Data Structures

Binary Trees

Phil Tayco

Slide version 1.2

Apr. 20, 2020

Binary Trees

Back to dynamic data structures

- The main benefit with a linked list is its dynamic memory allocation – we use nodes only when we need them
- Array memory allocation is static, but sorting them allows us to use binary search and go from O(n) to O(log n)
- Sorting linked lists can be done, but would not give us the benefit of doing so since binary search cannot be applied since we can't directly access any element at any time
- The goal is to now try to get the best of both worlds - dynamic memory allocation with the ability to execute at a binary search performance

Binary Trees

Binary search and dynamic nodes

- Binary search works by looking at the “middle” of the list and dividing the list in half with each search iteration
- Linked lists can only provide access from one or two nodes (head and tail) – direct access to the middle cannot be easily done
- As you saw with the programming 1 project, nodes can be modified to have multiple references to other nodes (next and previous)
- We can use this idea to simulate binary search by starting at a “middle” node and then going “left” or “right”

Binary Trees

Node definition

- We start with the structure by designing the node to hold 3 properties. The data itself and 2 pointers to other Nodes

```
public class Node {
   public int data;
   public Node left;
   public Node right;
}
```

- This is not much different from the structure of a Node in a doubly linked list. Instead of saying “next” and “previous”, we rename it to “left” and “right”
- “Next” and “Previous” imply the node that appears after or before it, and nothing more. With “left” and “right”, we can add more meaning

Binary Trees

Left and Right Sides

- Now instead of looking at just what node appears next or previous to it, we can extend it to mean the entire left “side” or right “side” of that node
- If we’re using this for something like binary search, remember the algorithm says to start in the “middle” of the list – if the value is not found and is less than where you currently are, look on the left side of the list. Otherwise, look right
- With this updated Node, a similar process can occur – if the search value is not found with the current node, go to the left or right node
- This suggests that the node values now relate to each other in some “sorted” way

Binary Trees

Node rules

- Since we're using this structure to simulate binary search, we will call this type of tree a Binary Search Tree
- The rule of this tree is such that for any node, the node to its "left" is less than it and the node to its "right" is greater
  - To support duplicate values, you can choose either the left or right, but it must be always left or always right if the node values are equal – more on this later
- In order to support this rule, nodes that are added into the tree must adhere to it during insert

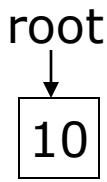
Binary Trees

Insert algorithm

- As new nodes are added to this structure, the appropriate location must be determined while maintaining the design intent:
- If the structure is empty, add the new node as the first element. We will call this the “root” node
- If root is not empty, “traverse” the structure by comparing the new node data value with the current node (current starts at root)
  - If the value is less than current, go to the left Node
  - Otherwise, go to the right Node
- Keep traversing this way until you see a null – at that point, the new Node is added there

Binary Trees

Example: Insert 10 – the root starts as empty so the new node becomes the root

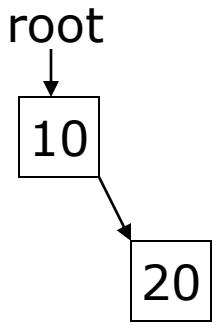


Binary Trees

Next, insert 20. We start at root and since it is not null, we must traverse

Since 20 is greater than 10, we must look right

The right pointer is null, so we add it there

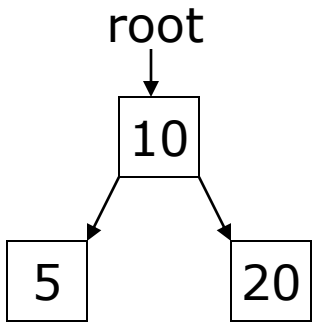


Binary Trees

Next, insert 5. Again, start at root and see that it not empty, so we traverse

5 is less than 10 so we go left

Left is null, so we add it there



Binary Trees

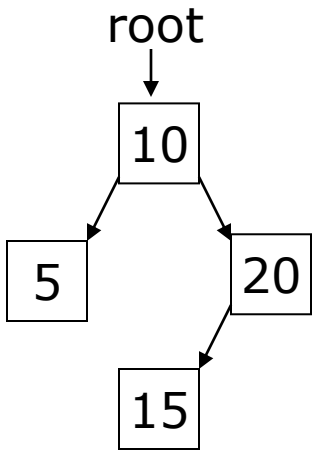
Now insert 15. Start at root, it's not empty, so we must traverse

15 is greater than 10 so we look right

20 is there (the current node we are at is not empty) so we traverse again

15 is less than it, so we look left

Left is null, so we add 15 there

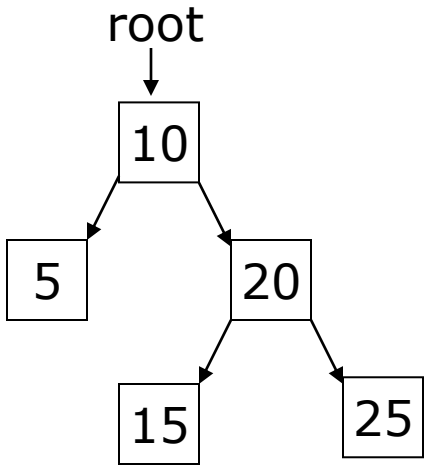


Binary Trees

Add 25. Starting at root, 25 is greater than 10 so we look right

25 is also greater than 20 so look right again

Right is null from 20, so we add the 25 there



Binary Trees

```
public class BinaryTree
{
    private Node root;

    public BinaryTree()
    {
        root = null;
    }
```

Binary Trees

```
public void insert(int n)
{
  Node current = root;
  Node newNode = new Node();

  newNode.data = n;
  newNode.left = null;
  newNode.right = null;

  if (root == null)
     root = newNode;
```

Binary Trees

```
    else
        while(true)
                if (newNode.data > current.data)
                        if (current.right == null)
                        {
                                current.right = newNode;
                                break;
                        }
                        else
                                current = current.right;
                else
                        if (current.left == null)
                        {
                                current.left = newNode;
                                break;
                        }
                        else
                                current = current.left;
}
```

Binary Trees

Code analysis

- The code follows the algorithm earlier stated
  - First, create a new node with no children (the new node will always have left and right as null)
  - If the root is null, simply set the root to the new node
  - Otherwise, perform the following loop until a null child pointer is found
    - If the new value is greater than the current node's value, it belongs on the "right" of the current node. If "right" is null, set it to the new node and exit the loop. Otherwise, traverse down the "right" pointer
    - Otherwise, the new node belongs on the "left" of the current node. If "left" is null, set it to the new node and exit the loop. Otherwise, traverse down the "left" pointer
- The loop will eventually reach a null pointer so there is no danger of an infinite loop

Binary Trees

Efficiency

- Each node "visited" effectively cuts off the other half of the list
- If the values being added have a random distribution of values, the performance is like a binary search and is thus, O(log n)
- Note that there is a dependency on the manner in which Nodes are inserted
  - If the root is too small or too large, subsequent levels will be on one side of it
  - If this pattern continues (such as inserting numbers in numeric order), the tree degrades into a linked list
  - Efficiency in such "unbalanced" trees degrades to O(n) for all functions
- There are ways to counter unbalancing in more advanced tree structures which we'll look at later
- As you can imagine, the other 3 major functions will follow a similar algorithm and efficiency. Let's look at "search" next

Binary Trees

Binary search enabled

- As more elements get added and continue to follow these rules, the structure starts to take the shape of a tree
- We call this a binary tree because the number of elements from each node is 2
- The binary tree structure and rules enable binary search to be simulated
  - If the search value is not equal to the root, traverse the left or right pointer based on the current node and search values (go left if search value is less, otherwise go right)
  - If you reach null, the search value is not present

Binary Trees

```
public Node search(int n)
{
  Node current = root;

  while (current != null)
     if (current.data == n)
          return current;
     else if (current.data > n)
          current = current.left;
     else
          current = current.right;
  return null;
}
```

Binary Trees

Code analysis

- We start with setting a temporary node to root (so we don't accidentally change root during the search)
- As long as the current node is not null, perform the following checks
  - If the current node value equals the search value, we found the node and we return it
  - Otherwise, if the new value is greater than the current node's value, the potential node we are looking for is on the "right" of the current node so we traverse down the right
  - Otherwise, the new node belongs on the "left" of the current node and we traverse down the "left" pointer
- If we reach this point, the current node ended up as null, meaning the search value doesn't exist

Binary Trees

Efficiency

- Once again, each node "visited" effectively cuts off the other half of the list
- As long as the tree is fairly balanced, the performance will be O(log n)
- If node values are maintained such that the tree is unbalanced, the performance degrades towards O(n)

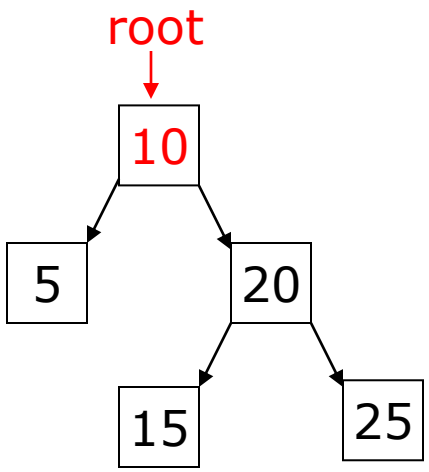
Binary Trees

Terminology

- Because this data structure takes the form of a tree graphically, tree terms are often used
- Familiarity with these terms is important for not only binary trees, but also for use with more advanced tree structures

