CMPT 225: Data Structures & Programming
– Unit 23 –
Binary Search Trees

Dr. Jack Thomas

Simon Fraser University

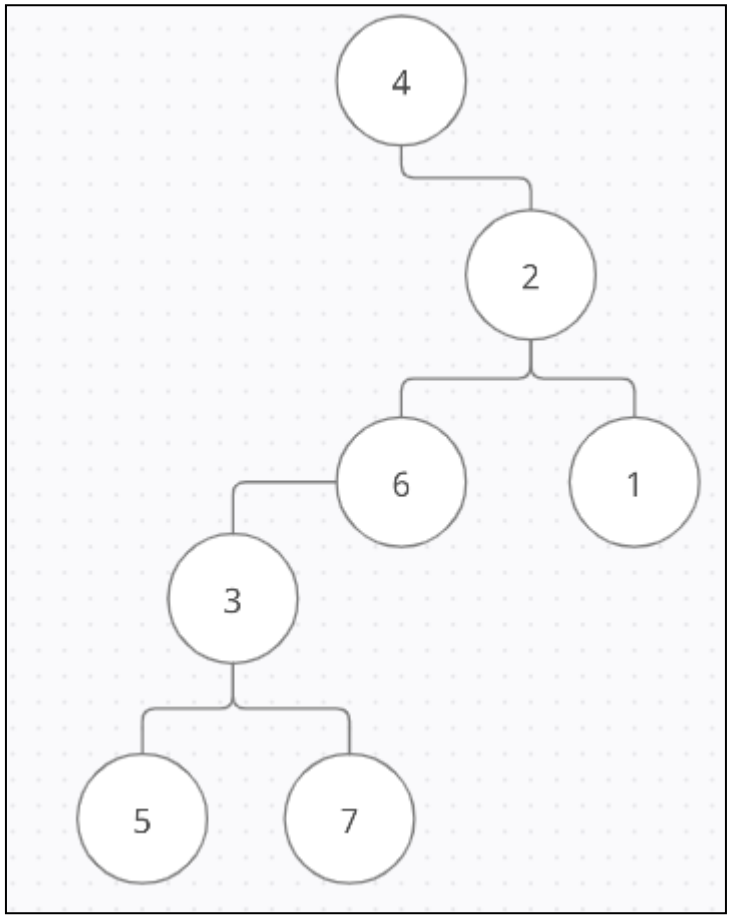
Spring 2021

Today's Topics

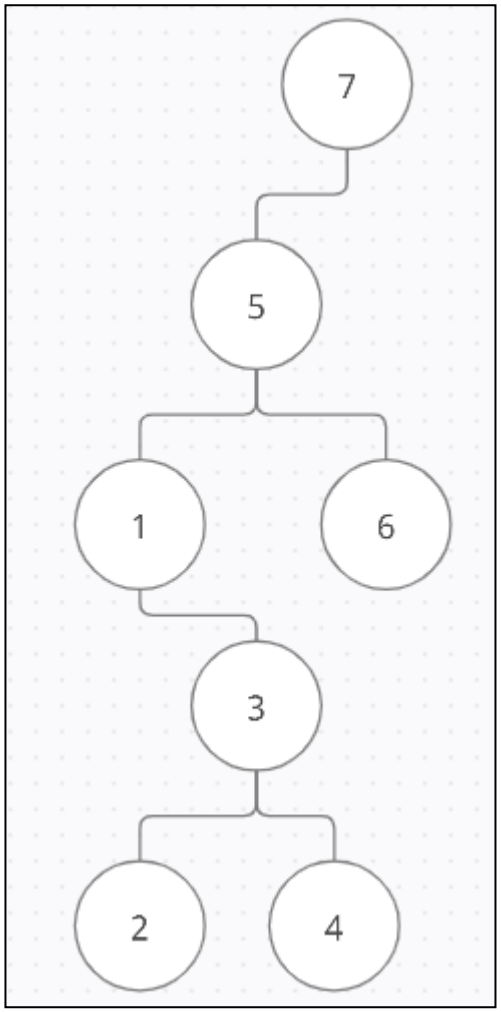
- Reviewing Binary Search Trees
- Tree Search
- Insertion and Removal in a BST
- Java Implementation for BST

Remember Trees?

- How about **Binary Trees**?

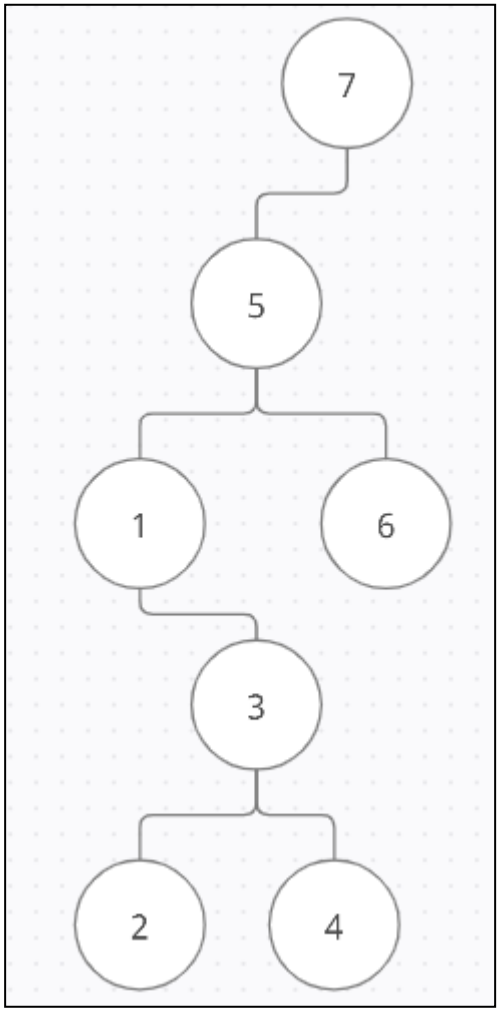


- How about **Binary Search Trees**?



To Refresh Your Memory…

- A **Binary Tree** restricts each node of a tree to at most two children.
- A **Binary Search Tree** also orders the children such that, for every node in the tree, the value of the element in their “left” child (and all of that child’s descendants) is less than that node’s element, while the value of the element in their “right” child (and all of that child’s descendants) is greater.

