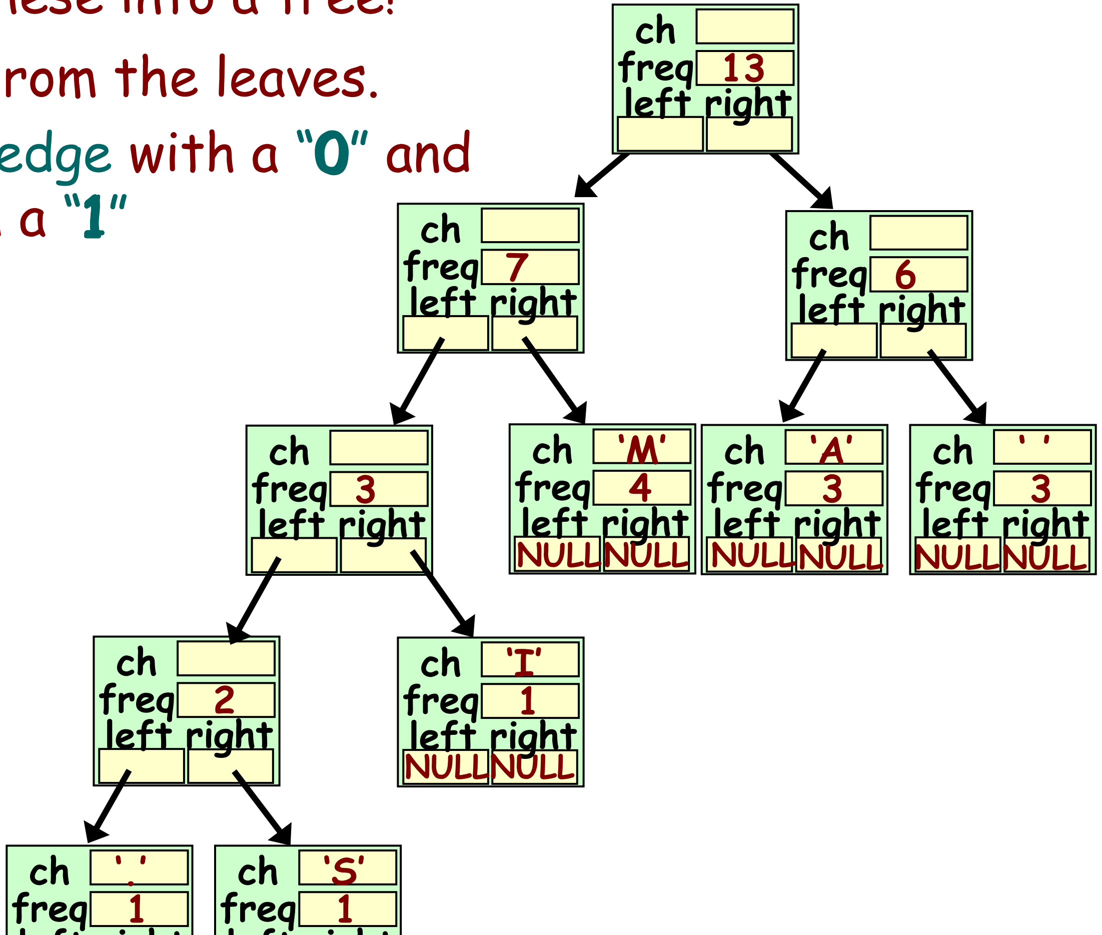
Huffman Encoding: Step #2

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. Let's imagine that each left branch is labeled "0", and each right branch is labeled "1".

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 reach the character of interest.