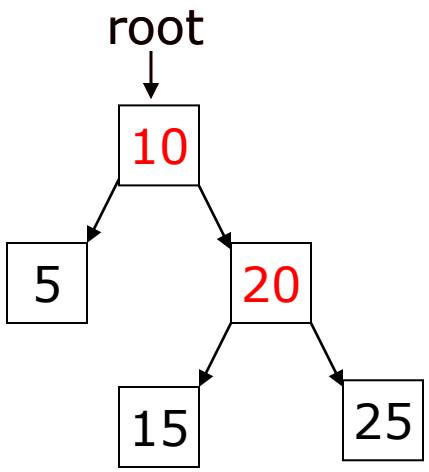
Binary Trees

"Root" is the first node of the tree



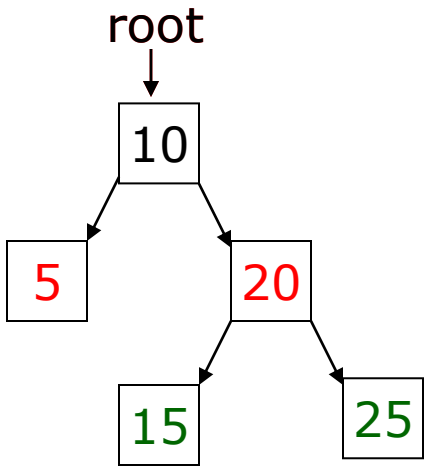
Binary Trees

"Parent" nodes have either left and/or right pointers of it pointing to existing nodes. 10 is the parent of 5 and 20. 20 is the parent of 15 and 25



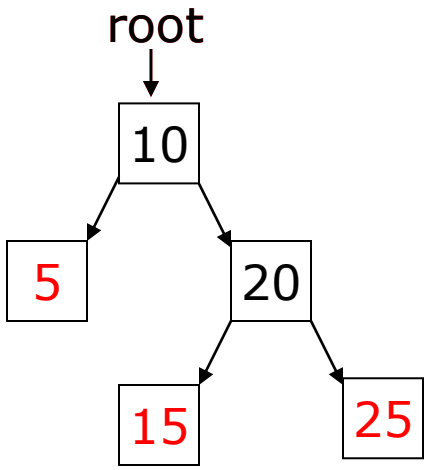
Binary Trees

"Child" nodes are nodes with a parent. 5 is a child of 10 and so is 20. 15 and 25 are children of 20. These nodes are also "siblings" to each other because they share the same parent



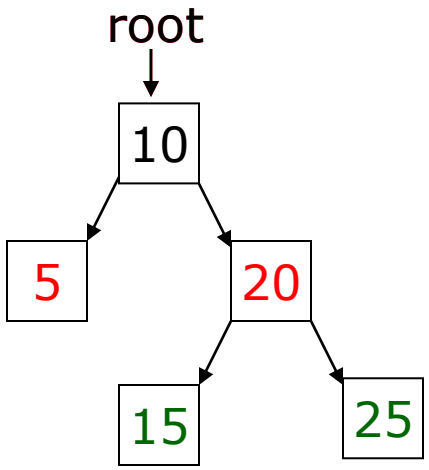
Binary Trees

"Leaf" nodes are nodes with no children. 5, 15 and 25 are such leaves



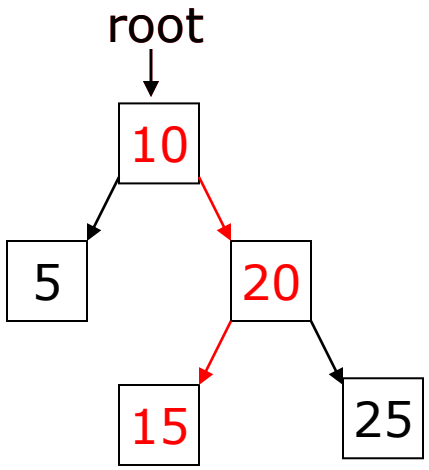
Binary Trees

Each “generation” of nodes is called a “level”. 10 is at level 0, 5 and 20 are at level 1 and 15 and 25 at level 2. The number of levels a tree has is called its “height”



Binary Trees

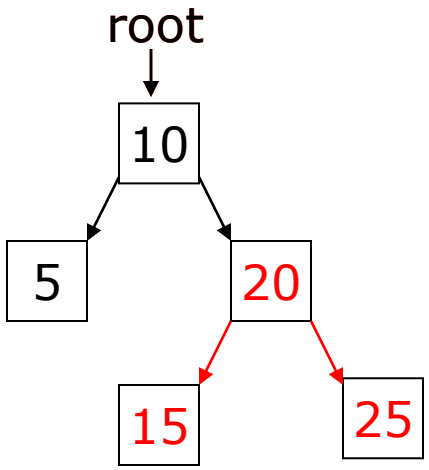
Traversing a tree is similar to traversing a linked list in that you follow the node pointers to get where you want. In a tree, such traversals are called a “path” or a “branch”



This example shows a path to node 15

Binary Trees

Subsets of a tree that form their own tree are called "subtrees". All nodes in a subtree are connected



Subtree 20 with children 15 and 25 are highlighted. Note that 20 and 5 together is not considered a subtree (they are not directly connected)

Binary Trees

Update and Delete

- The remaining 2 functions is where the complexity of tree structures begin to show
- Update is usually what we would tackle next, but let's think about what happens here:
  - An update is a search and change in value. Search is no problem
  - However, when the search results in an existing node to change, the new node value will very likely put it out of order in the tree
  - The node would then have to move to the correct place in the tree
- Such an update would be a challenge to do to find the new node's appropriate location relative to where the node was changed from
- It makes more sense instead to perform an update as a delete of the old value for the update followed by an insert of the new value if the node was found
- Thus, we'll look at the delete function first…

Binary Trees

Delete

- Okay, no problem right? Simply perform a search and find the node. If we find it, remove it
- However, with linked lists, we saw this was not everything because we also needed to maintain visibility to the "previous" node of the "current" node to appropriately maintain the pointers after the current node is removed
- Removing a node in the linked list had the previous' "next" pointer point to the current node's "next" pointer
- The same idea is used here treating a node's "parent" as its "previous"
- However, in a tree, the "next" pointer of node could be either the left or right child. Thus, the parent's child pointer must point to the correct current node's child node

Binary Trees

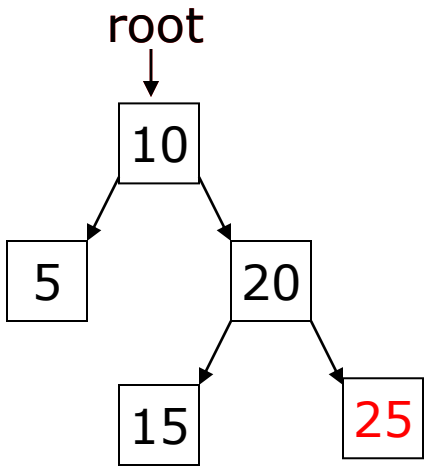
Situations

- How do we algorithmically accomplish this? Start with understanding that when a node is found to be removed, there will be 3 possible situations:
  - The node is a leaf (no children)
  - The node has 1 child on its left or right
  - The node has 2 children
- As you can probably see, the complexity increases as the node has 0, 1 or 2 children

Binary Trees

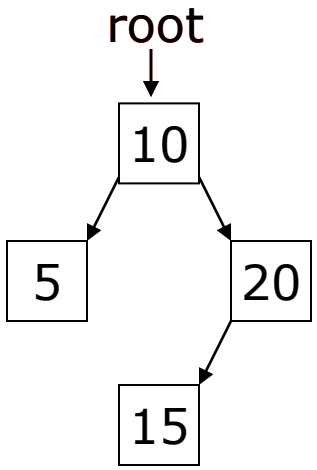
These situations are best seen with examples.

