Binary Trees: Deletion

Computer Science Department
University of Central Florida

COP 3502 – Computer Science I

Binary Trees: Deletion

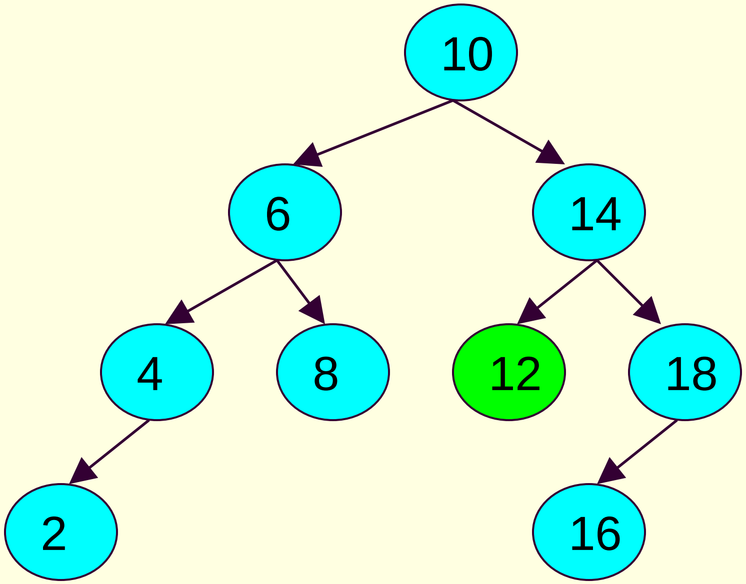
- Deletion From a Binary Search Tree
  - Deleting nodes from a BST requires some thought
  - There are 3 possible cases
    - And we deal with each in a different fashion
  - <u>The 3 cases are:</u>

1) Deleting of a leaf node
2) Deleting a node with one child
3) Deleting a node with two children

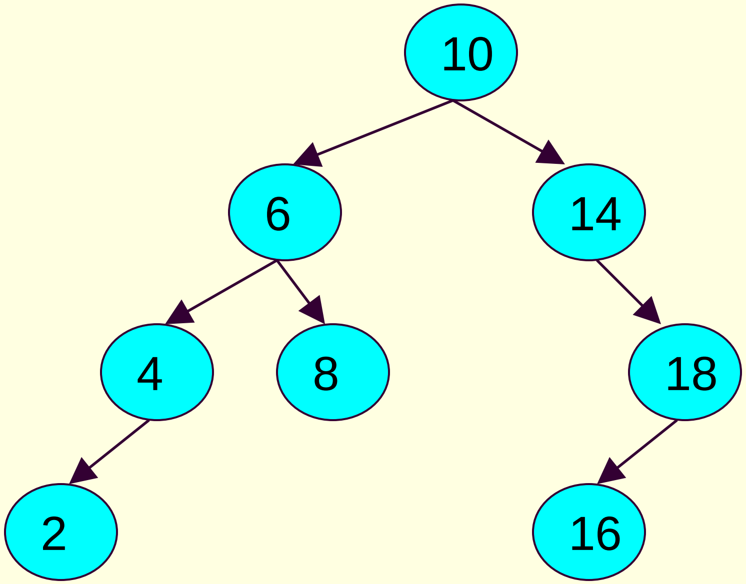
  - We examine each case separately

Binary Trees: Deletion

- Deleting a Leaf Node
  - This one is pretty easy



Initial BST with node 12 marked for deletion



BST after deletion of node 12

Binary Trees: Deletion

- Deleting a Leaf Node
  - This one is pretty easy
  - We start by identifying the parent of the node we wish to delete
    - **Which we actually do in ALL three cases**
  - We then free that node, accessing it via parent:
    - `free(parent->left)` or
    - `free(parent->right)`
  - Now we need to simply update the parent's left or right pointer, signifying that the parent no longer has that child