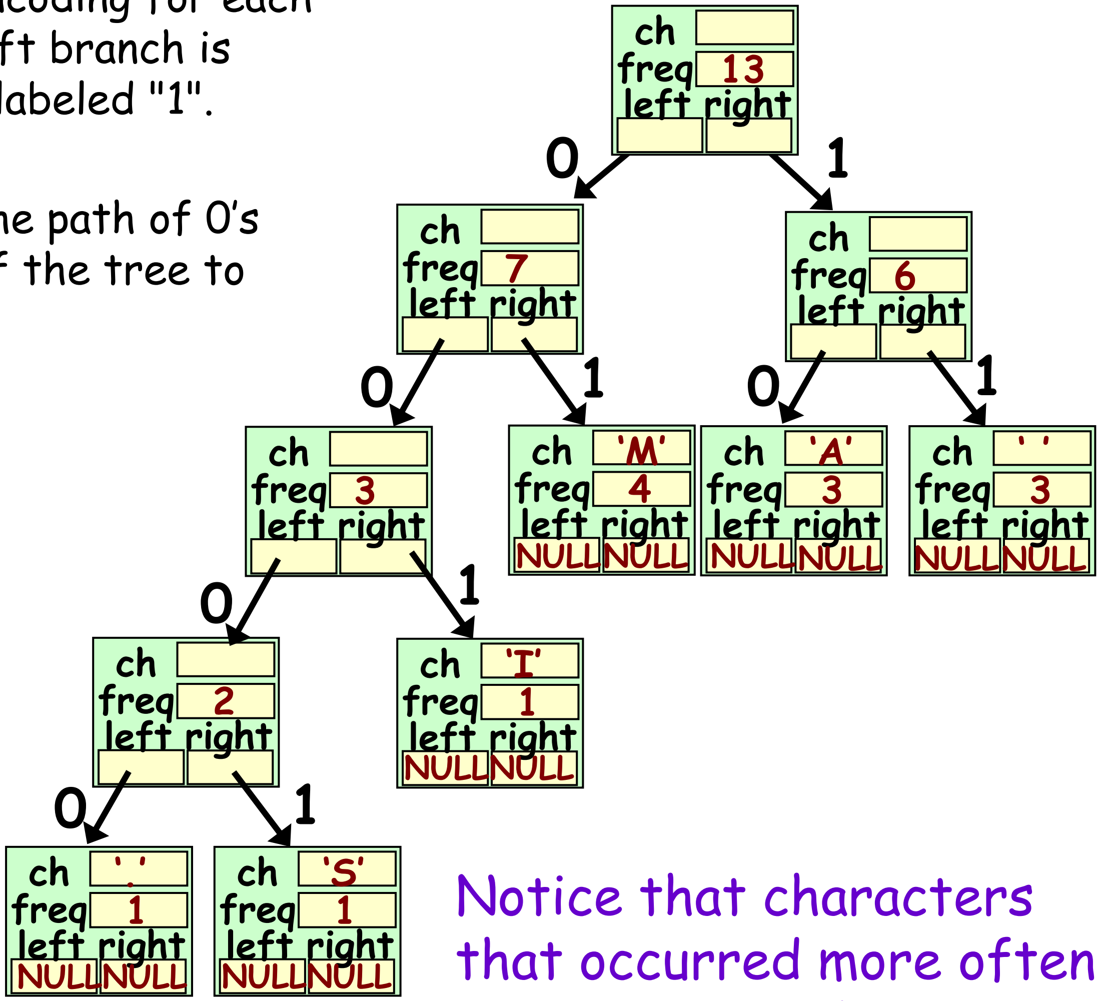
For example:

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!