Looking at situation 1, that would be trying to delete 5, 15 or 25 below. Let's delete 25



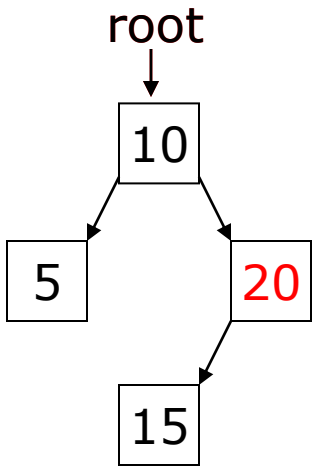
Binary Trees

Because 25 has no children, its parent node's "right" pointer can simply point to null



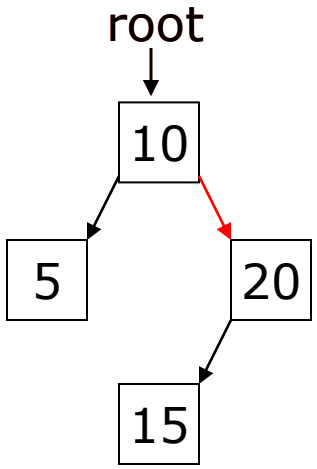
Binary Trees

Easy enough. Next situation is removing a node with 1 child. That would be 20 in the tree below



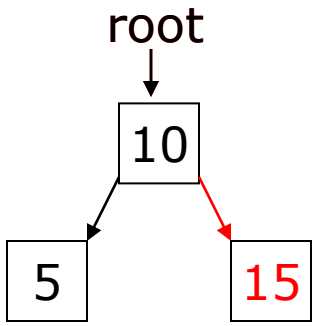
Binary Trees

Now we have to make sure 20's parent points to the correct child of 20. More specifically, since 20 is on the "right" of 10, 10's "right" pointer must point to the correct child of 20



Binary Trees

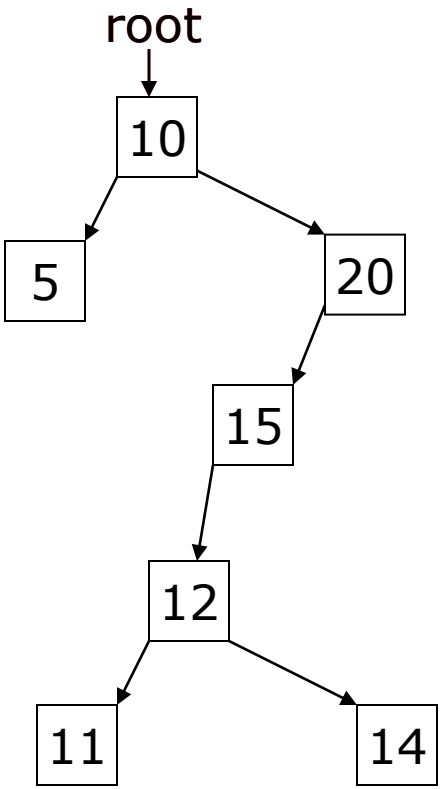
It turns out this is not terribly difficult because 20 (in this situation) only has one child to choose from. Since that child is on 20's "left", we make 10's "right" point to 20's "left" which is 15



Note that the order of the entire tree is maintained even though 15 was on the "left" of 20 (this was because 15 was first added by going to the "right" of 10 during insert)

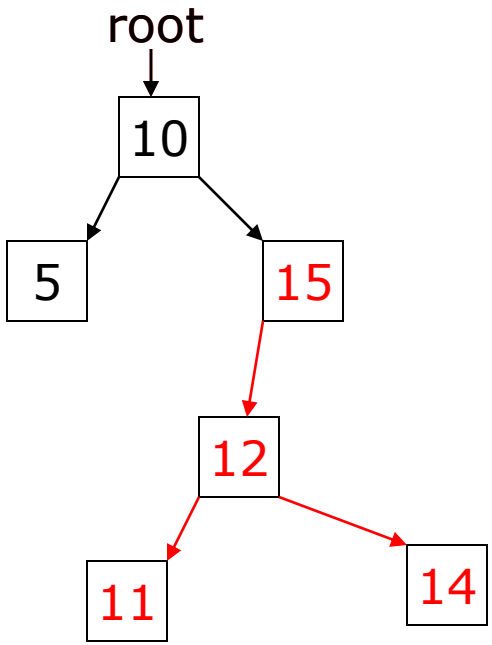
Binary Trees

What if the one child of 20 was a large subtree? Because of the way the pointers are set up and how insert works, the overall tree order still remains intact. For example:



Binary Trees

If we remove 20, 20 still only has 1 direct child, so when we assign 20's parent's right child to 20's left child, that entire subtree becomes 10's "right" child and the order is still intact



Binary Trees

Situations 1 and 2 addressed

- Search for the node – if not found, you're done
- If found and the node to delete has no children, remove it and set the appropriate child pointer of the parent to null
- If the node has 1 child, have the parent's appropriate child pointer point to the child of the node being removed
- So far so good!...

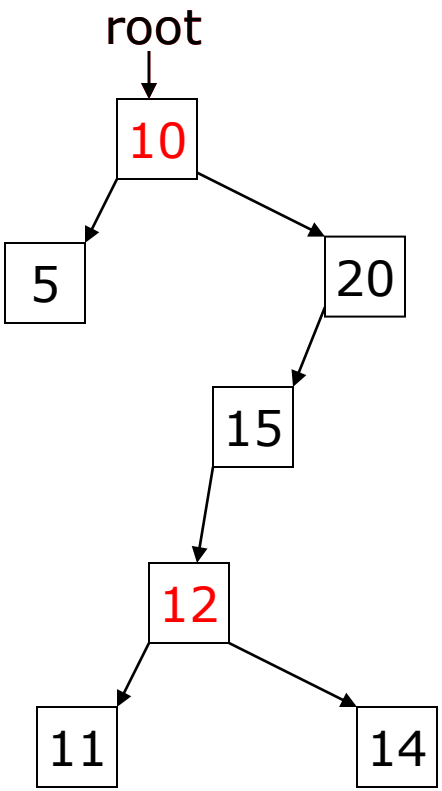
Binary Trees

Situation 3 though…

- The idea is the same with 2 children, but now we have to choose which child the parent will take on
- As with all algorithms, the steps are systematic – there is no ambiguity on the logic
- Let’s take a look at the same tree on the previous slide and note the two nodes that have two children. Can you spot them?

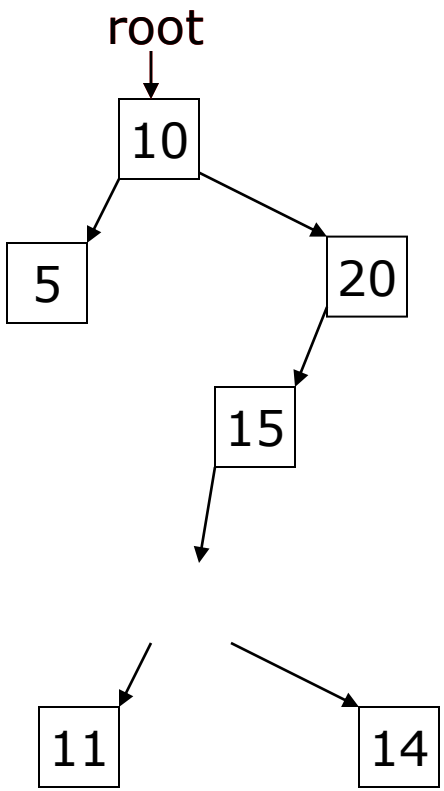
Binary Trees

10 and 12 are both nodes that have 2 direct children. Notice that there is a difference, though, if we delete 12 versus 10. The situations appear different as far as picking the “correct” child node goes



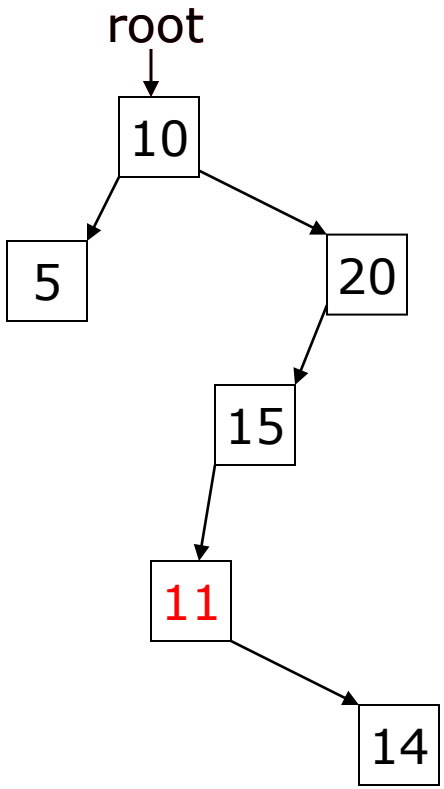
Binary Trees

If we remove 12, we have to choose from either 11 and 14 to take its place and the structure order will remain intact



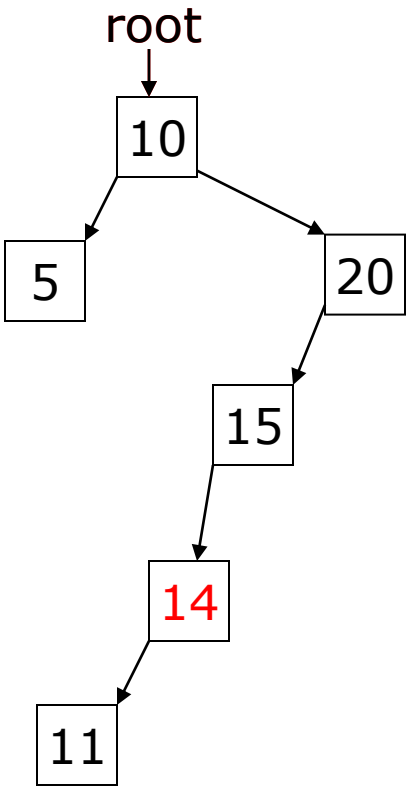
Binary Trees

If 11 as its left child took its place, 14 would then become the right child of 11 – tree design is still intact



Binary Trees

Similarly, 14 could take its place and 11 would then have to become 14's left child – tree structure also intact



