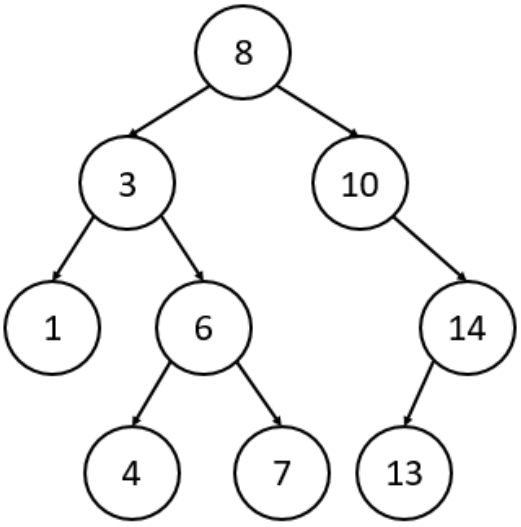
Binary Search Trees

Step 1

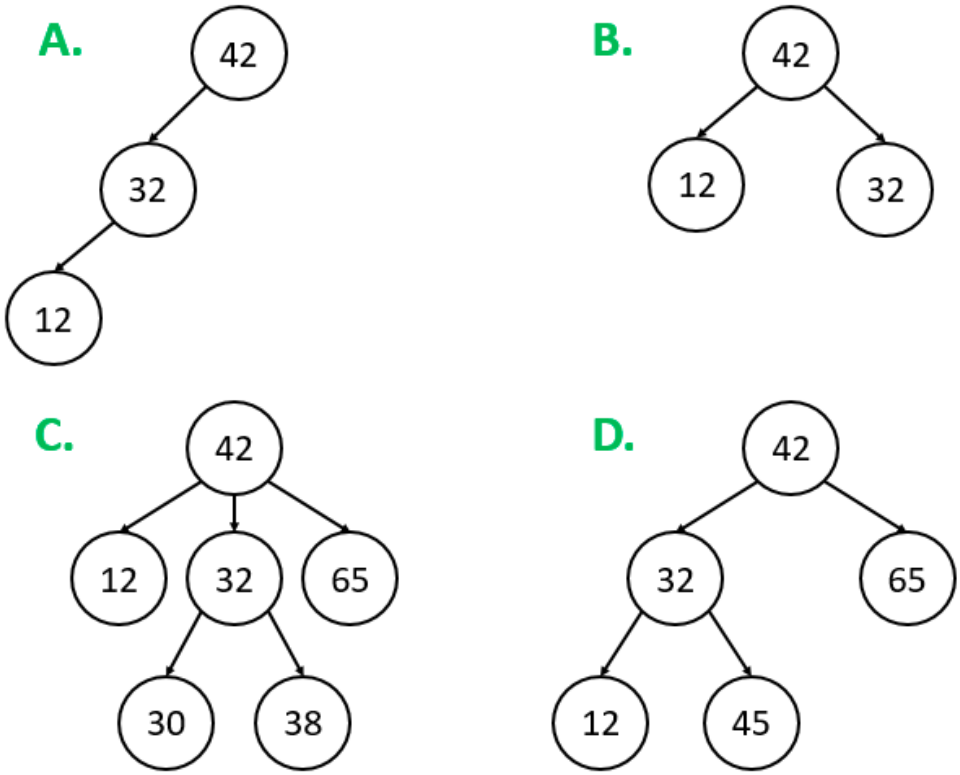
In the previous section, we were motivated by the goal of storing a set of elements in such a manner that we could quickly access the *highest priority* element in the set, which led us to the **Heap** data structure (and its use to implement the **Priority Queue** ADT). However, although we obtained fast access to the highest priority element, we had no way of efficiently looking at any other element. What if, instead, we want a data structure that could store a set of elements such that we could find any arbitrary element fairly quickly?

The second type of binary tree we will discuss is the **Binary Search Tree (BST)**, which is a rooted binary tree (i.e., there is a single "root" node, and all nodes have either 0, 1, or 2 children) in which any given node is larger than all nodes in its left subtree and smaller than all nodes in its right subtree. As can be inferred, a **BST** can only store elements if there exists some way of comparing them (e.g. strings, characters, numbers, etc.): there must be some inherent order between elements. We will be learning about the following functions of a **BST**: **find**, **size**, **clear**, **insert**, **empty**, **successor**, and the **iterator** pattern. Below is an example of a typical valid **Binary Search Tree**:



Step 2

EXERCISE BREAK: Which of the following are valid **Binary Search Trees**? (Select all that apply)



To solve this problem please visit https://stepik.org/lesson/28727/step/2

Step 3

EXERCISE BREAK: What is the space complexity of a **Binary Search Tree** with *n* elements?