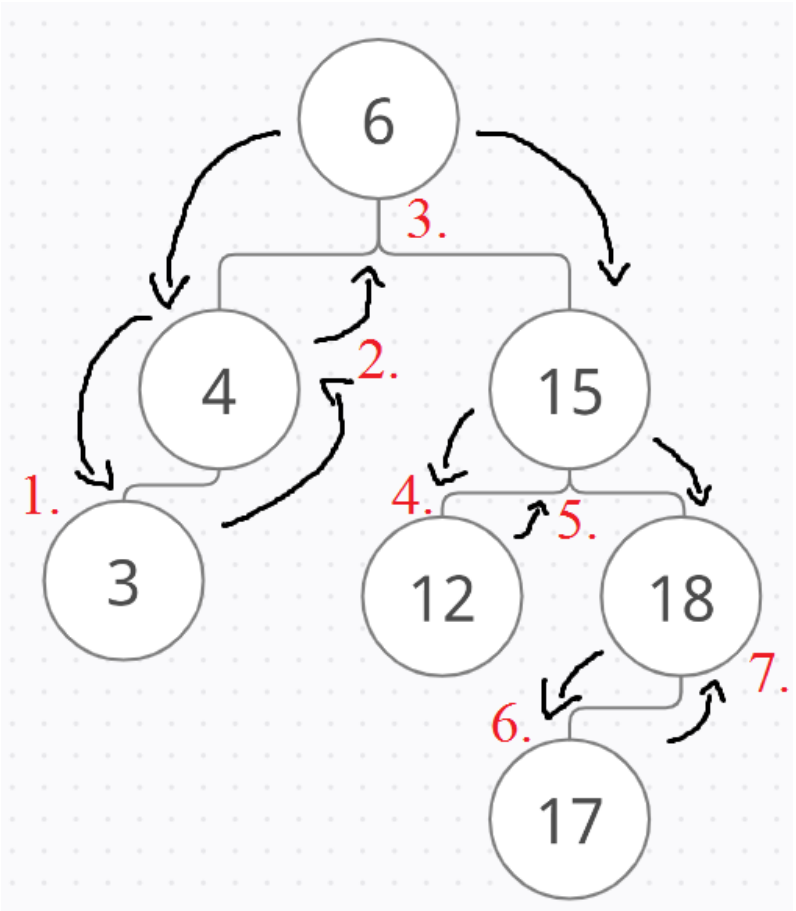


A Binary Search Made Into A Structure

- Like the Skip List, a Binary Search Tree (**BST**) is a way to benefit from **the insight behind Binary Search algorithms** in the building of our structure.
- Makes them suitable for **implementing Ordered Maps and Dictionaries** efficiently.

Extracting an Order

- An **in-order traversal**, where we visit a node's left child, then the node itself, then the right child, will allow us to visit every node in our BST **in order from smallest to largest key**.



Tree Search

- We can **search a BST** for a given key and return the matching value with an **in-order traversal algorithm**.
- To simplify our algorithms going forward, we're going to **leave external nodes blank** rather than use them to store entries.

```
Algorithm TreeSearch(k,v):
        if T.isExternal(v) then
                return v
        if k < key(v) then
                return TreeSearch(k, T.left(v))
        else if k > key(v) then
                return TreeSearch(k, T.right(v))
        return v
```

Analyzing BST Searching

- Each recursive call of our search algorithm for a BST uses only primitive operations, so our **run-time** will be dependent on the **height of the tree**.
- Unfortunately, this is O(h), not O(log n), because this isn't a complete binary tree (yet), so we **can't guarantee anything about the height**. Still, better than O(n)! Probably?

Insertion

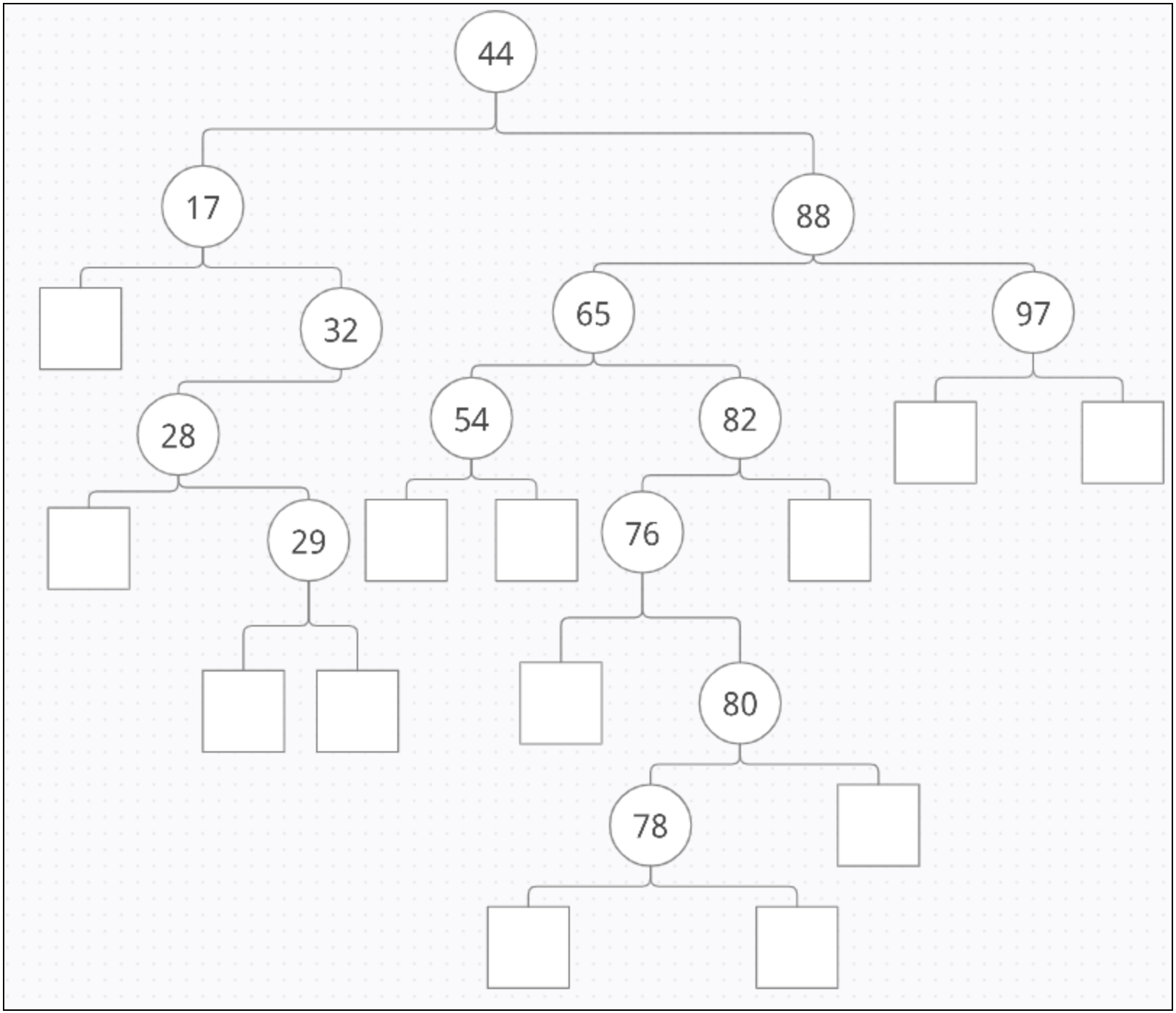
- Our insertion needs to find the external node the new node would belong to and turn it into an internal node.