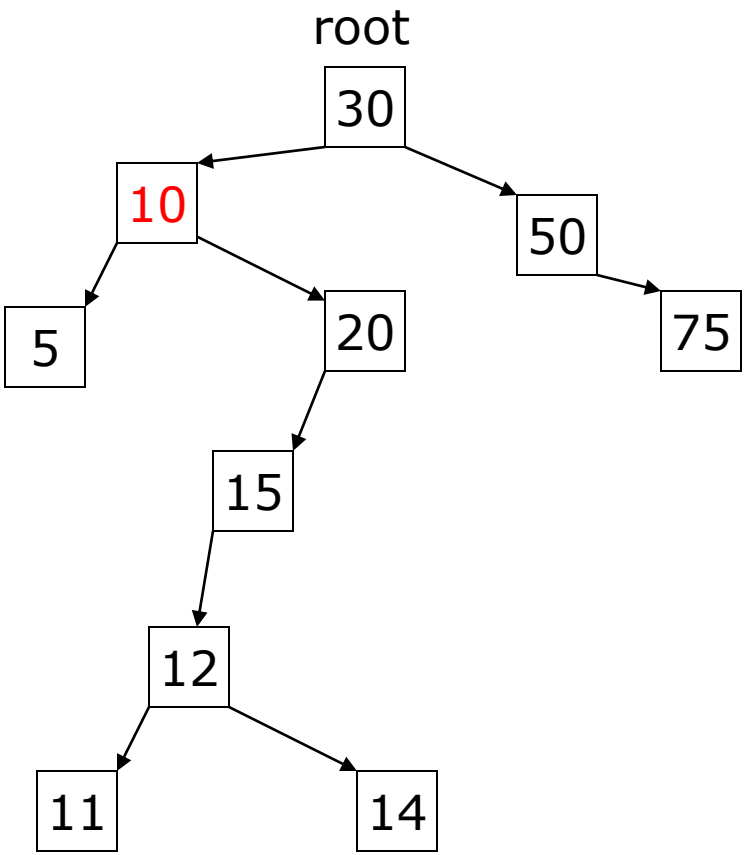
Binary Trees

Situation 3

- The selection of which child to replace 12 when it is removed is arbitrary. Either 11 or 14 will work
- This is because both 11 and 14 are leaf nodes making picking either one keeps the tree order intact
- The child links still need to be arranged with the replacing node inheriting the child node of the one removed:
  - When 11 replaced 12, it took 12’s right child as its own right child
  - When 14 replaced 12, it took 12’s left child on its left
- Most delete situations with nodes having 2 children, though, will not have children that are leaves

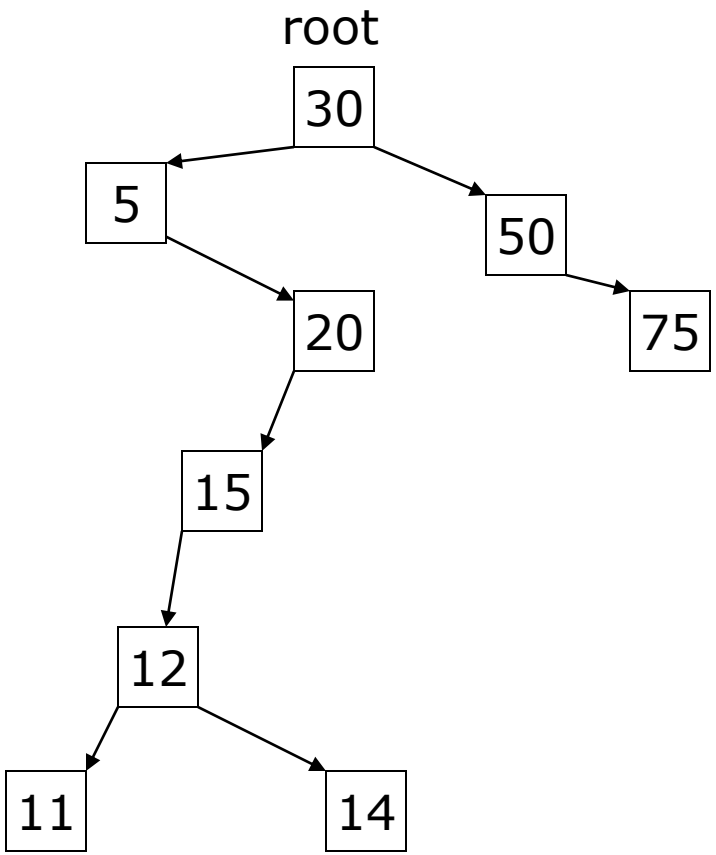
Binary Trees

Here's another example with a slightly larger tree. Let's look at removing 10. Note we are still in situation 3: node 10 has 2 children



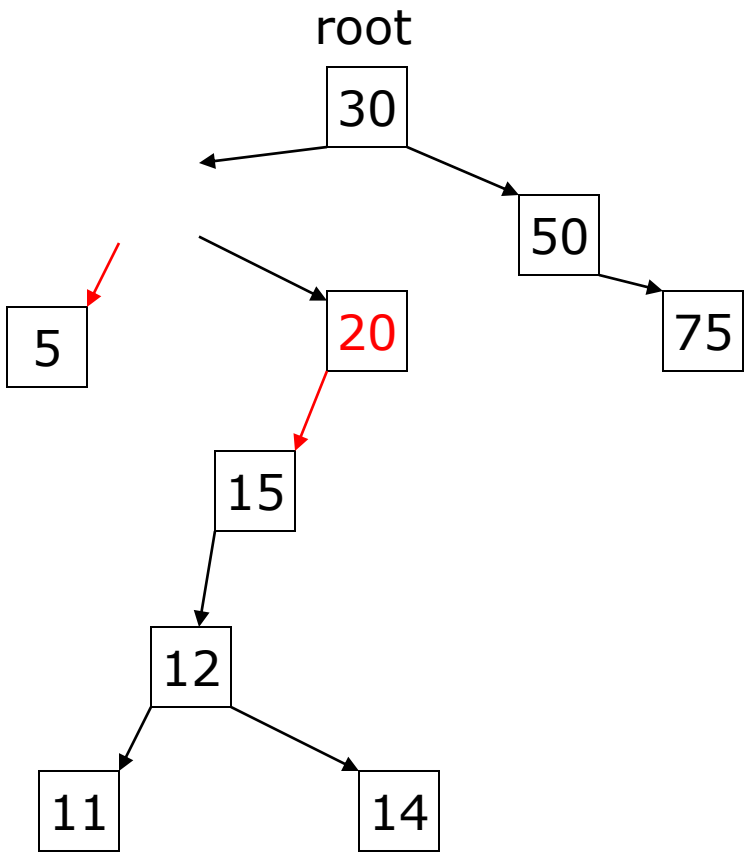
Binary Trees

If we replaced it with node 5, the structure is intact implying that if one of the child nodes is a leaf, it can replace the node being removed



Binary Trees

We could not do the same with 20. Notice in this example that 20 already has a left child and 10 had one as 5. If 20 takes its place, where does 5 go?



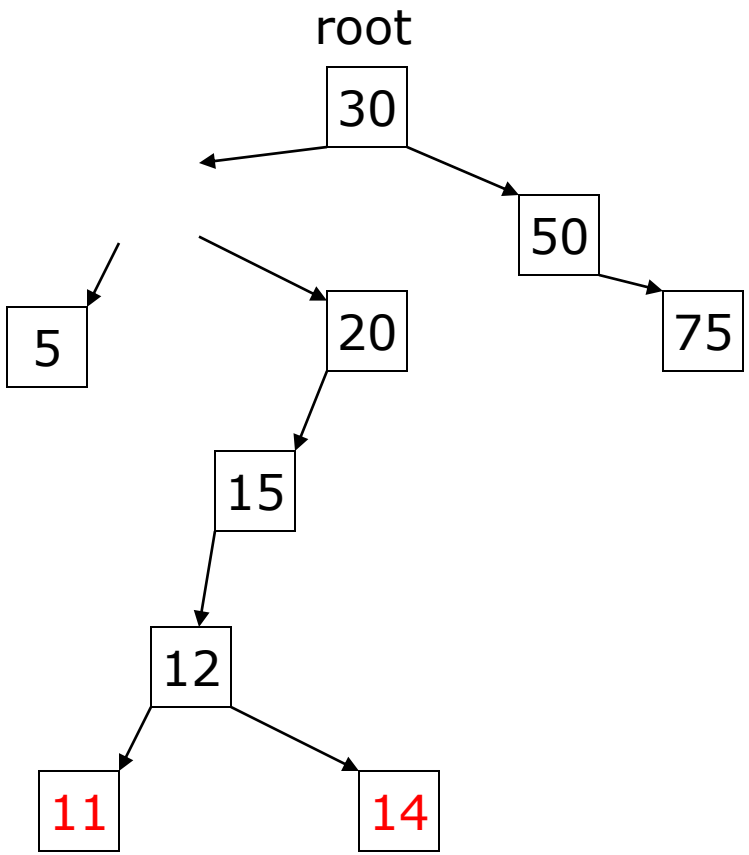
Binary Trees

Situation 3

- Deleting a node with 2 children is easier when one of the nodes is a leaf
- The problem with this is that most of the time with larger trees, the child nodes will not be leaves making an initial check to see if they are leaf nodes effectively unnecessary
- We need to find an algorithm that identifies the correct replacing node in an arbitrary subtree instead of a subtree with specific situations like it's a leaf
- Take a look at the subtree with 10 removed. Which node can replace 10 with minimal work required to rearrange node pointers?