Binary Trees: Deletion

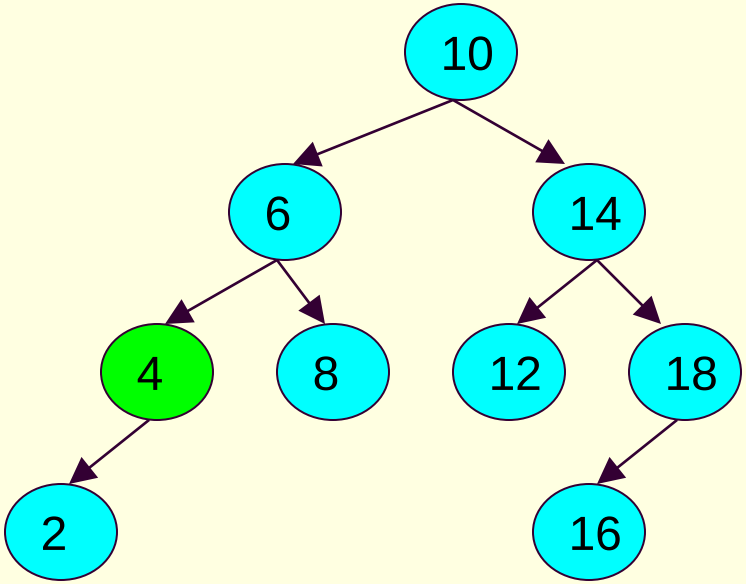
- Deleting a Leaf Node
  - This one is pretty easy
  - We start by identifying the parent of the node we wish to delete
    - <u>**Which we actually do in ALL three cases**</u>
  - Just set the appropriate node to NULL:
    - `parent->left = NULL` or
    - `parent->right = NULL`
  - So now instead of pointing to the toBeDeleted node
    - The parent simply points to `NULL`

Binary Trees: Deletion

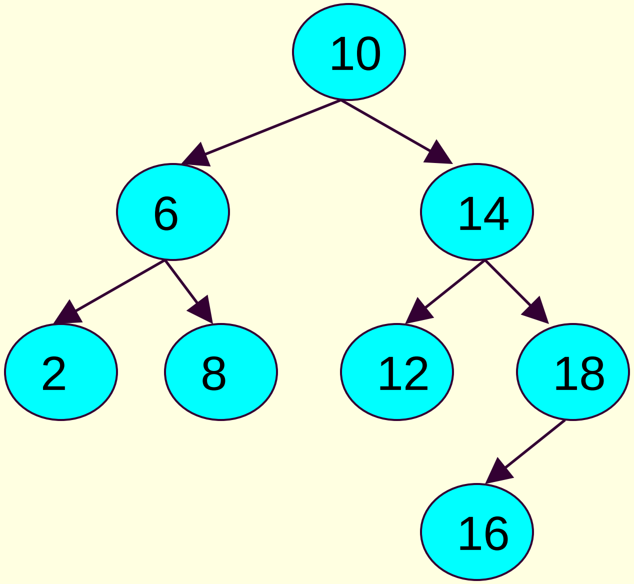
- Deleting a Node with One Child
  - This one is also not too complicated
    - But does require more thought than deleting a leaf node
  - Again, we start by finding the parent
    - meaning, the parent of the node we want to delete
  - The parent's pointer to the node is changed to now point to the deleted node's child
  - This has the effect of lifting up the deleted nodes child by one level in the tree
    - All great-great-…-great-grandchildren will lose one 'great' from their kinship designations

Binary Trees: Deletion

- Deleting a Node with One Child
  - `parent->left = parent->left->left;`



Initial BST with node 4 marked for deletion



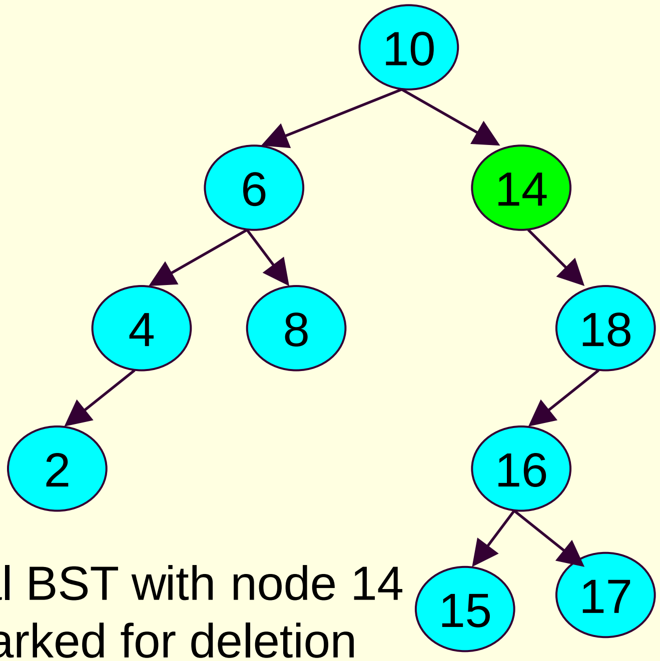
BST after deletion of node 4. Node 2 has taken the place of the deleted node