To solve this problem please visit https://stepik.org/lesson/28727/step/3

Step 4

Below is pseudocode for the basic **BST** functions. They should be somewhat intuitive given the structure of a **BST**. In these examples, we assume that a **BST** object has two global variables: the *root node* (`root`), and an integer keeping track of the *size* of the BST (`size`). Note that pseudocode only represents the logic behind the algorithms, and when you actually implement these functions, you may need to do worry about language-specific things (e.g. syntax, memory management).

```
find(element): // returns True if element exists in BST, otherwise returns False
    current = root                  // start at the root
    while current != element:
        if element < current:       // if element < current, traverse left
            current = current.leftChild
        else if element > current: // if element > current, traverse right
            current = current.rightChild
        if current == NULL:         // if we traversed and there was no such child, failure
            return False
    return True // we only reach here if current == element, which means we found element
```

```
insert(element): // inserts element into BST and returns True on success (or False on failure)
    current = root                          // start at the root
    while current != element:
        if element < current:
            if current.leftChild == NULL:  // if no left child exists, insert element as left
child
                current.leftChild = element
                return True
            else:                           // if a left child does exist, traverse left
                current = current.leftChild
        else if element > current:
            if current.rightChild == NULL: // if no right child exists, insert element as right
child
                current.rightChild = element
                return True
            else:                           // if a right child does exist, traverse right
                current = current.rightChild
    return False // we only reach here if current == element, and we can't have duplicate elements
```

```
size(): // returns the number of elements in BST
    return size
```

```
clear(): // clears BST
    root = NULL
    size = 0
```

```
empty(): // returns True if BST is empty, otherwise returns False
    if size == 0:
        return True
    else:
        return False
```

Step 5

CODE CHALLENGE: Printing the Elements of a Binary Search Tree in Sorted Order

We have defined the following C++ class for you:

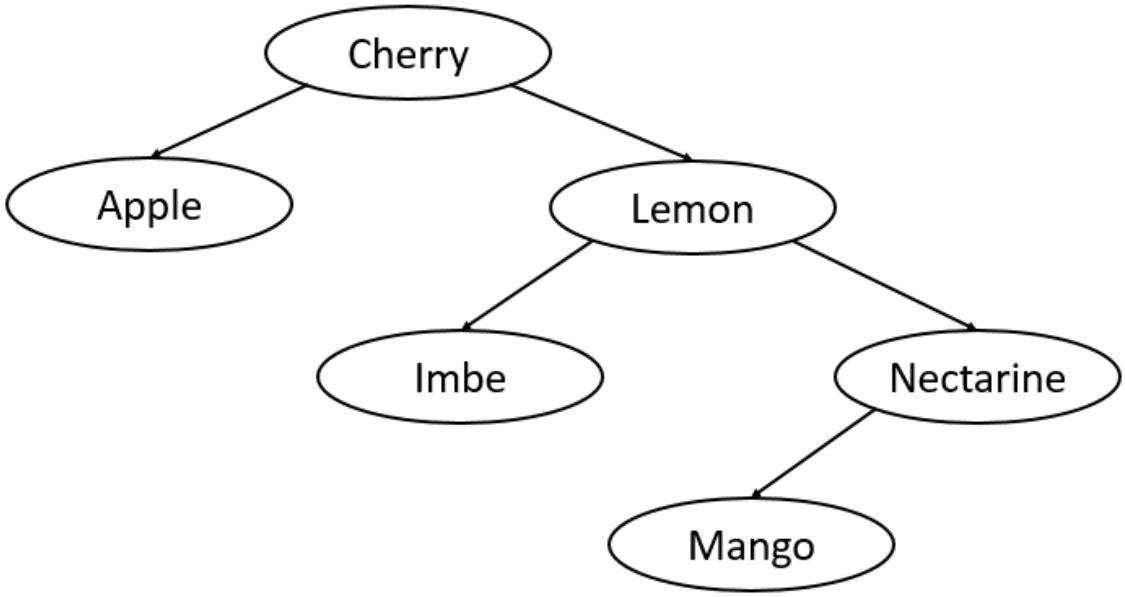
```
class Node {
    public:
        string label;
        Node* leftChild = NULL;
        Node* rightChild = NULL;
};
```

Write a function `sortedPrint(Node* node)` that recursively prints the labels of all nodes in the subtree rooted at `node` in ascending sorted order. For example, if we had a tree with some root node `root`, then calling `sortedPrint(root)` would print the labels of all nodes in the tree in ascending sorted order. Print a single label per line. You can assume that the **Binary Search Tree** structure is valid (i.e., all nodes are larger than all of their descendants in their left subtree and smaller than all of their descendants in their right subtree).