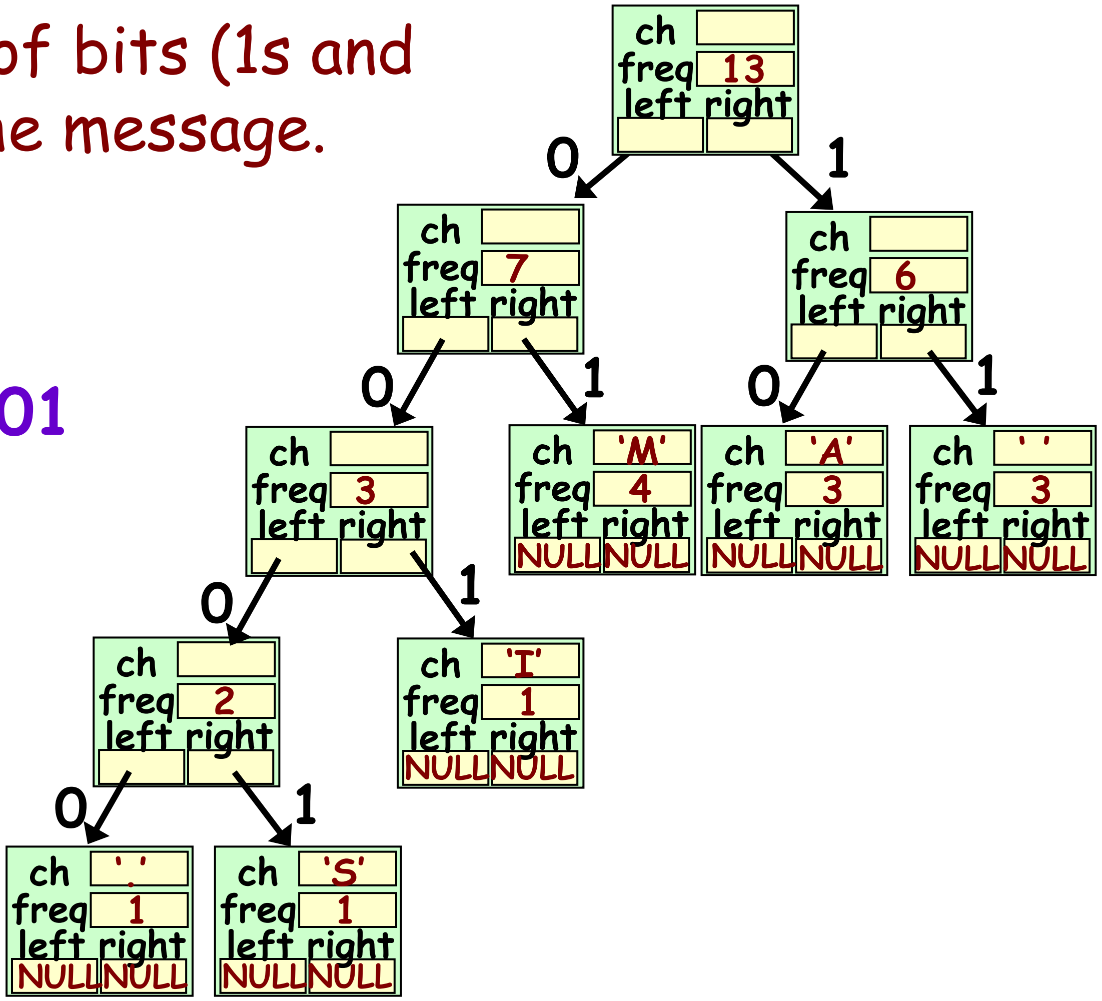
Huffman Encoding: Step #3

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

```
001 1110
01110001
10011101
10010000
```

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

originalfile.dat

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

Our original file is 13 bytes or 104 bits long!

We saved over 69%!

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

But we have a problem! 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 (e.g., A == "10" and I == "001", etc)…

Now clearly this adds some overhead to our file…

But usually there's a pretty big savings anyway!