Binary Trees: Deletion

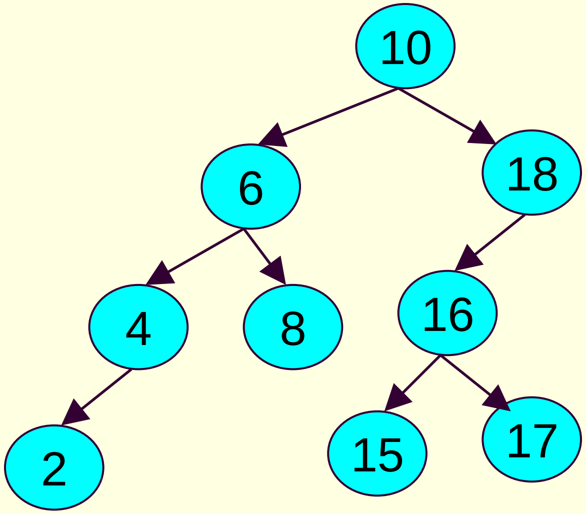
- Deleting a Node with One Child
  - The previous example illustrated when we delete a **left node that has a left child**
  - Notice that it makes no difference whether the only child is a left child or a right child
  - The next example illustrated when we delete a **right node that has a right child**
    - Again, the deletion simply lifts up the subtree of the deleted node by one level
  - The other possibilities is a left node that has a right child or a right node that has a left child

Binary Trees: Deletion

- Deleting a Node with One Child
  - `parent->right = parent->right->right;`

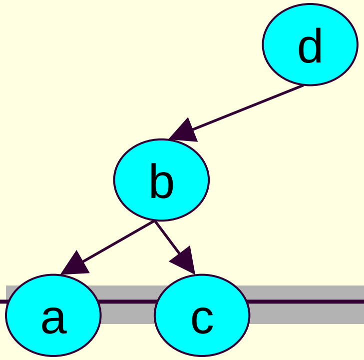


Initial BST with node 14 marked for deletion



BST after deletion of node 14. Node 18 has taken the place of the deleted node and the entire subtree moved up one level.

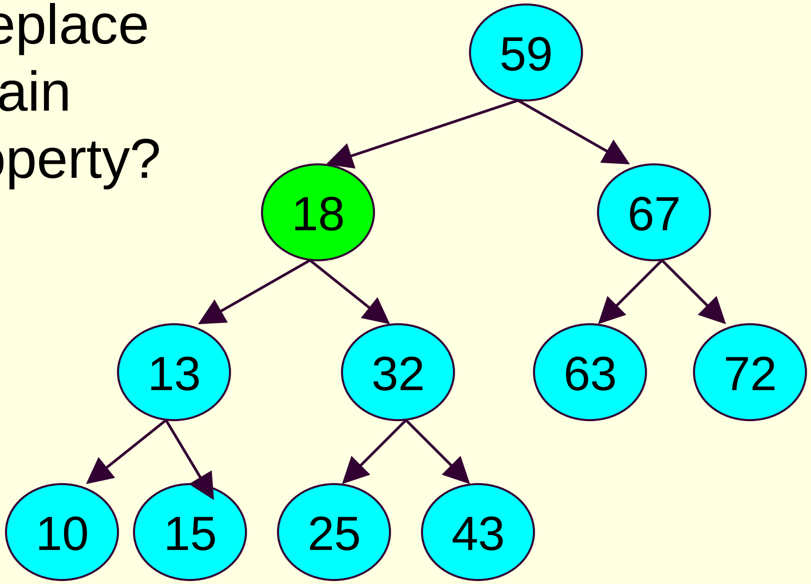
Binary Trees: Deletion



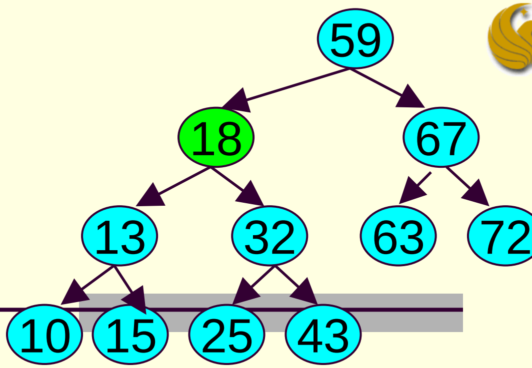
- Deleting a Node with two children
  - This is the scenario that requires a bit of thought
    - If we wish to delete node b (example above)
    - We can't just raise up b's children
      - since node d can't use its left pointer to point to more than one child
      - d's left can't point to both node a and node c
  - So think about what we need to do in order to maintain the structure (and ordering property)

Binary Trees: Deletion

- Deleting a Node with two children
  - Consider this example tree
  - We want to delete node 18
  - Ask yourself:
    - What two nodes could I replace the 18 with and still maintain the binary search tree property?