Algorithm TreeInsert(k,x,v):
Input: A search key k, an associated value x, and a node v of T
Output: A new node w in the subtree T(v) that stores the entry (k,x)

```
w<-TreeSearch(k,v)
if T.isInternal(w) then
        return TreeInsert(k,x,T.left(w))
T.insertAtExternal(w,(kx))
return w
```

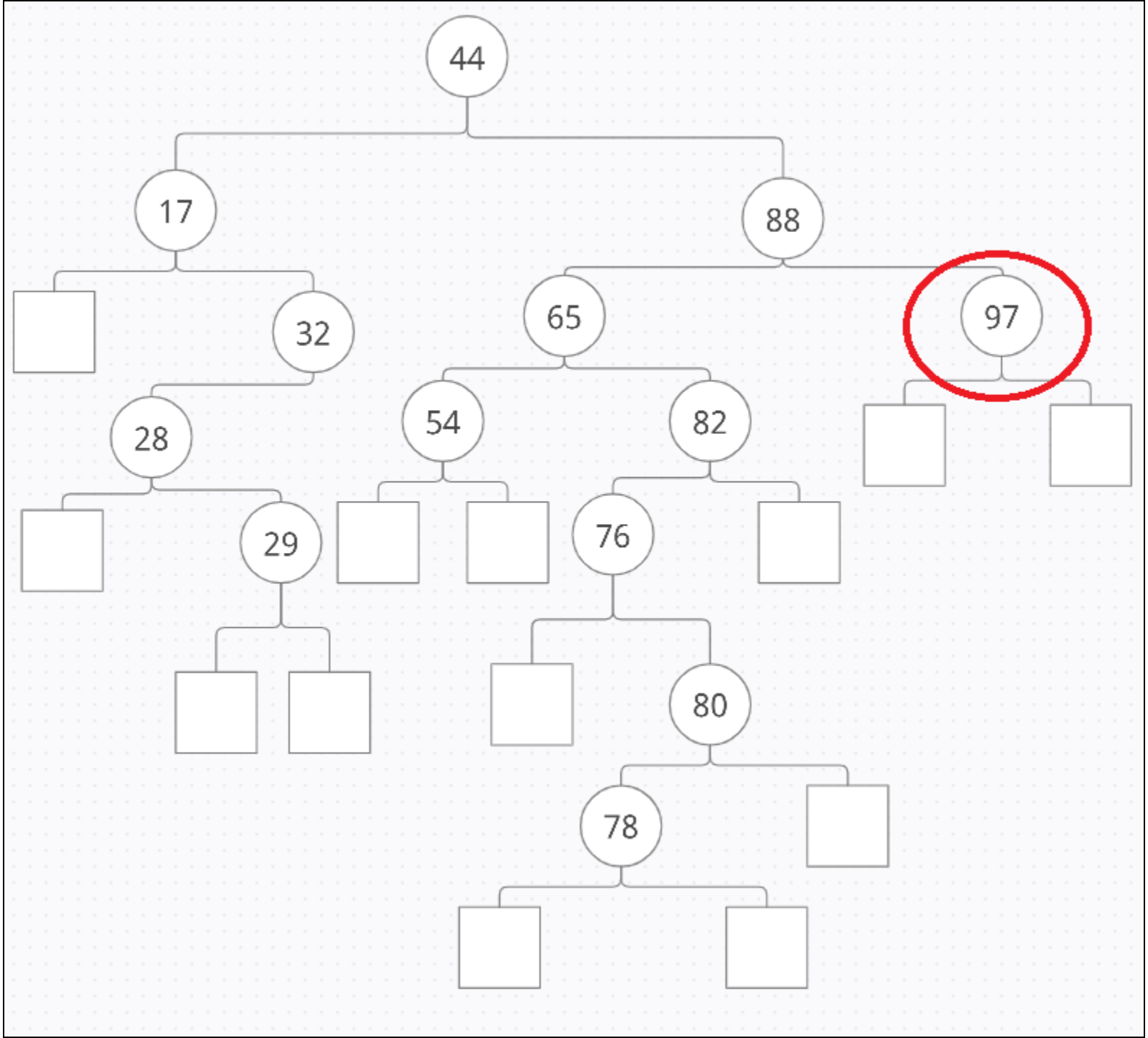
Removal

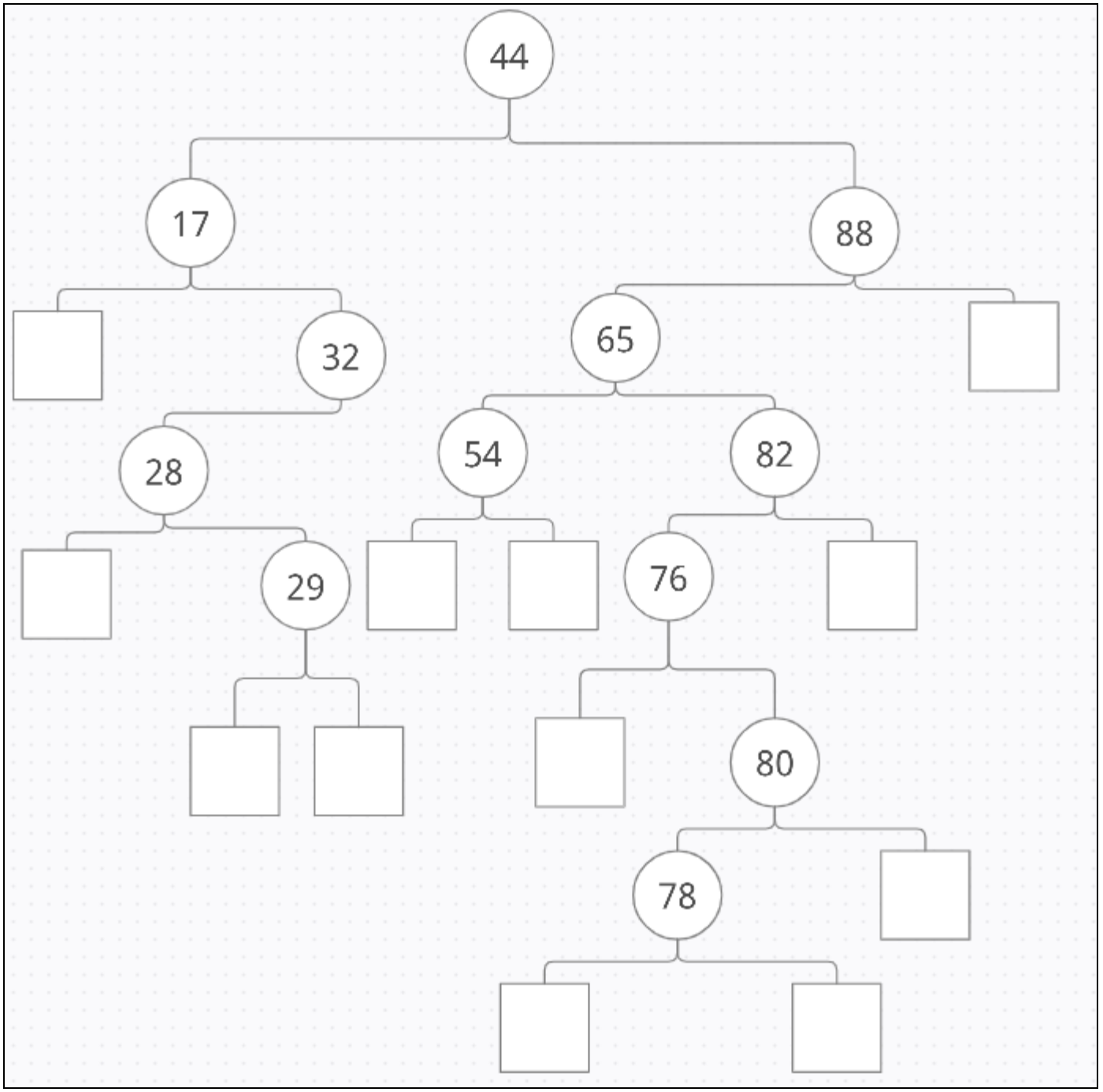
- Removing a node is more complicated, **depending on the removed node's children**.
- You can find the node to remove with a Tree Search, but how to remove it differs for each of **three cases**:
  1. If the node has **two external children**.
  2. If the node has **one external and one internal child**.
  3. If the node has **two internal children**.


44
17
88
32
65
97
28
54
82
29
76
80
78

Removing a Node With Two External Children

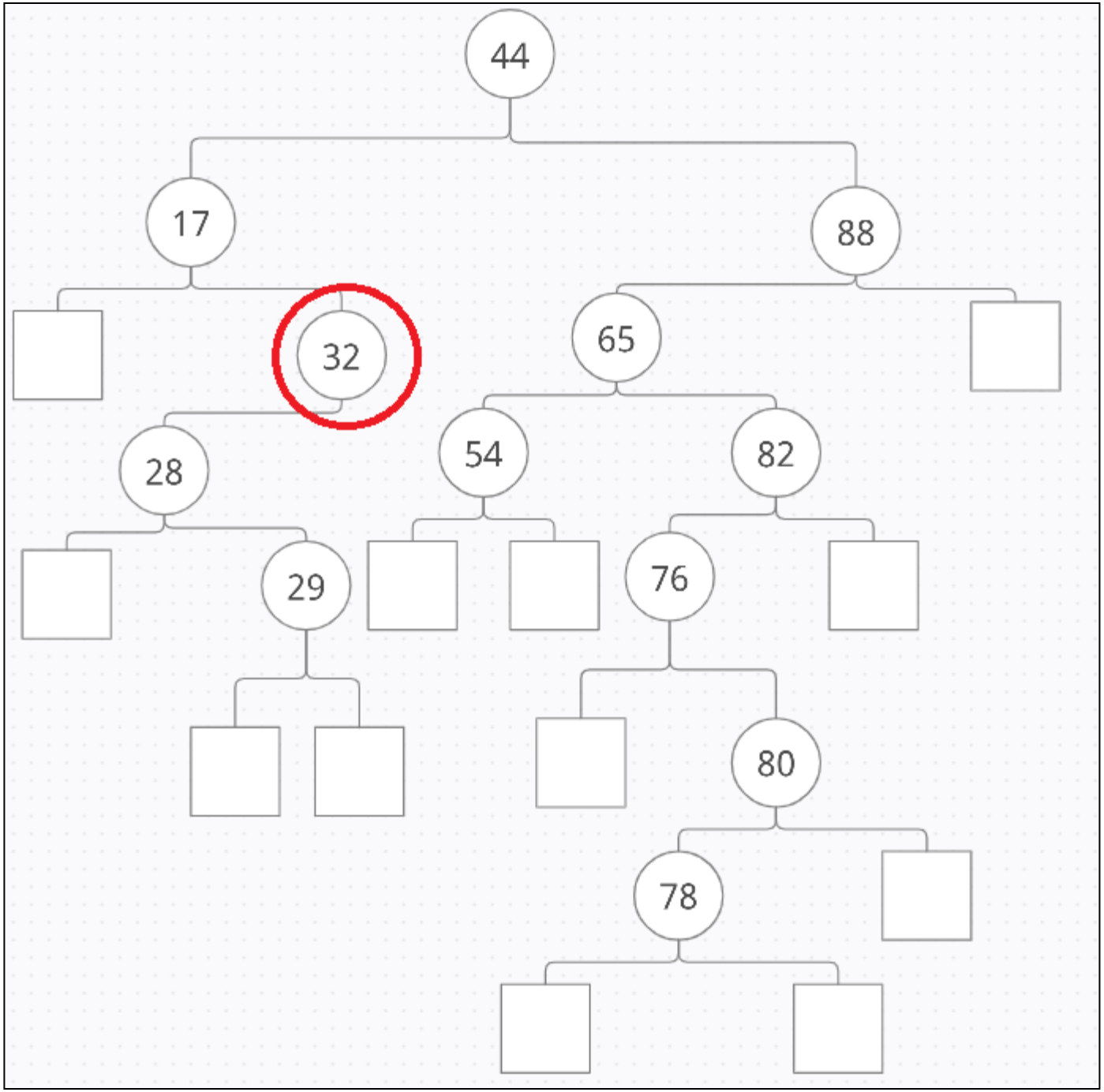
- Easy – we can just null the external nodes and **convert the node being removed to an external node**, no loose ends.
- Converting a node from internal to external is as simple as **setting its element and children to null**, while retaining the connection to its parent (and the parent's connection to its now external node child).