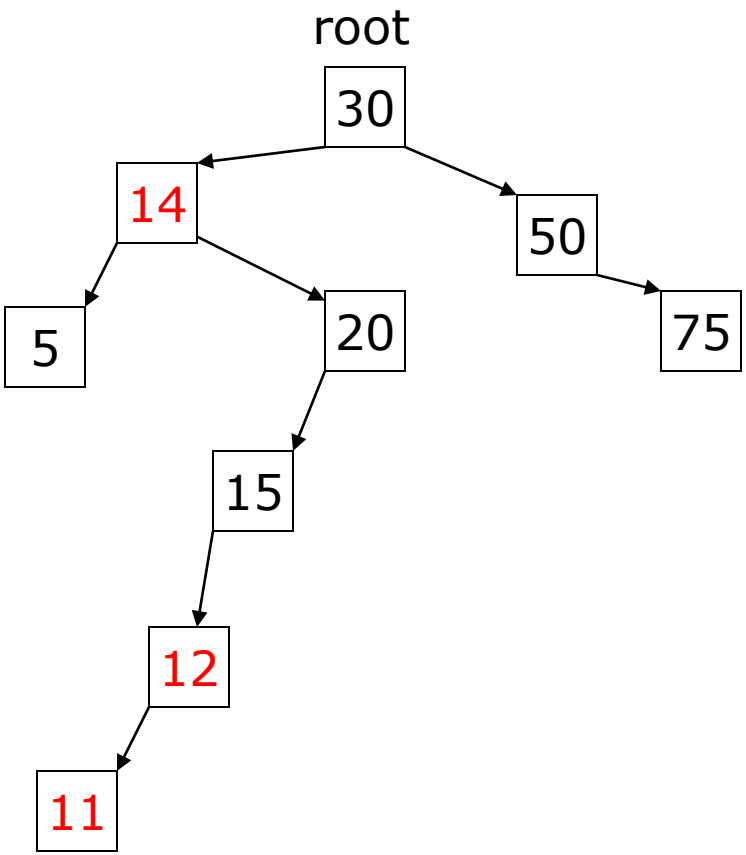
Binary Trees

A leaf node may be a good candidate because it has no child nodes to deal with. If we go with a leaf node in the subtree of 10, it would have to be either 11 or 14.



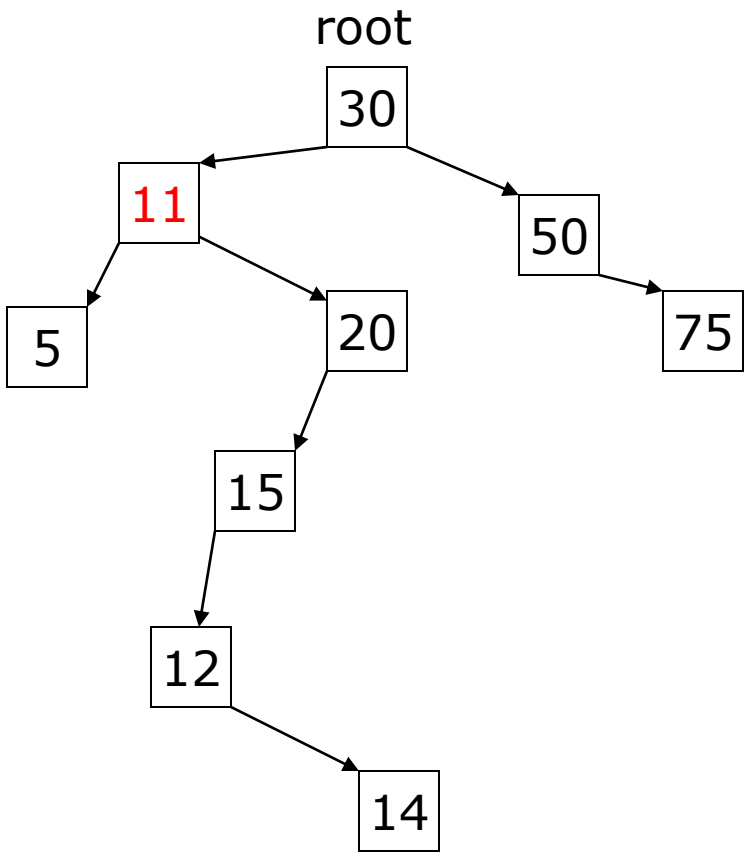
Binary Trees

14 would not work though because subtree 12 should not be on the “right” of 14



Binary Trees

However, 11 works and works very well!



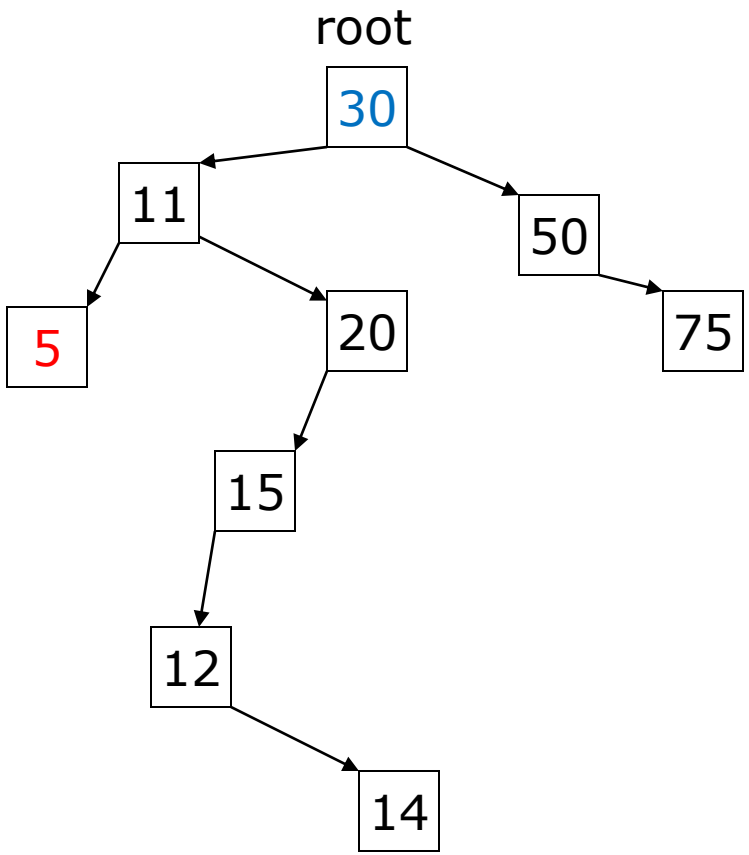
Binary Trees

Situation 3 – almost there!

- Replacing 10 with 11 looks like it worked because it was a leaf node, but there is actually a better reason
- What made 11 be the best candidate? The answer lies in the node value that was being removed, which was 10 - **11 is the closest value to 10 of all values in the right child subtree**
- Could we have gone down the left subtree instead? Sure, and it turns out if you do, the same idea of finding the value closest to it in that subtree is the key
- We'll go with always picking the right subtree for now. Our algorithm for situation 3 is:
  - Go down the right subtree
  - Descend the subtree until you reach the correct node and use it to replace the node being removed updating the parent and child pointers appropriately
- How do know what the "correct" node is?

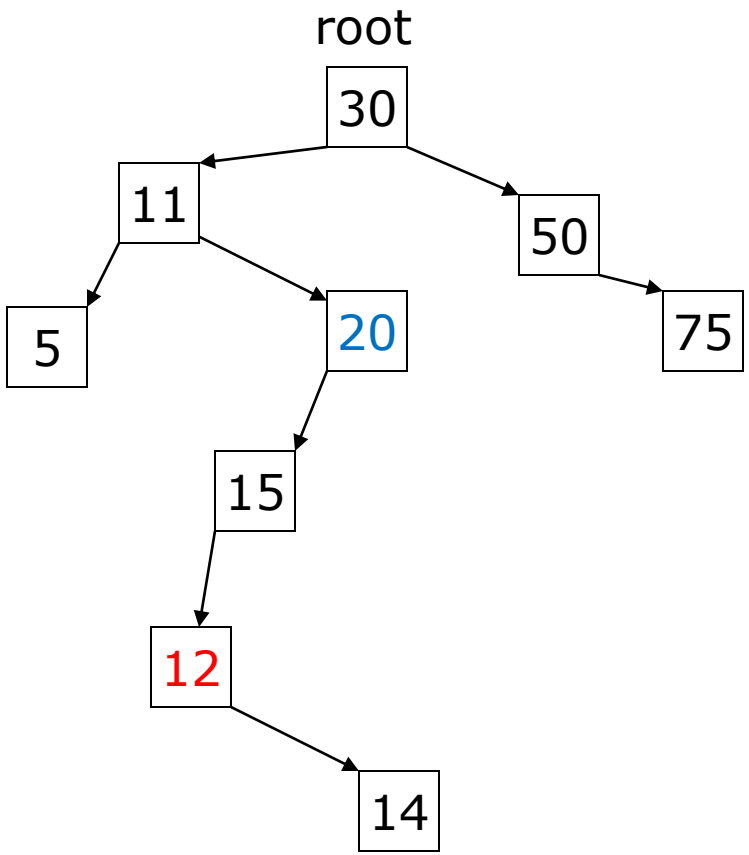
Binary Trees

The smallest value in any tree will be its “leftmost” child. For the entire tree starting at 30, that value is 5