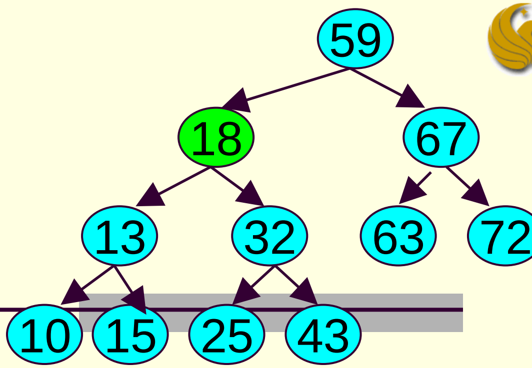
Binary Trees: Deletion



- Deleting a Node with two children
  - Remember:
    - All the nodes to the left of 18 MUST be smaller than 18
    - All the nodes right of 18 MUST be greater than 18
  - Thus, if we delete 18
    - there are only two nodes we could put at 18's position without causing serious repercussions:

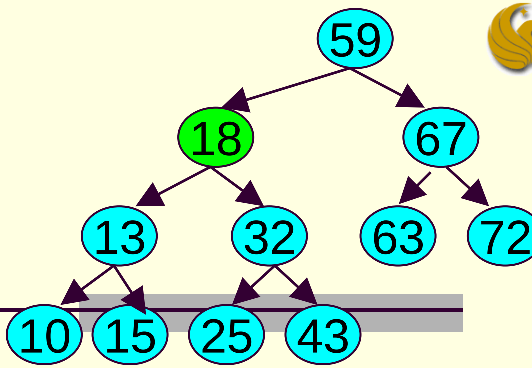
1) The **maximum value in the left subtree** of node 18
   - Which is 15
2) The **minimum value in the right subtree** of node 18
   - Which is 25

Binary Trees: Deletion



- Deleting a Node with two children
  - Thus, if we delete 18
    - There are two possible nodes that could go into 18's position:
      1) Node 15 (greatest value in left subtree)
      2) Node 25 (smallest value in right subtree)
    - We simply pick one of these to put at 18's position
      - We essentially **copy** the node to 18's position
    - Then we have to delete the actual node that we just copied
      - Meaning, if we copy node 15 into 18's positon
      - We will have two 15s
      - So we now need to delete the leaf node 15