44
17
88
32
65
97
28
54
82
29
76
80
78

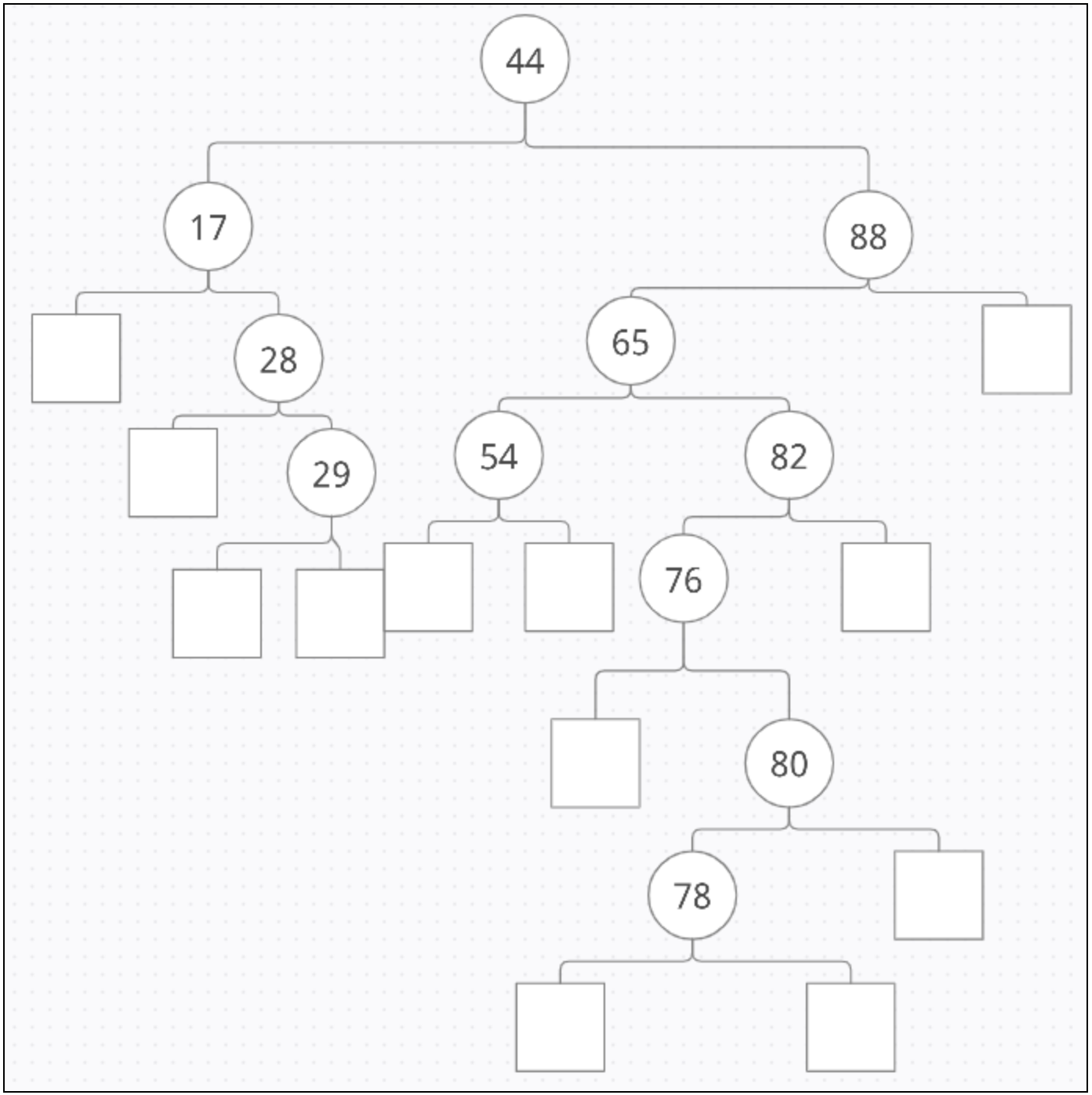

44
17
88
32
65
28
54
82
29
76
80
78

Removing a Node With One Internal and One External Child

- A little more work – the external node can be ignored, while **the internal node takes the place of the node that was removed**.
- It makes no difference whether the one internal child is a left child (smaller than their parent) or right child (larger), since we know **either will be a valid child** of the same type as their parent was to their grandparent.



(Argh I forgot to draw in this node's right external node child, just imagine another empty box there. It doesn't make a difference here anyway.)