Decoding Algorithm

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:

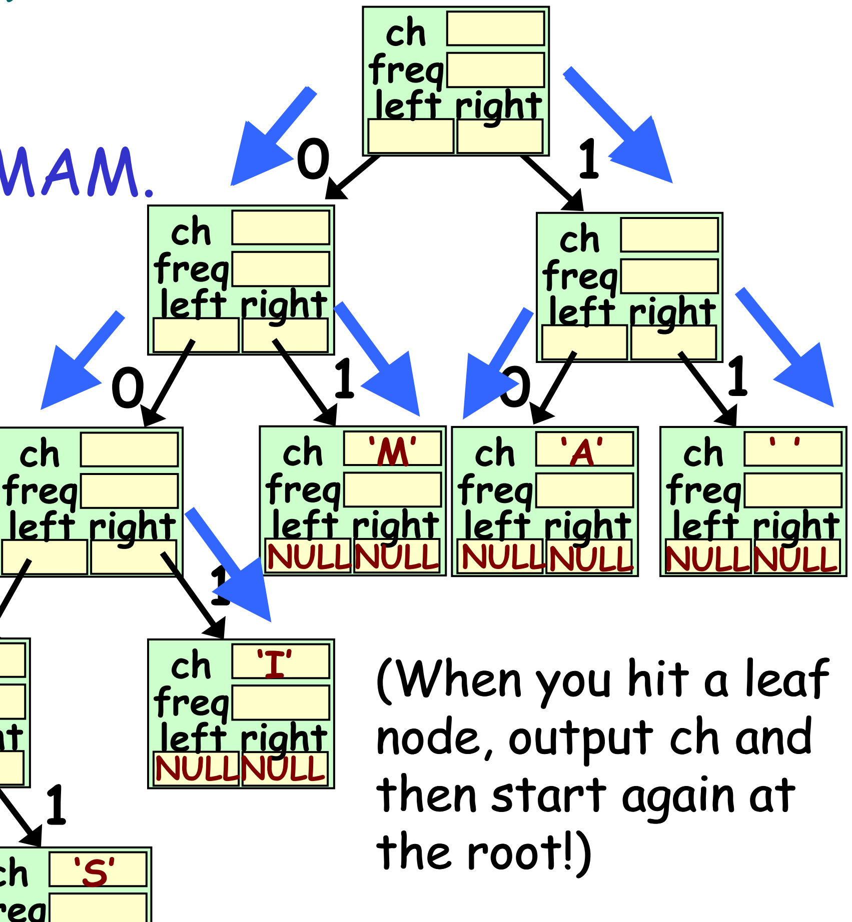
001 1110
01110001
10011101
10010000

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

Right! We get an unbalanced tree!