Binary Trees

For the subtree starting at 20, that value is 12



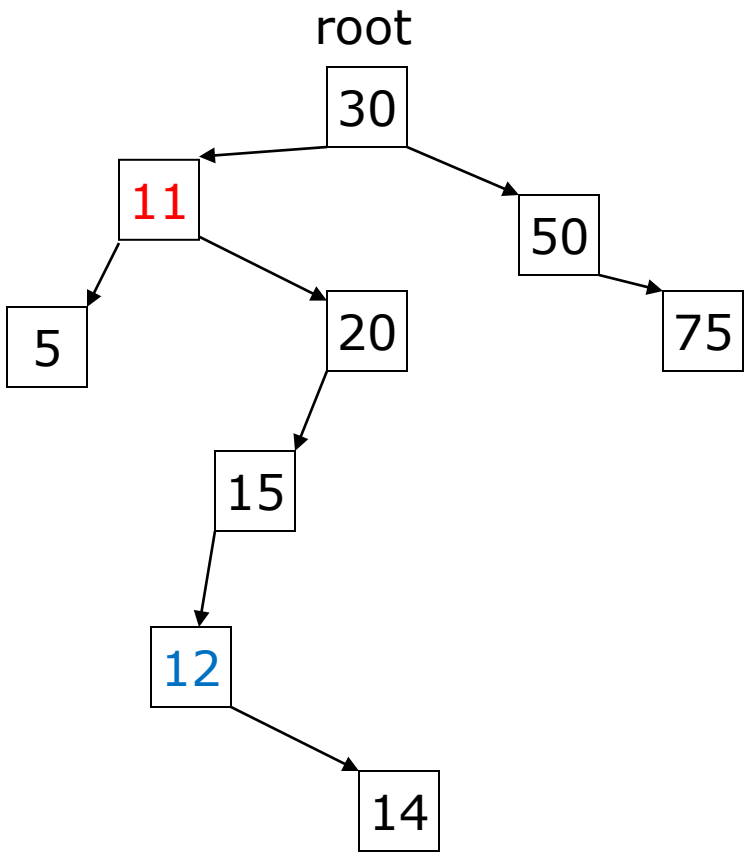
Binary Trees

Leftmost value

- Now notice that for node 11, the right subtree starts at node 20
- To find the closest value to 11 in that right subtree, we want the smallest value in it
- This means once you enter the right subtree, go as far left as possible – the last node you see is the closest value to 11
- Our algorithm for situation 3 now is:
  - Go to the right subtree
  - Descend the subtree left branch until you cannot traverse any more left child nodes
  - Use it to replace the node being removed
- Replacing the node being removed, though, is no easy feat. Let's see go back to the same tree and see when we try to delete 11…

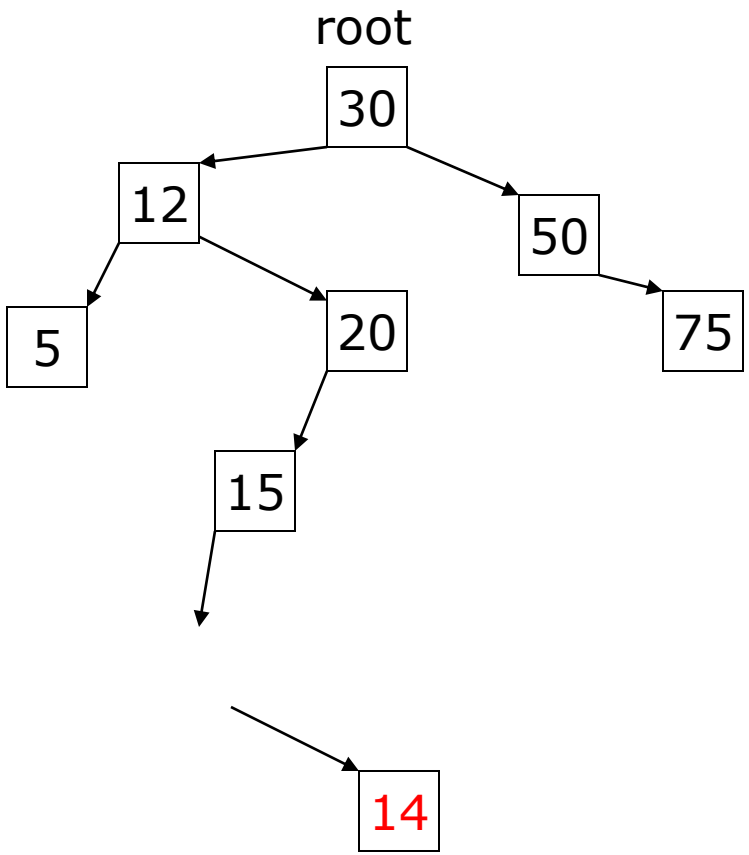
Binary Trees

We want to delete 11. According to our algorithm, 12 is the closest value to it in the right subtree…



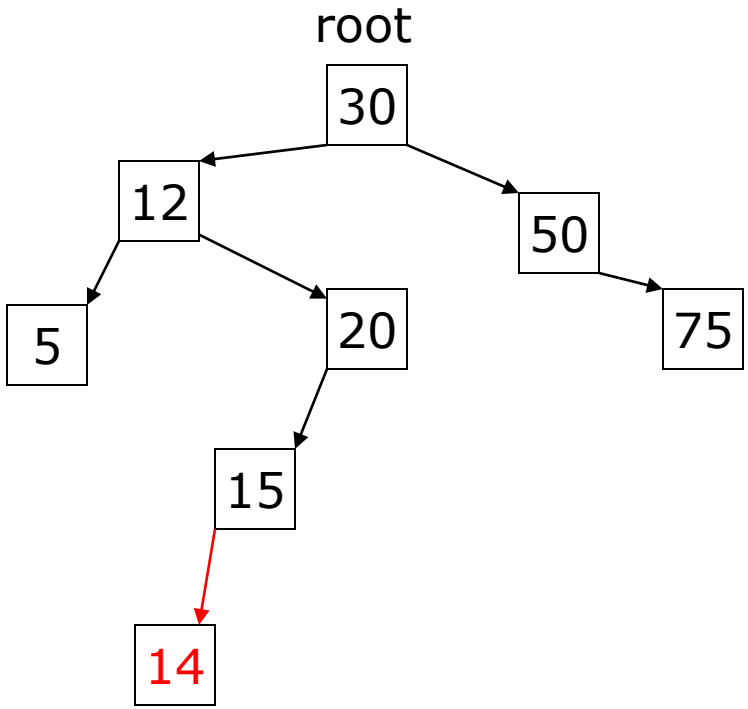
Binary Trees

But 12 has a child (node 14) – if we replace the deleted 11 with, what do we do with 14?



Binary Trees

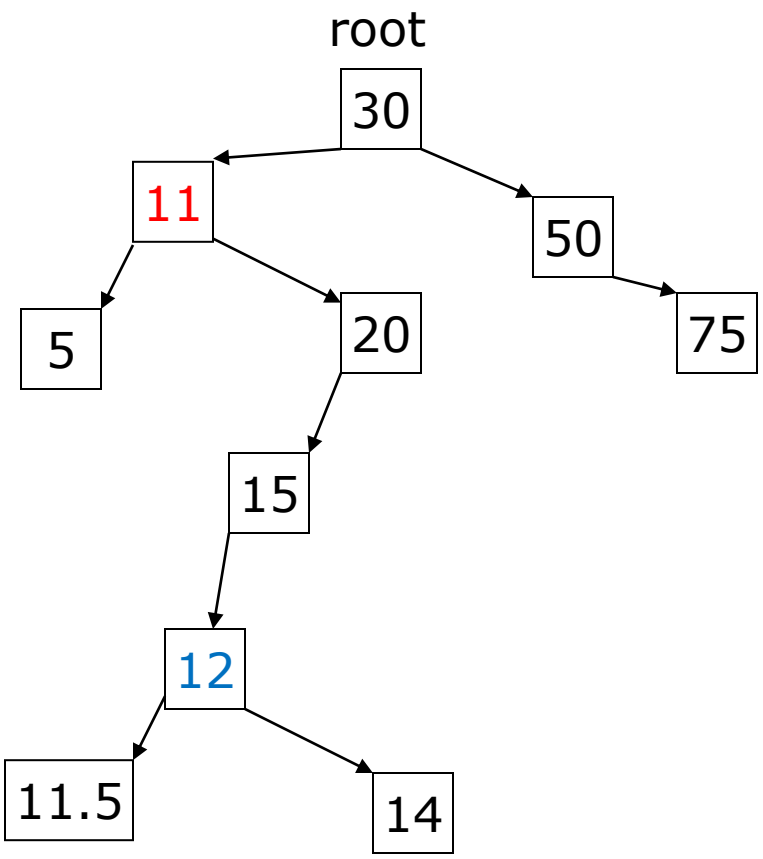
To keep the tree design intact, the parent of 12 must take on 12's child



Question: what if 12 had 2 children?