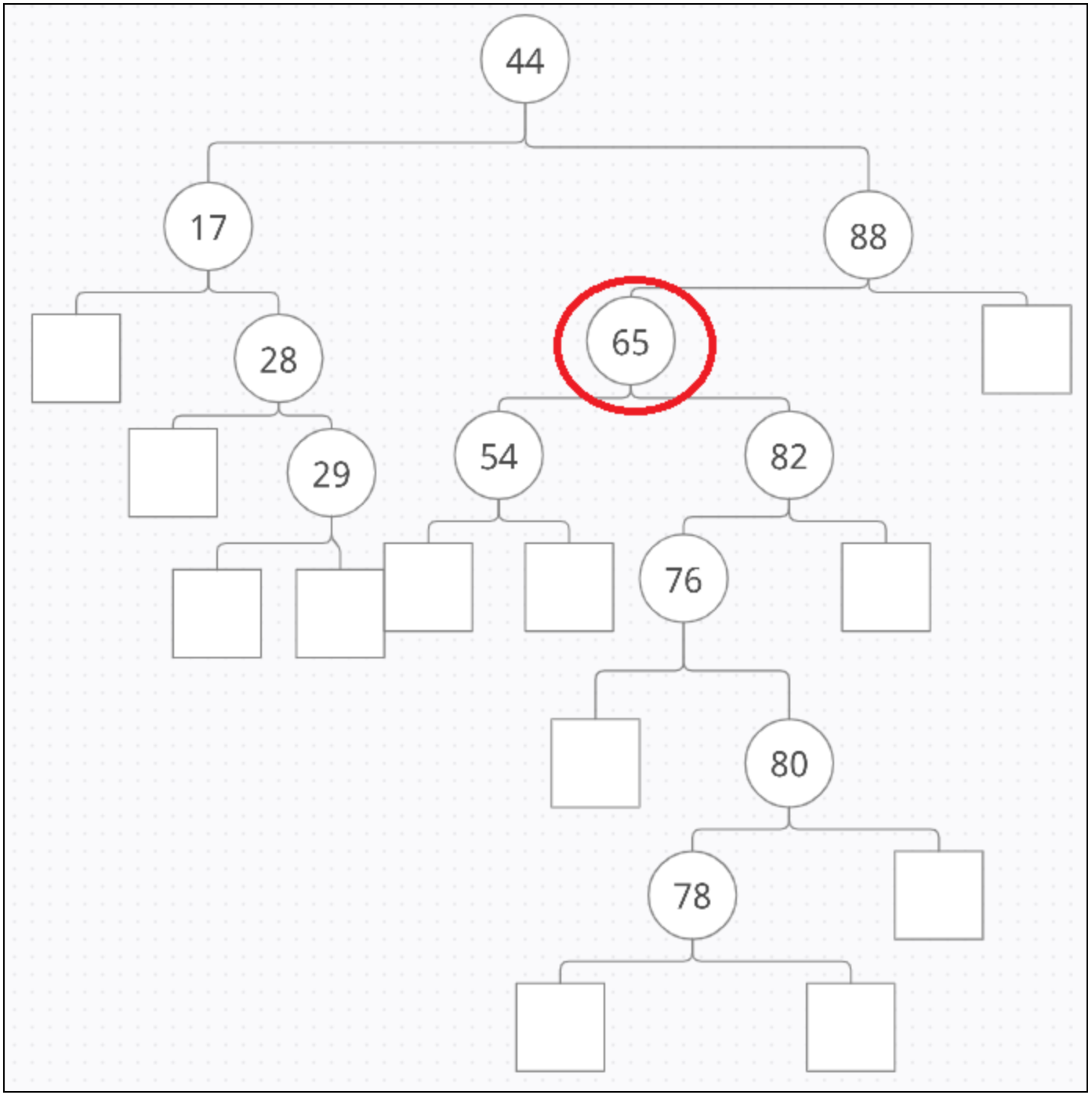

44
17
88
28
65
29
54
82
76
80
78

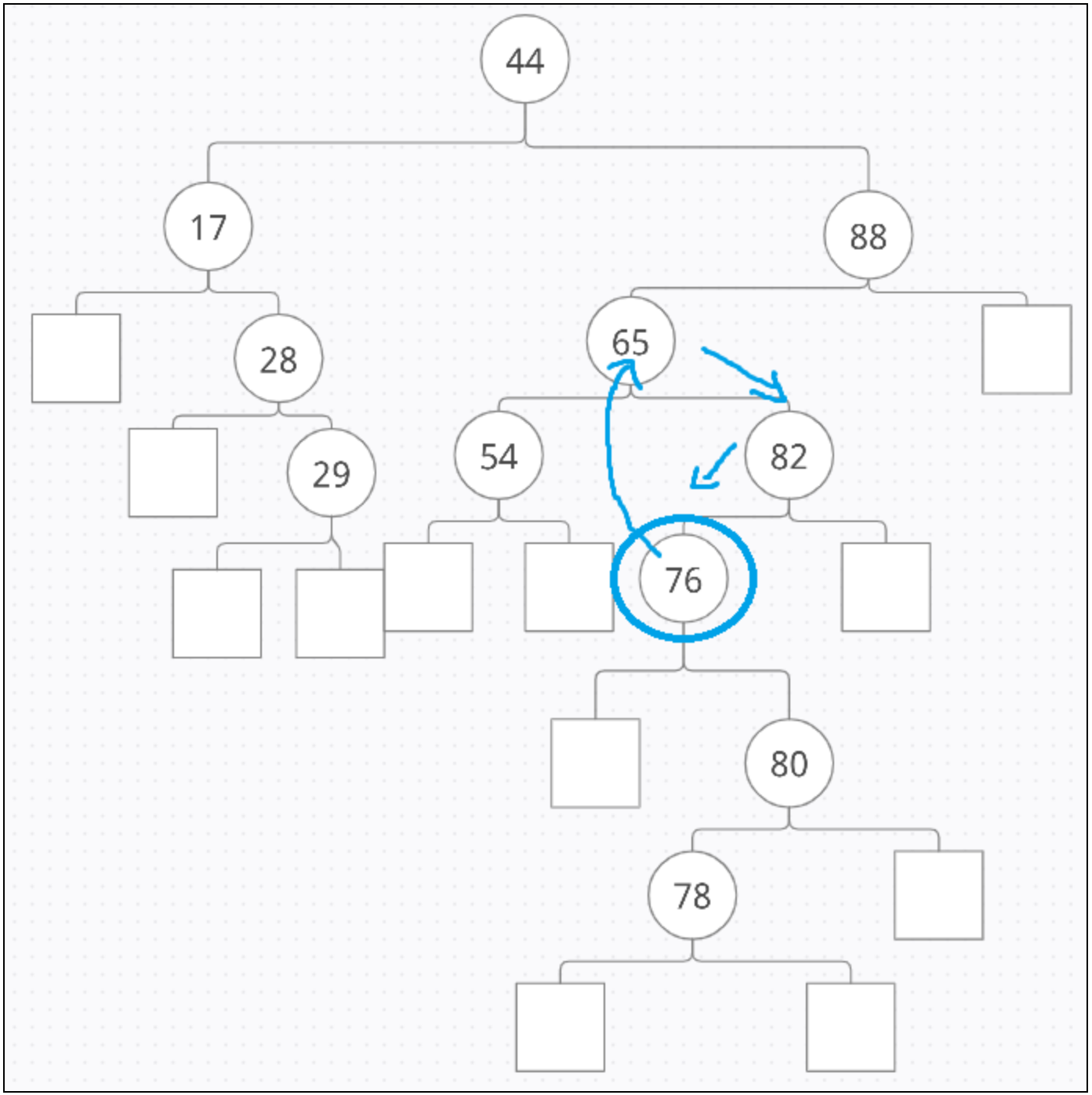
The RemoveExternal Helper Function

- A common helper-function for the case where you're removing a node with one internal and one external child is **RemoveExternal**, which takes that external child as a parameter.
- This function takes advantage of the external node's relative relationships to **connect their grandparent to their sibling**, cutting out their parent (and themselves).
- The reason we use the external node to perform this operation is so that our overall **Remove function can call this function** when it finds an external node.

Removing a Node With Two Internal Children

- The tricky one.
- While either child would be a valid child to the removed node's parent, **how do their descendants mesh together**?
- We'll run an **in-order traversal** to find the **leftmost descendant** of the removed node's **right child**.
- The "leftmost descendant" is defined as the internal node you find from the removed node's right child if you keep moving left.