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

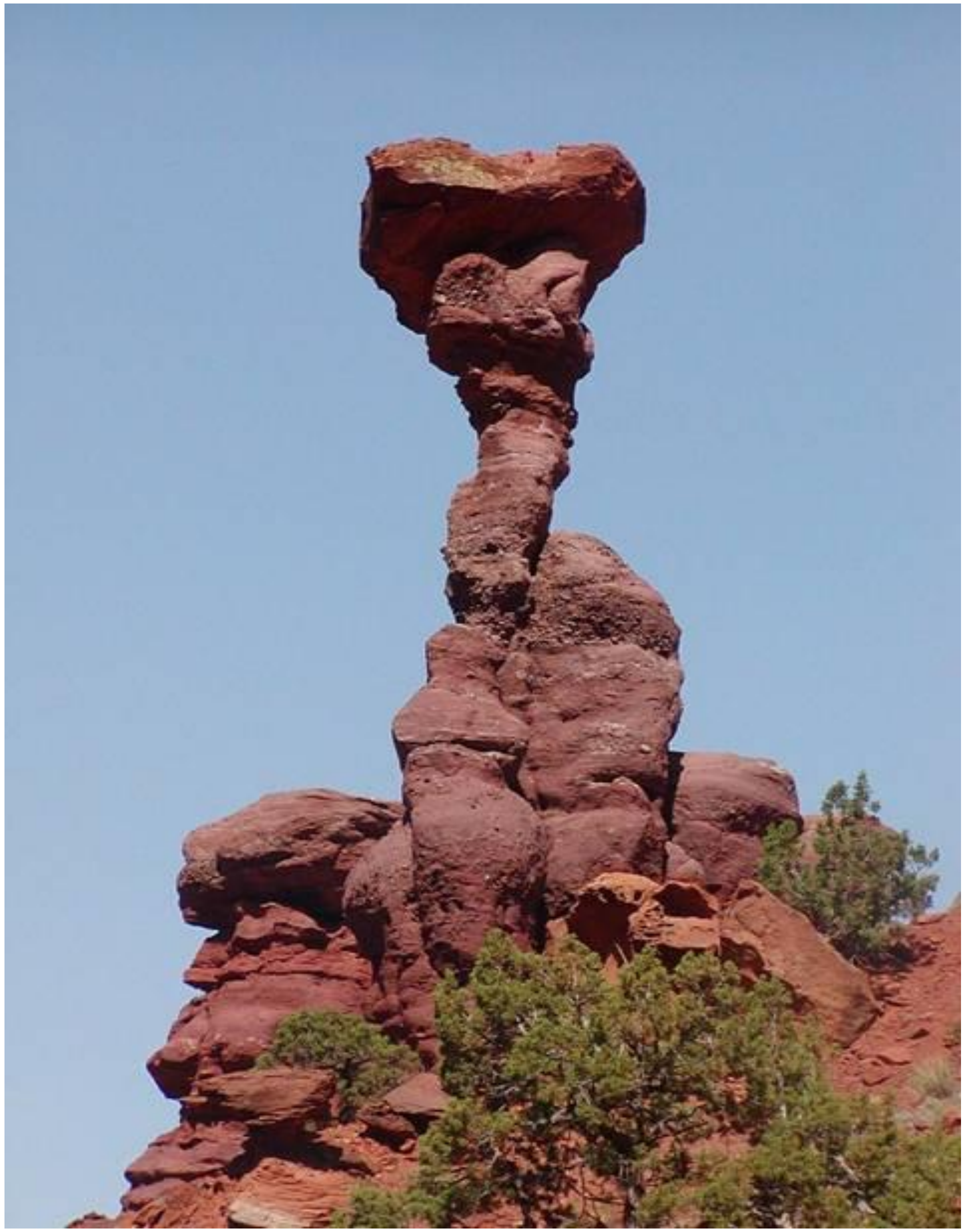
O(N)... YUCK!

Balanced Search Trees

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

It'd be nice if we could come up with an improved BST ADT that *always maintains its balance*.

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



Balanced Search Trees

Well, guess what?

CS nerds have come to the rescue!