Binary Trees

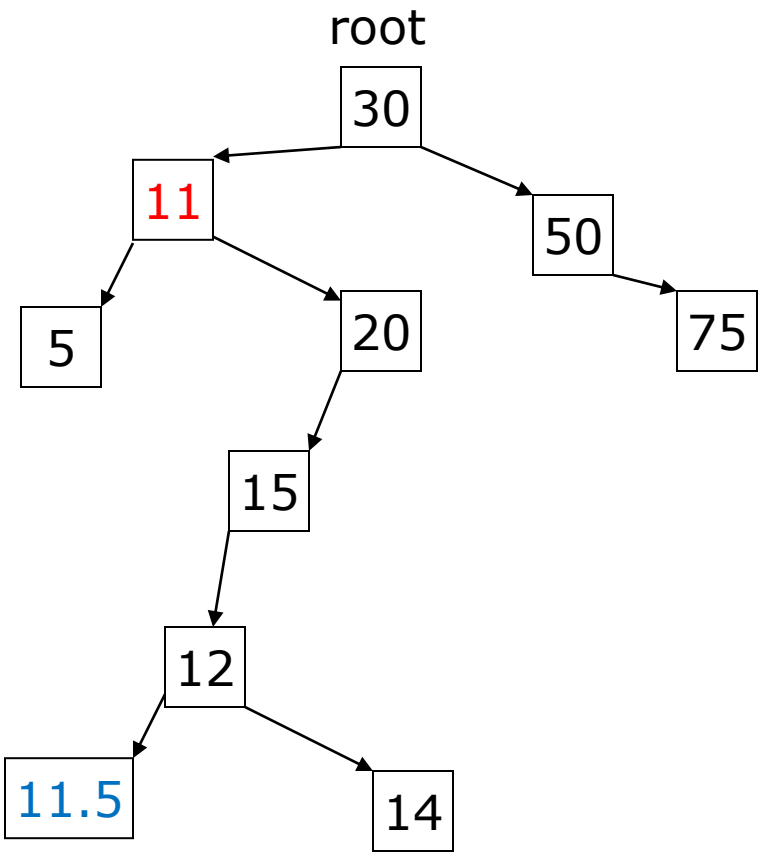
If 12 had two children, the tree could look like this:



Is this a problem with replacing 11 with 12 now?

Binary Trees

Yes, but 12 is no longer the correct value to replace 11. It's now 11.5 because it is the leftmost child of the right subtree of 11!



Binary Trees

Child of the closest value

- By logic, then, the closest value in the right subtree of the node being deleted will either only have 1 child or none at all
- It cannot have 2 children because it would then, not be the closest value (its left child's value is closer to the value of the node being deleted_
- That simplifies things for when replacing the deleted node with the closest value node in the right subtree as that node will either be a leaf or have 1 child
  - If there are no children, simply replace the deleted node
  - If there is 1 child, the parent of the node replacing the deleted node takes on that child

Binary Trees

The Situation 3 algorithm

- If the node to be deleted has 2 children, go to the right subtree
- Go as far left as possible until you reach a node with no left child – let's call this the "successor" node
- Before replacing the deleted node with the successor, set the "right" child of the successor to its parent's "left" child (by logic, this will always be the case)
- Replace the deleted node with the successor and set its "right" child to the "right" child of the node that was deleted
- Finally, set the deleted node's parent's appropriate child pointer to the new successor
  - Exception: If the deleted node is root, set root to be the successor node

Binary Trees

Put it all together

- It may seem like a lot of checks, but the code follows the logic systematically
- First, check if the tree is empty (as always) – we're done if it is
- Search the tree for the node to remove and keep track of not only the parent and current nodes, but also whether the current node is on the left or right of the parent
- If current ends up as null, the node to remove is not found and we're done
- If we find the node, then we handle 1 of the 3 delete node situations previously discussed (no children, 1 child, or 2 children)
- Let's look at the code to see this all in action

Binary Trees

```
public boolean remove(int n)
{
  // Check empty tree
  if (root == null)
     return false;

  // Prepare search for node
  Node current = root;
  Node parent = root;
  boolean currentIsLeft = true;
```

Binary Trees

```
while (current.data != n)
{
     // currentIsLeft is true when current is finds n
     // and is a “left” child of parent
     parent = current;
     if (current.data > n)
     {
            currentIsLeft = true;
            current = current.left;
     }
     else
     {
            currentIsLeft = false;
            current = current.right;
     }

     // If current is null, node n was not found
     if (current == null)
            return false;
}
```

Binary Trees

```
// At this point, current is the node to delete
// Now, we check for the situations

// Situation 1 - leaf node
if (current.left == null && current.right == null)

    // Check if current node is root
    if (parent == current)
          root = null;

    // Check which child pointer of parent to set
    else if (currentIsLeft)
          parent.left = null;
    else
          parent.right = null;
```

Binary Trees

```
// Situation 2 - one child. Parent inherits child
// or if current is root, root takes child
else if (current.left == null)
    if (parent == current)
          root = current.right;
    else if (currentIsLeft)
          parent.left = current.right;
    else
          parent.right = current.right;
else if (current.right == null)
    if (parent == current)
          root = current.left;
    else if (currentIsLeft)
          parent.left = current.left;
    else
          parent.right = current.left;
```

Binary Trees

```
    // Situation 3: two children
    else
    {
         Node successor = getSuccessor(current);

         // Replace current node with successor
         if (parent == current)
                root = successor;
         else if (currentIsLeft)
                parent.left = successor;
         else
                parent.right = successor;

         // Successor will always come from the right, so
         // it must also take deleted node's left child
         successor.left = current.left;
    }
    return true;
}
```

Binary Trees

```
private Node getSuccessor(Node removedNode)
{
   // Prepare successor search by keeping track
   // of parent and current
   Node successorParent = removedNode;
   Node successor = removedNode;
   Node current = successor.right;

   // Starting at the right child of the node to be
   // removed, go down the subtree's left children
   // until there are no more children on the left
   while (current != null)
   {
       successorParent = successor;
       successor = current;
       current = current.left;
   }
```

Binary Trees

```
    // if the successor is somewhere down the subtree,
    // the parent's left child must take the
    // the successor's right child. Then, the
    // successor's right child takes the node
    // to delete's right child (because successor will
    // be replacing it.
    if (successor != removedNode.right)
    {
         successorParent.left = successor.right;
         successor.right = removedNode.right;
    }

    // Note that if the successor is the immediate
    // right child of the node to delete, we just
    // return that node (it has no left children and what
    // ever is on successor's right stays that way even
    // after successor replaces the removed node.
    return successor;
}
```

Binary Trees

An easy way out

- The code is complex as there are many selection statements implying a large number of test cases
- Another approach is to add a property to the node class flagging if the node “is deleted”. There are pros and cons to this:
  - + The complexity of delete is not required
  - + It allows for an easier “undo” of a delete
  - + Useful in situations where delete is not often
  - – Data space will be used indefinitely
  - – Physical removal requires traversing entire tree and recreating with balance

Binary Trees

Update

- The 4th function is update, which requires a search, followed by a change in value
- In order to maintain the order of the structure though, the update will likely require moving the node to a new location in the tree
- The "move" is the equivalent of removing the node, changing its value and re-inserting it back into the tree
- This is much easier to do instead of developing a way for nodes to move around in the tree from a relative position
- The question is efficiency, specifically how well do search, insert and delete perform?

Binary Trees

```
public boolean update(int oldValue,
                      int newValue)
{
  Node n = search(oldValue);
  if (n == null)
     return false;

  remove(oldValue);
  insert(newValue);
  return true;
}
```

Binary Trees

Efficiency

- With a random distribution of adding nodes, search and insert perform at O(log n)
- With delete, a search is performed followed by a series of checks for the different situations. This is at least O(log n)
- In the first 2 situations, the code is constant. The 3rd situation only uses an additional loop to find the successor node
- In a more balanced tree, the number of nodes visited to find the successor is not significant enough to alter the performance category (a worst case successor search is an unbalanced tree leading to O(n) performance anyway which is an already accepted risk)
- Updates will perform a delete and insert making it O(2 log n). This is still logarithmic performance
- Thus, we get the same O(log n) category performance for all 4 functions as sorted arrays and we also get dynamic memory management!

Binary Trees

	Sorted Arrays	Binary Trees
Search	O(log n)	O(log n)
Insert	O(n)	O(log n)
Update	O(n)	O(log n)
Delete	O(n)	O(log n)
	Static memory usage	• Dynamic memory usage • All degrade to O(n) if tree becomes more unbalanced

Binary Trees

Arrays revisited

- With binary trees, the question now is why bother with sorted arrays?
- The code is simpler to implement and use
- Binary trees still have the risk of O(n) performance based on the manner in which data is inserted
- While the categories are the same for performance between binary trees and sorted arrays, array performance is slightly better and more consistent
- Traversing arrays is also easier using the index values (good for report tables). In fact, how would we traverse the elements of a binary tree?

Binary Trees

Tree Traversal

- Say you need to display all the elements in sorted order
- Given an infinite number of different possible trees, the algorithm to traverse a tree requires some thought
- We can start small and work our way to larger trees to find patterns in the logic to develop the algorithm
- Does this process sound familiar…?