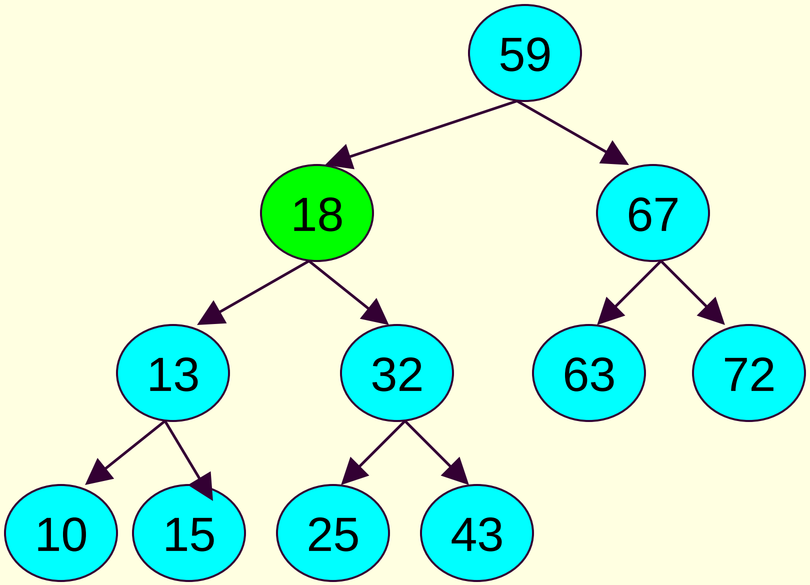
Binary Trees: Deletion

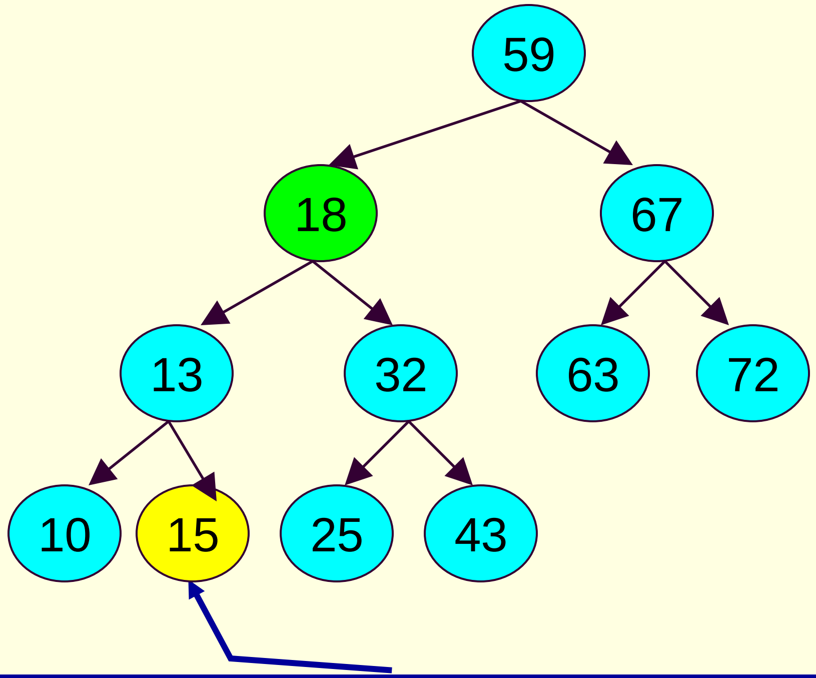


- Deleting a Node with two children
  - <u>We are **guaranteed** that this node</u>
    - Node 15 in this example
  - <u>Has AT MOST only one child</u>
    - Meaning it will be easy to delete!
  - Why is that? How is this guarantee true?
    - The <u>greatest node in a left subtree</u> cannot have two children
      - If it did, its right child would be greater than it
    - Similarly, the <u>smallest node in a right subtree</u> cannot have two children
      - If it did, its left child would be smaller than it

Binary Trees: Deletion

- Deleting a Node with two children (example)



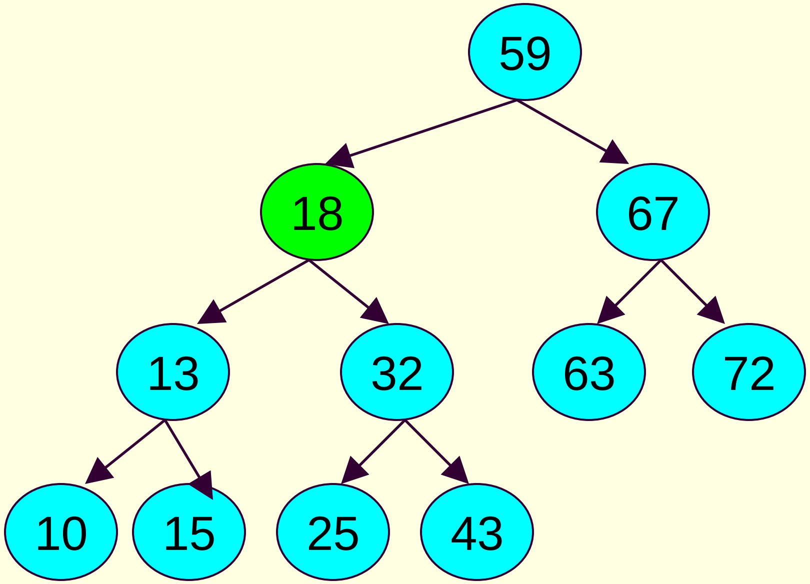


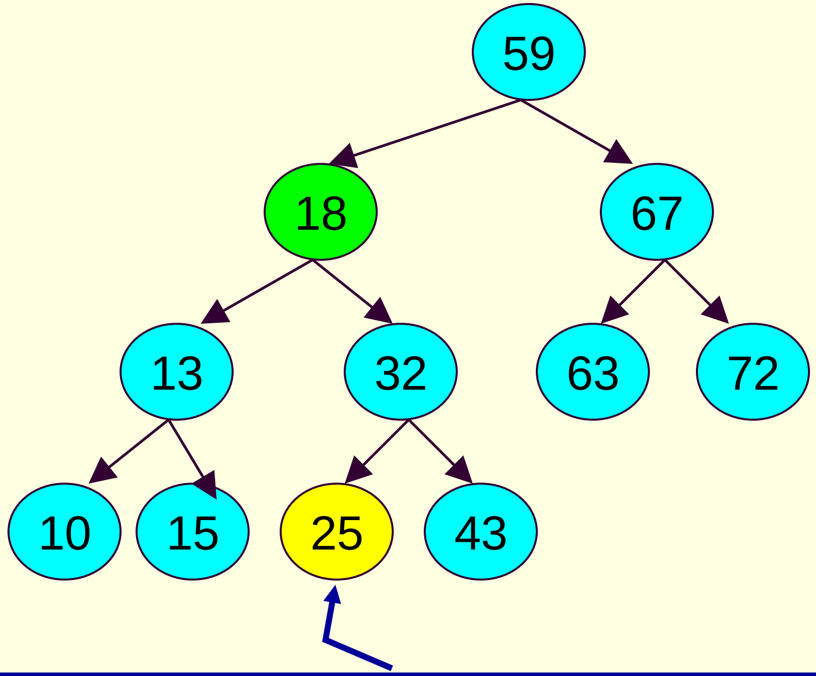
Initial BST with node 18 marked for deletion. Note that this node has two children with values 13 and 32.