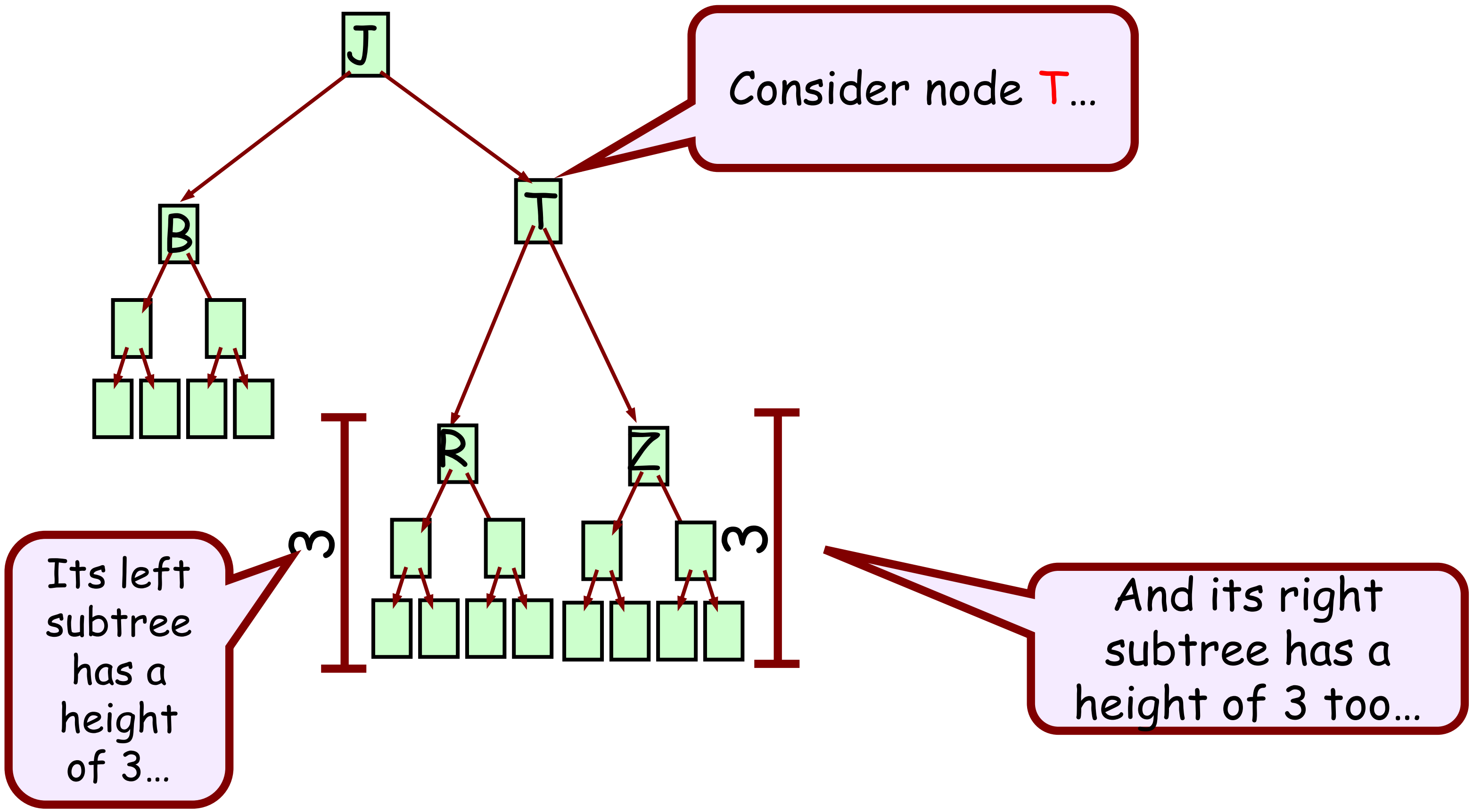
They've invented numerous improved binary search tree ADTs like 2-3 Trees, Red-Black Trees, and AVL Trees.

These BST variations work (mostly) just like a regular binary search tree…

but every time you add/delete a value, they automatically shift the nodes around so the tree is balanced!