44
17
88
28
65
29
54
82
76
80
78

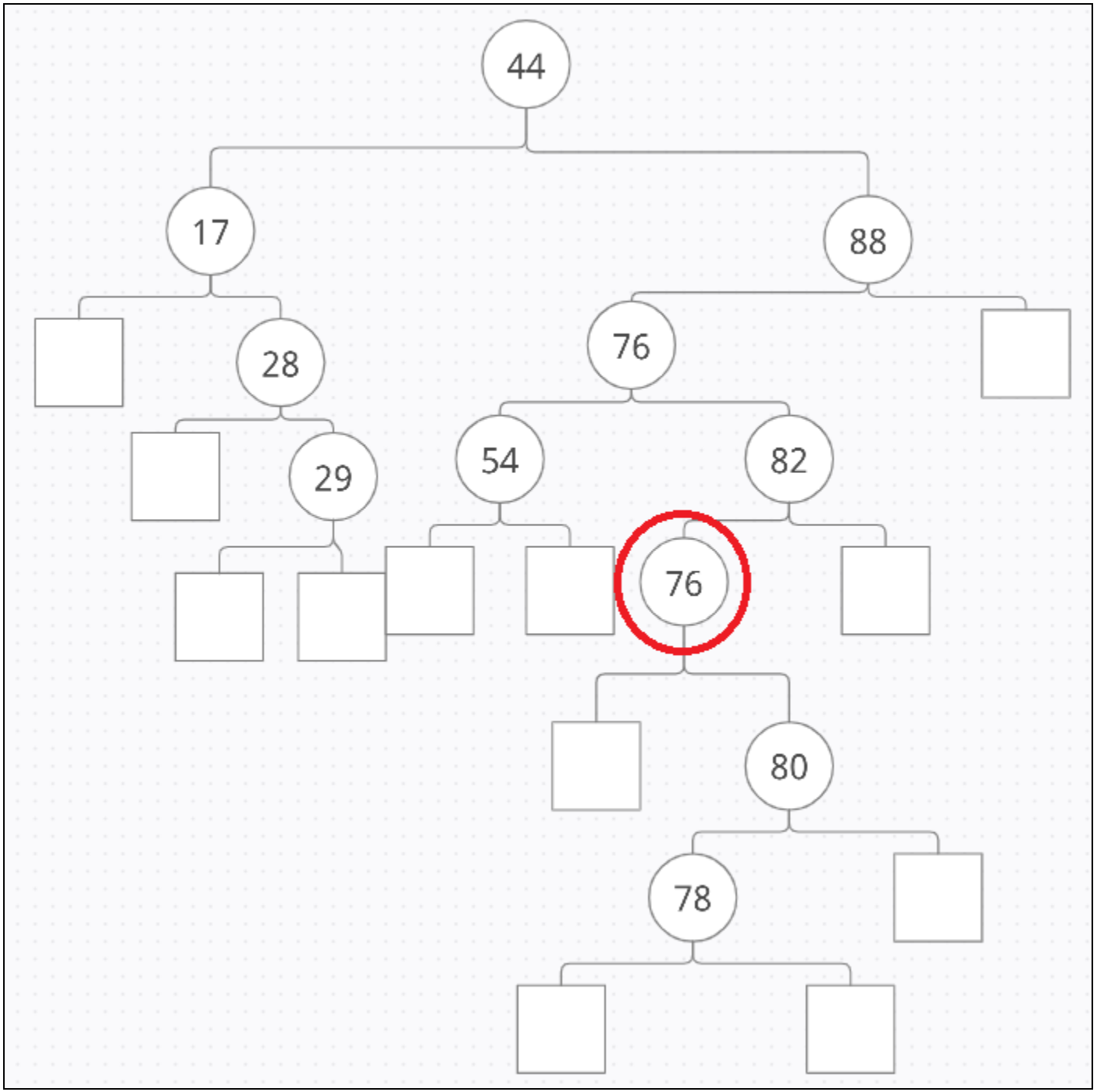

44
17
88
28
65
29
54
82
76
80
78

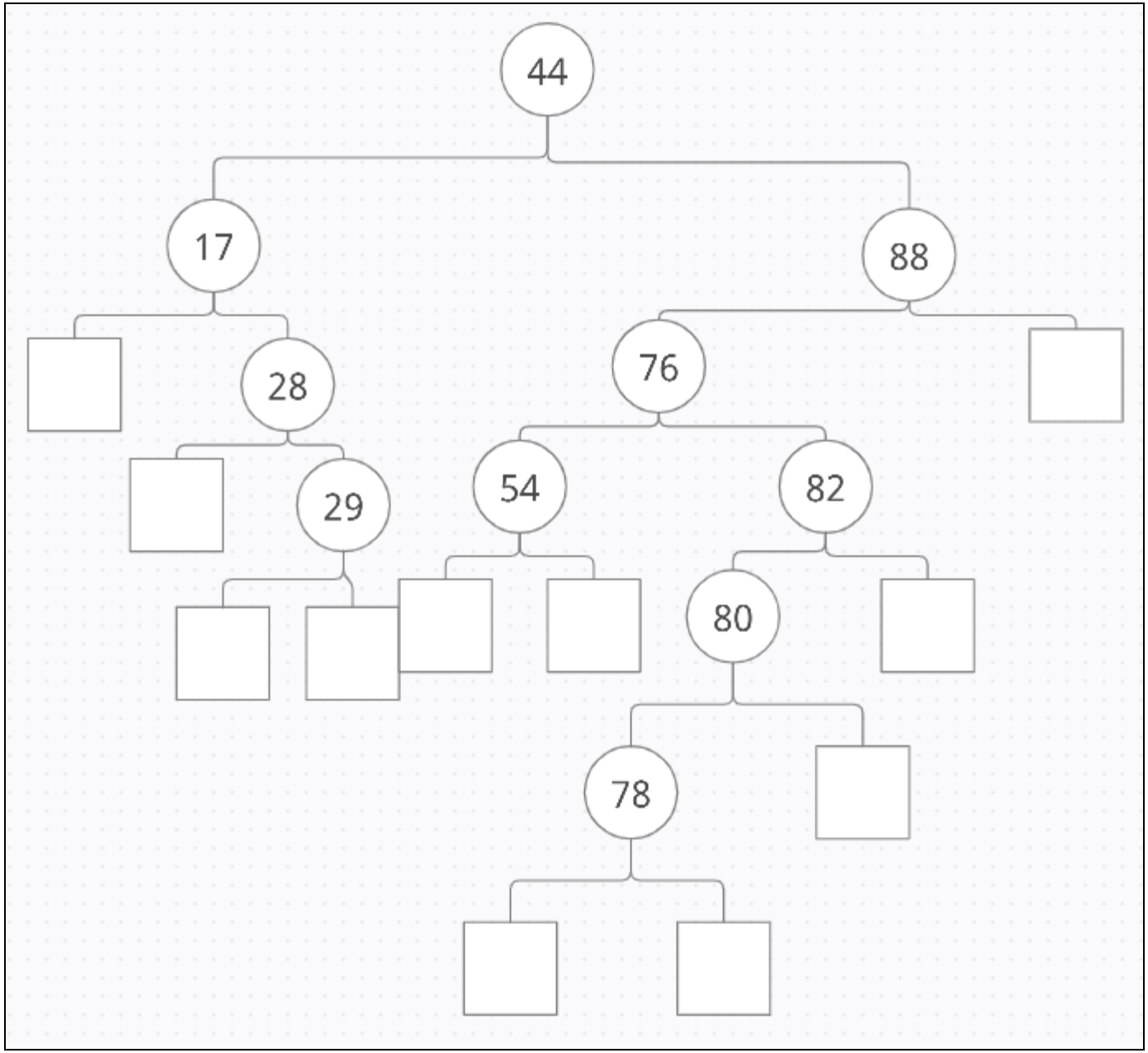
Removing a Node With Two Internal Children

- Why 76? Why not 78?
  - 76 is the leftmost child of 65's right subtree. To get to 78, we'd have first had to go right once to 80, and we can't move right.
- What if 76 had an internal left child of its own?
  - We're doing an inorder traversal, so we'd have moved to 76's left child first before "visiting" 76, so that's the one we'd be using instead.
- What if 82'd had no internal children, or just a right child?
  - Then 82'd be the leftmost child and we'd be using it.

Removing a Node With Two Internal Children

- Next, we **swap the element** from the leftmost child into the node we're supposed to remove, overwriting it.
- The "removed" node is now effectively gone, and the value we swapped in its place will be **valid for both children and the parent**.
- The last step, then, is actually **removing the leftmost child node** whose value we swapped out, by treating it as a node with one or zero internal node children.


44
17
88
28
76
29
54
82
76
80
78


44
17
88
28
76
29
54
82
80
78

Where's the ADT?
The Standard Java Class?

- Binary Search Trees are a constrained form of Tree, but they're **not supported as distinct Abstract Data Types or as standard classes**.
- Partly, this is because Binary Search Trees are more like a **set of properties** than they are a whole and distinct class – in the same way that a Complete Binary Tree is also a variant of the basic tree.
- That means it's up to you to implement one.

Java Implementation

- We don’t have all day to grind this one out, but I’ll get you started with **TreeSearch** for a Binary Search Tree of BSTNodes with int keys.

```
public BSTNode TreeSearch(int input, BSTNode current) {
    if (isExternal(current)) {
        return current;
    } else {
        if (current.element > input) {
            return TreeSearch(input, current.leftChild);
        } else if (current.element < input) {
            return TreeSearch(input, current.rightChild);
        } else {
            return current;
        }
    }
}
```

Recap – The External Slide

- **Binary Search Trees** are a form of Binary Tree constrained by the left child's key being smaller than the parent and the right child's key being larger.
- The add and remove functions are built around **Tree Search**, which efficiently navigates the structure since it's already laid out for in-order traversals.
- By building a binary search into its structure, it may provide us more efficient searching opportunities by bounding searches to the height of the Tree, **but so far there's nothing stopping h = n.**
- There is **neither an ADT nor standard Java support** for BSTs, requiring manual implementation.