This node contains the logical **<u>predecessor</u>** of the node to be deleted. Note that it is the **<u>greatest node</u>** in the **<u>left subtree</u>** of the node to be deleted.

Binary Trees: Deletion

- Deleting a Node with two children (example)



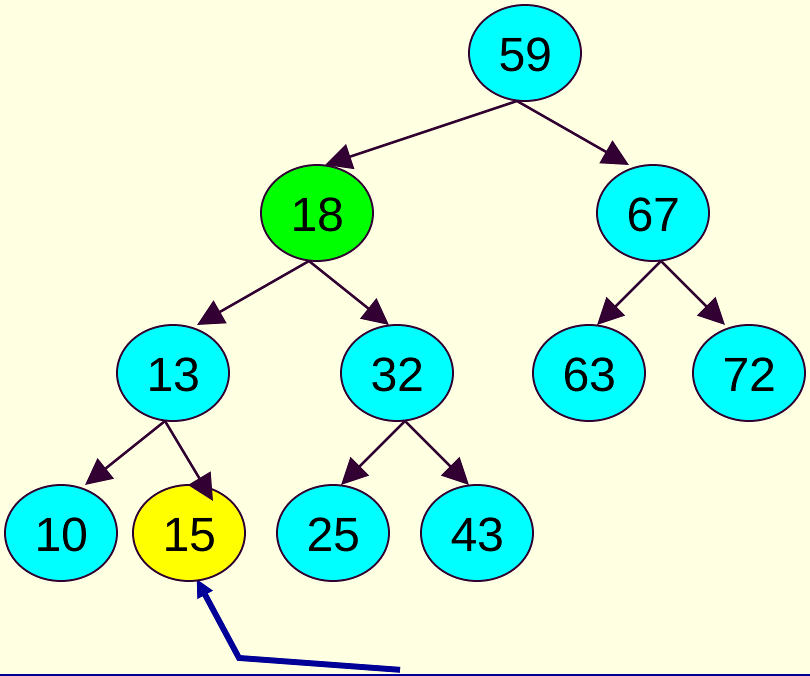


Initial BST with node 18 marked for deletion. Note that this node has two children with values 13 and 32.

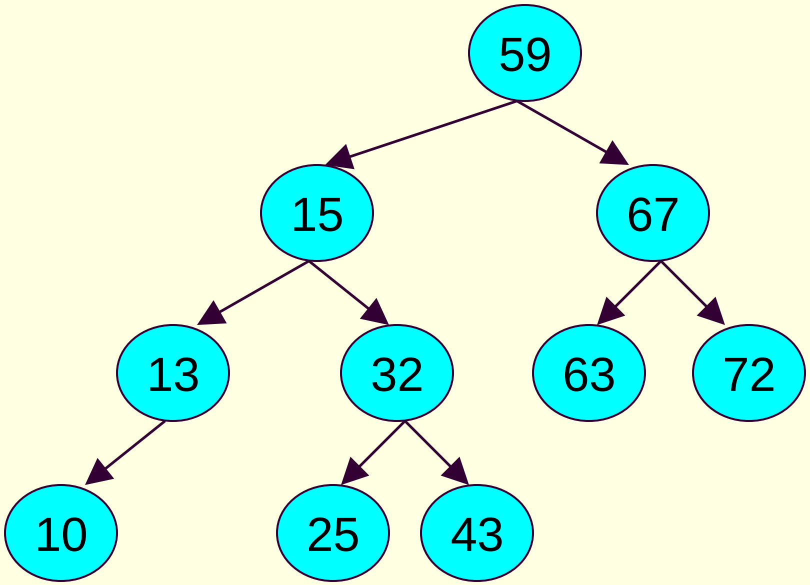
This node contains the logical **successor** of the node to be deleted. Note that it is the **smallest node** in the **right subtree** of the node to be deleted.

Binary Trees: Deletion

- Deleting a Node with two children (example)