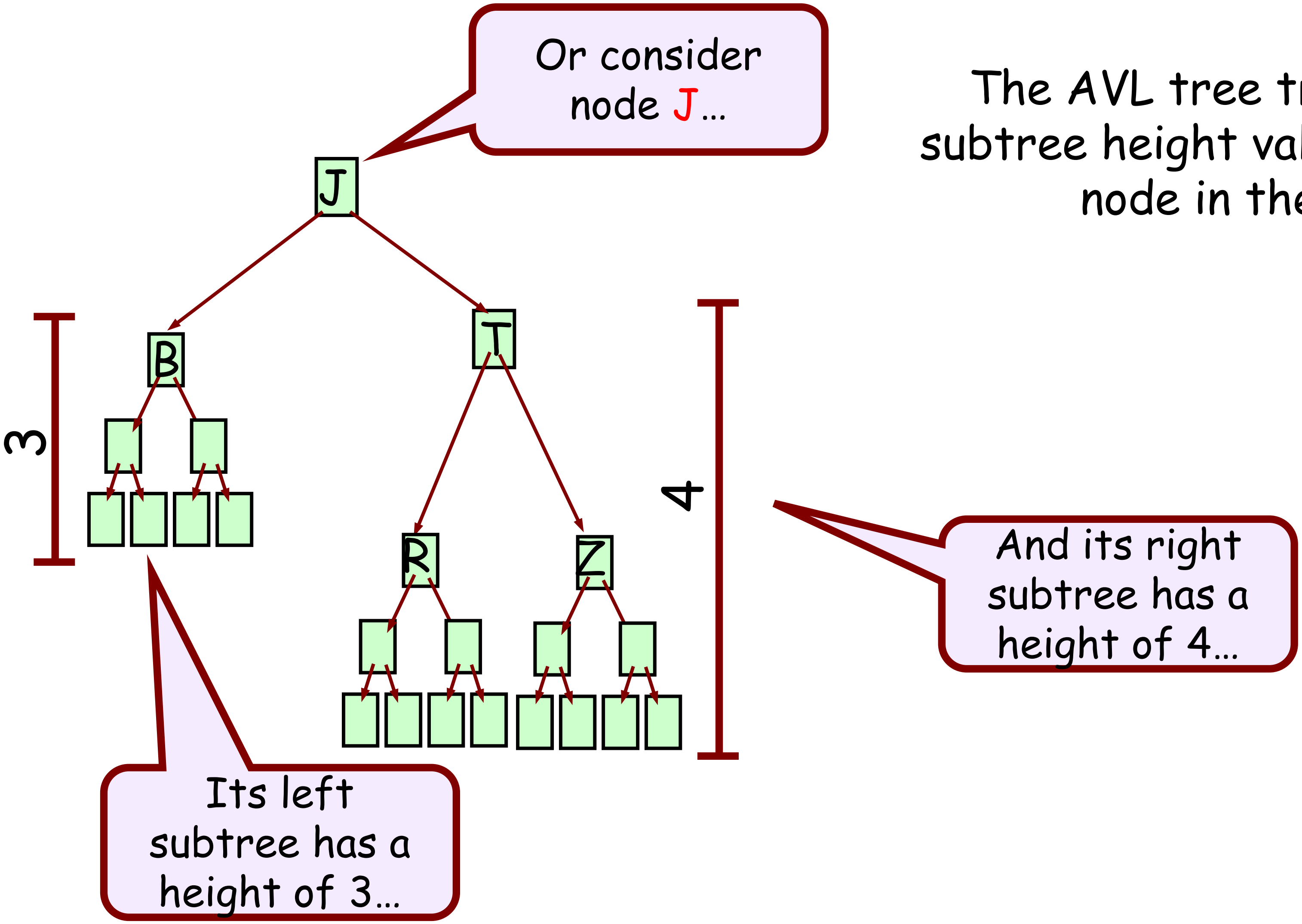
Balancing a Tree On Insertion

For example, the AVL Tree tracks the height of ALL subtrees in the BST.


J
B
T
R
Z
3
3
Consider node T…
Its left subtree has a height of 3…
And its right subtree has a height of 3 too…

Balancing a Tree On Insertion

For example, the AVL Tree tracks the height of ALL subtrees in the BST.