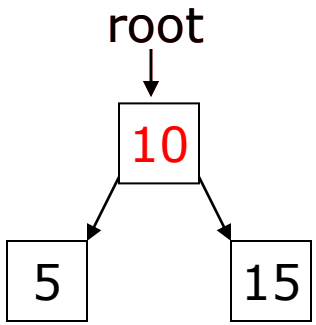
Binary Trees

Tree Traversal

- Start with an empty tree. This may sound redundant, but it helps with the algorithm. Put simply, if there is no tree, don’t display anything (duh!)
- Okay, so big deal. Then, we want to do something if there is a tree of course, right?
- Let’s take a look at a balanced 2 level tree to get an idea of what we do when there is a tree to display

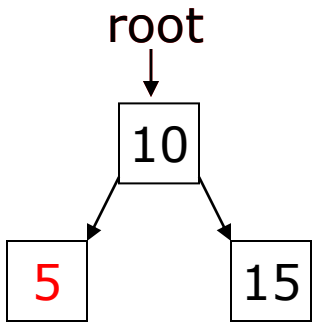
Binary Trees

Remember that all access begins with the root. This would start us at 10



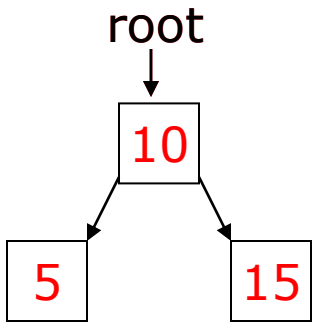
Binary Trees

If we want to show this tree in sorted order, 10 would not be the first number. From looking at it, we know we want to display 5 first. However, how do we state this in logical coding terms?



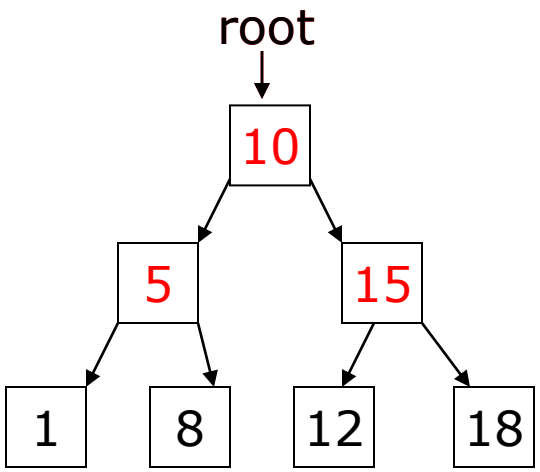
Binary Trees

One way to state it is, “if there is a node to the left, display the node’s value”. This is then followed by, “then show my value and then if there is a node to the right, show its value”



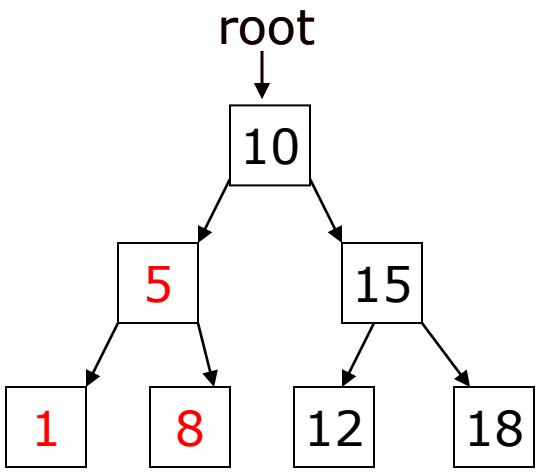
Binary Trees

However, trees are not always 2 levels. That logic is incomplete starting at node 10



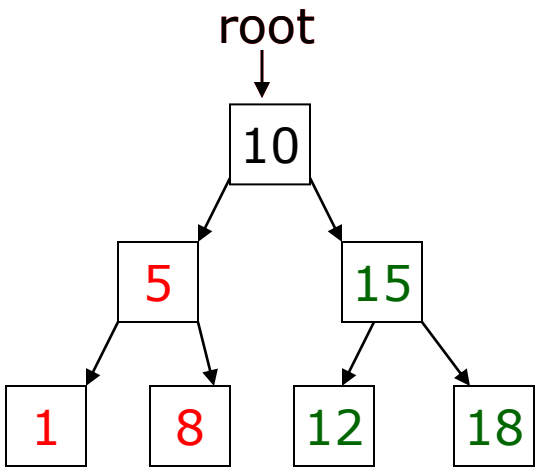
Binary Trees

If we look at the subtree 1-5-8, that does fit our logic of display left node, then current, then right



Binary Trees

From the perspective of node 10, we can then modify our logic to from "display the node to the left, display current, display the node to the right" to "display the subtree to the left, display current, display the subtree to the right"



Binary Trees

Back to recursion!

- It turns out the code for traversing a tree is a very “simple” form of recursion. As before, we need our base case and inductive case
- The redundant statement of the obvious a few slides back turns out to be the base case - If there’s no tree, don’t do anything
- Otherwise, the inductive case is used:
  - Display the tree on the left
  - Print the current node’s value
  - Display the tree on the right
- As you would guess, the code easily follows this logic

Binary Trees

```
public void display()
{
   displayInOrder(root);
}

void displayInOrder(Node current)
{
   if (current != null)
   {
       displayInOrder(current.left);
       System.out.println(current.data);
       displayInOrder(current.right);
   }
}
```

Binary Trees

Traversals

- Note that because the code uses recursion, the function to display the tree actually makes the call to the recursive function using root as the parameter
- This is the most popular form of traversing a tree. Imagine writing the code to traverse a tree without recursion. It's not impossible, but certainly the recursive case is a lot easier to code (once the recursive logic is understood of course!)
- This logic can also be used to take advantage of other types of traversals. Imagine taking the same 3 level tree and displaying the current node first followed by the recursive calls to display the trees on the left, then right

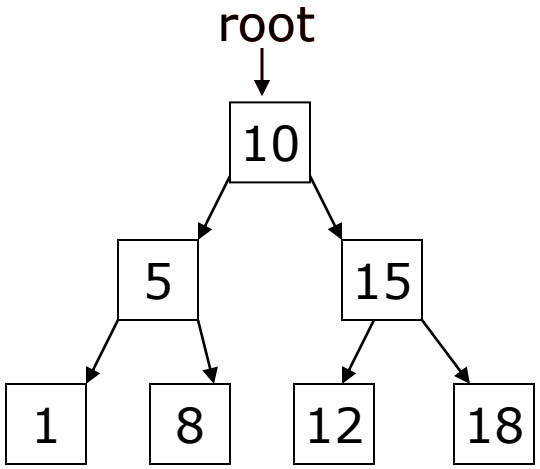
Binary Trees

```
void displayPreOrder(Node current)
{
  if (current != null)
  {
     System.out.println(current.data);
     displayPreOrder(current.left);
     displayPreOrder(current.right);
  }
}
```

Binary Trees

The output of doing this traversal is:

10 5 1 8 15 12 18



This traversal is called “preorder”. Another approach is “postorder”. Display the left and right subtrees first and then print the current node

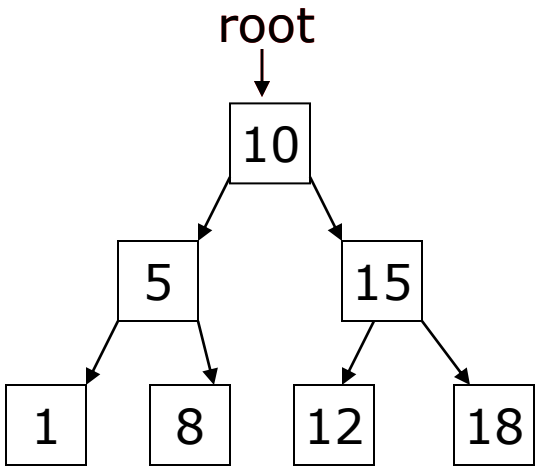
Binary Trees

```
void displayPostOrder(Node current)
{
  if (current != null)
  {
     displayPostOrder(current.left);
     displayPostOrder(current.right);
     System.out.println(current.data);
  }
}
```

Binary Trees

The output of doing postorder traversal is:

1 8 5 12 18 15 10



Binary Trees

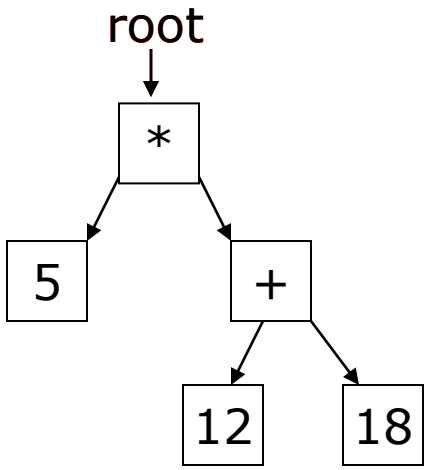
Traversals

- Why bother with pre and post order traversals?
- Remember that just because the data in the tree is ordered, it doesn’t necessarily mean the data is sorted by values
- Information can be inserted into the tree to follow a particular order as well
- Observe the following tree…

Binary Trees

Preorder traversal: * 5 + 12 18

Postorder: 5 12 18 + *



Both traversals present a calculation notation that can be used to solve equations entered into a tree

Binary Trees

Summary

- Binary trees merge the best of both worlds with sorted arrays and dynamic memory management
- The code is more complex, but the resulting performance is comparable
- The primary concern with binary trees is the potential for the tree to degrade into a linked list
- Our next topic continues with the tree type structure, while emphasizing balance