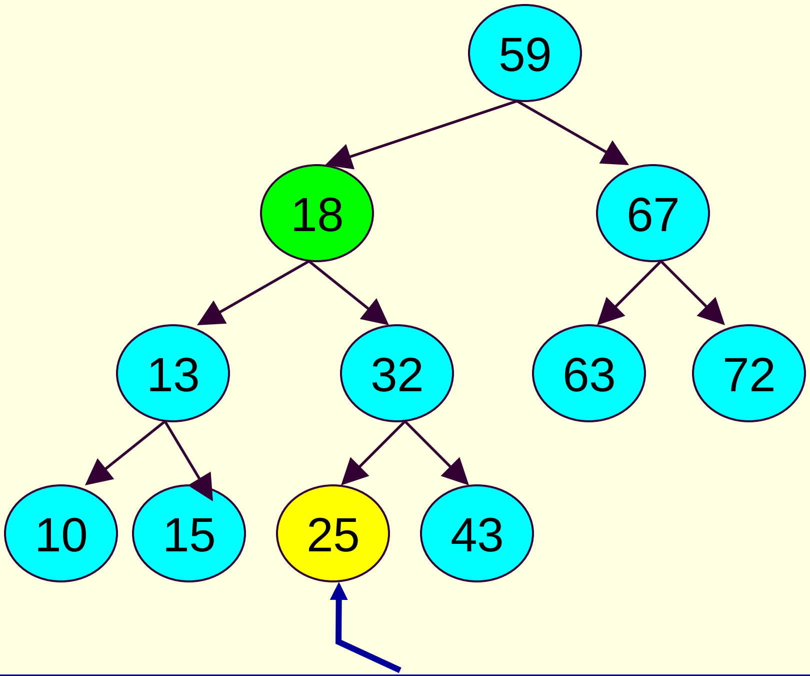
This node contains the logical **predecessor** of the node to be deleted. Note that it is the **greatest node** in the **left subtree** of the node to be deleted.



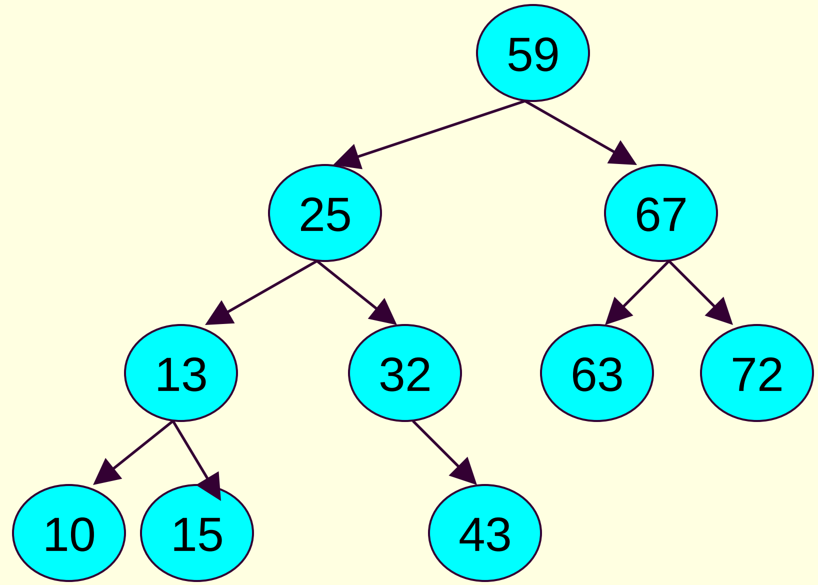
The BST after the deletion of node 18 using the replacement by the logical predecessor node.

Binary Trees: Deletion

- Deleting a Node with two children (example)



This node contains the logical **successor** of the node to be deleted. Note that it is the **smallest node** in the **right subtree** of the node to be deleted.



The BST after the deletion of node 18 using the replacement by the logical successor node.

Binary Trees: Deletion

- Deleting a Node with two children
  - For the previous example,
    - The nodes that we copied to 18's position were both leaf nodes
    - How did this help?
      - Once copied over, we need to delete those nodes
      - Since they are leaves, this process is easy
    - But what if they are not leaf nodes?
      - Meaning, they have one child
        - Remember, we are guaranteed that they have AT MOST one kid
      - It is still easy!
      - We would simply be deleting a node with one child
      - Which simply "lifts" up that subtree one level

Brief Interlude: FAIL Picture

Weekly UCF Bike Fail

Courtesy of
Kristina Lister

Binary Trees: Deletion

- Deleting a Node with two children
  - There's a lot going on for deletion
  - So when you examine the code,
    - You will see many auxiliary functions used, such as:

1) **`findNode`**: returns a pointer to a node in a given tree that stores a particular value
2) **`parent`**: finds the parent of a given node in a given binary tree
3) **`minVal`**: finds the minimum value in a given binary tree
4) **`maxVal`**: finds the maximum value in a given binary tree
5) **`isLeaf`**: determines if a node is a leaf node or not
6) **`hasOnlyLeftChild`**: determines if a node ONLY has a left child
7) **`hasOnlyRightChild`**: determines if a node ONLY has right kid

Binary Trees: Deletion

- Auxiliary functions:
  - `findNode`: returns a pointer to a node in a given tree that stores a particular value
    - The first step in deletion is finding the node in the tree
    - This is basically the search function from last time
    - The arguments to the function are:
      - A pointer to the root of some tree (or subtree)
      - The value we are searching for
    - IF found, the function returns a pointer to the node
    - Else, NULL is returned