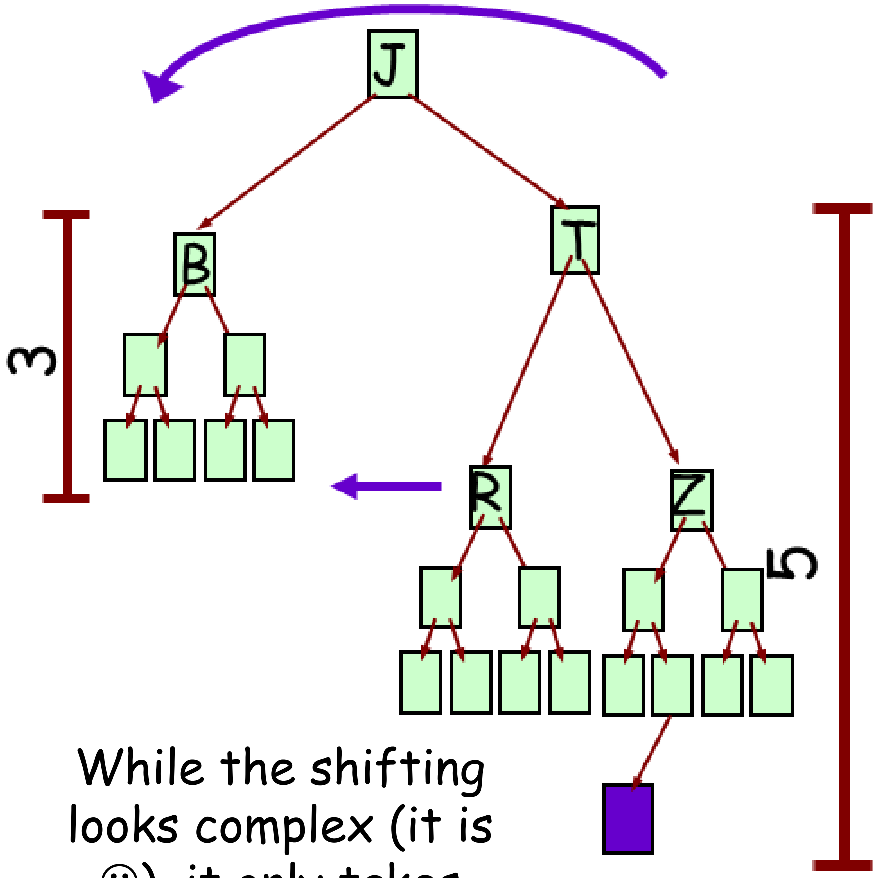

Or consider node J…
J
B
T
R
Z
3
4
Its left subtree has a height of 3…
And its right subtree has a height of 4…

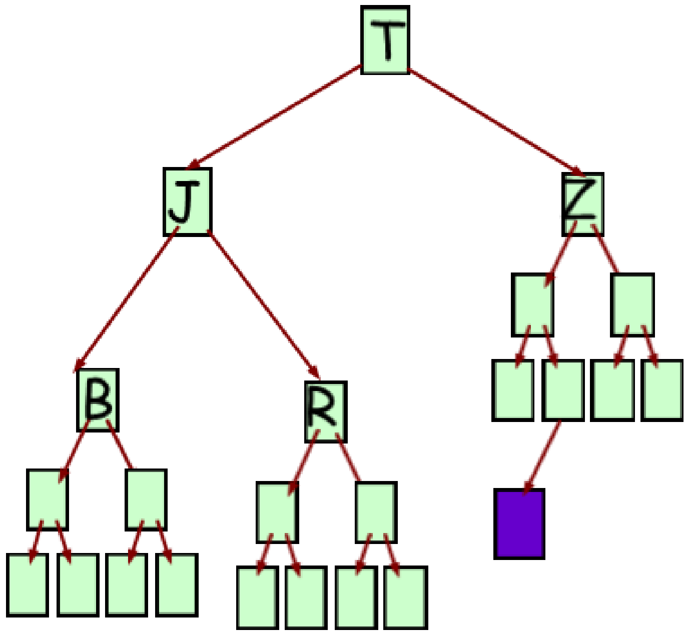
The AVL tree tracks these subtree height values for every node in the tree!

For example, the AVL Tree tracks the height of ALL subtrees in the BST.

After an insertion/deletion (see left), if the height of the subtrees under any node is different by more than one level…

Then the AVL algorithm shifts the nodes around to maintain balance (see right).





While the shifting looks complex (it is ☺), it only takes O(log n) time!

So with just a little extra work, the tree is always balanced and can always be searched in log n time!

Balanced Search Trees

You don't need to know the gory details of any of these balanced BSTs for your final or projects.

Just remember, that balanced BSTs are always O(log n) for insertion and deletion.

And if you're ever in a job/internship interview and are asked a BST question…

Always make sure to ask the interviewer if you may assume the BST is balanced!

It could make or break your interview!