Binary Trees: Deletion

- Auxiliary functions:
  - `findNode`: returns a pointer to a node in a given tree that stores a particular value

```
struct tree_node* findNode(struct tree_node *current_ptr, int value) {
        if (current_ptr != NULL) {
                // Found the value at the root.
                if (current_ptr->data == value)
                        return current_ptr;
                // Search to the left.
                if (value < current_ptr->data)
                        return findNode(current_ptr->left, value);
                // Or...search to the right.
                else
                        return findNode(current_ptr->right, value);
        }
        else
                return NULL; // No node found.
}
```

Binary Trees: Deletion

- Auxiliary functions:
  - `parent`: finds the parent of a given node in a given binary tree
    - Remember: we need to know the parent of the node we wish to delete
    - This allows us to modify the left/right pointers of the parent, effectively removing the child node
    - The arguments to the function are:
      1) The root of the tree
      2) A pointer to the node we want to find the parents of
    - If found, a pointer to the parent node is returned

Binary Trees: Deletion

- Auxiliary functions:
  - `parent`: finds the parent of a given node in a given binary tree

```
struct tree_node* parent(struct tree_node *root, struct tree_node *node) {
        // Take care of NULL cases.
        if (root == NULL || root == node)
                return NULL;
        // The root is the direct parent of node.
        if (root->left == node || root->right == node)
                return root;
        // Look for node's parent in the left side of the tree.
        if (node->data < root->data)
                return parent(root->left, node);
        // Look for node's parent in the right side of the tree.
        else if (node->data > root->data)
                return parent(root->right, node);
        return NULL; // Catch any other extraneous cases.
}
```

Binary Trees: Deletion

- Auxiliary functions:
  - `minVal`: finds the minimum value in a given binary tree
    - Remember: when we delete a node with two children
    - We need to replace that node with either:
      - The minimum value in the right subtree, or
      - The maximum value in the left subtree
    - The argument to the function is the root of the tree
    - The function simply returns a pointer to the node containing the minimum value

Binary Trees: Deletion

- Auxiliary functions:
  - `minVal`: finds the minimum value in a given binary tree
    - Remember:
      - The minimum value in a given binary tree will either be the root OR it will be found in the left subtree

```
struct tree_node* minVal(struct tree_node *root) {

        // Root stores the minimal value.
        if (root->left == NULL)
                return root;

        // The left subtree of the root stores the minimal value.
        else
                return minVal(root->left);
}
```

Binary Trees: Deletion

- Auxiliary functions:
  - `maxVal`: finds the maximum value in a given binary tree
    - Remember:
      - The maximum value in a given binary tree will either be the root OR it will be found in the right subtree

```
struct tree_node* maxVal(struct tree_node *root) {

	// Root stores the maximal value.
	if (root->right == NULL)
		return root;

	// The right subtree of the root stores the maximal value.
	else
		return maxVal(root->right);
}
```

Binary Trees: Deletion

- Auxiliary functions:
  - `isLeaf`: determines if a node is a leaf node or not
    - Remember:
    - The easiest form of deletion is when the node to be deleted is a leaf node
      - That's why we need this function
    - How can you tell if a node is a leaf?
      - Leaves don't have kids!
      - BOTH the left and right pointers will be NULL

```
// Returns 1 if node is a leaf node, 0 otherwise.
int isLeaf(struct tree_node *node) {

        return (node->left == NULL && node->right == NULL);
}
```

Binary Trees: Deletion

- Auxiliary functions:
  - `hasOnlyLeftChild`: determines if a node ONLY has a left child
    - Remember:
    - The second easiest form of deletion is when the node to be deleted has only one child
      - We simply delete that node and “lift” the child subtree 1 level
    - How would you determine if a node has only a left kid?

```
// Returns 1 if node has a left child and no right child.
int hasOnlyLeftChild(struct tree_node *node) {

        return (node->left!= NULL && node->right == NULL);
}
```

Binary Trees: Deletion

- Auxiliary functions:
  - `hasOnlyRightChild`: determines if a node ONLY has a right child
    - Same as with left child
    - Just we're now checking for an only right child

```
// Returns 1 if node has a right child and no left child.
int hasOnlyRightChild(struct tree_node *node) {

        return (node->left== NULL && node->right != NULL);
}
```

Binary Trees: Deletion

- Deletion From a Binary Search Tree
  - So now we can examine the full delete function
  - Too large to put on these slides

  - Here's the link to the binary tree code:
    - http://www.cs.ucf.edu/courses/cop3502/spr2011/programs/trees/bintree.c

Binary Trees: Deletion

WASN'T

THAT

STUNNING!

Daily Demotivator

Binary Trees: Deletion

Computer Science Department
University of Central Florida

COP 3502 – Computer Science I