HINT: Judging by the structure of a **Binary Search Tree**, does one of the rooted binary tree traversal algorithms (**pre-order**, **in-order**, and **post-order**) seem useful in this context?

HINT: What happens if one of the two child pointers is `NULL`?

The tree represented by the Sample Input has been reproduced below for clarity:



Sample Input:

```
Cherry -> Apple
Cherry -> Lemon
Lemon -> Imbe
Lemon -> Nectarine
Nectarine -> Mango
```

Sample Output:

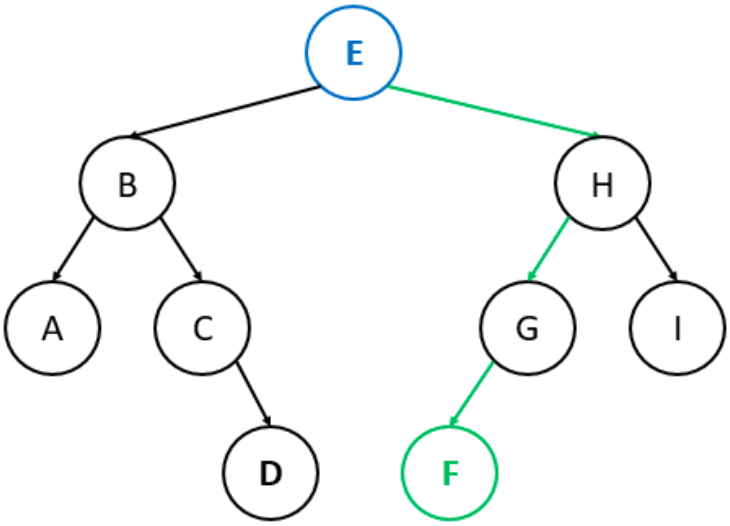
```
Apple
Cherry
Imbe
Lemon
Mango
Nectarine
```

To solve this problem please visit https://stepik.org/lesson/28727/step/5

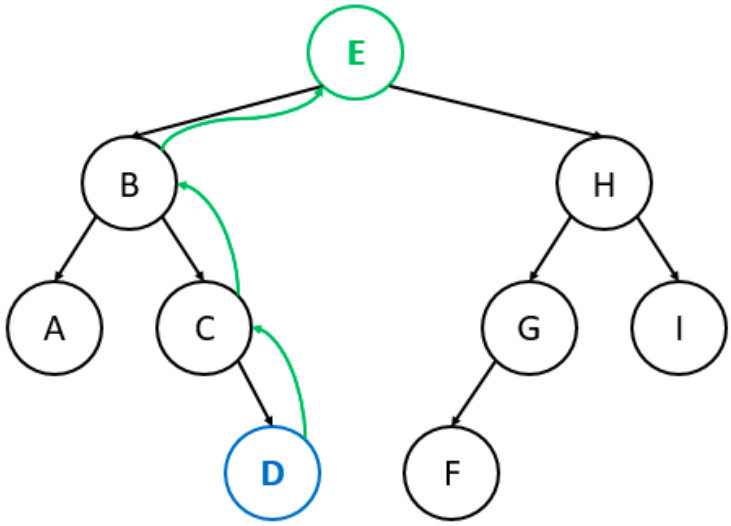
Step 6

The **successor** function of a **Binary Search Tree** (or of a node, really) finds the "successor"—the next largest node—of a given node in the **BST**. We can break this function down into two cases:

Case 1: If node *u* has a right child, we can somewhat trivially find *u*'s successor. Recall that, by definition, a node is smaller than all of the nodes in its right subtree. Therefore, if *u* has a right child, *u*'s successor must be in its right subtree. Note that *u*'s successor is not necessarily its right child! Specifically, if *u* has a right child, *u*'s successor must be the smallest node in its right subtree. Thus, to find *u*'s successor, we can traverse to *u*'s right child and then traverse as far left as possible (What does this repeated left traversal do?). In the example below, we will find D's successor:



Case 2: If node *u* does not have a right child, we have to be a bit more clever. Starting at *u*, we can traverse up the tree. The moment we encounter a node that is the left child of its parent, the parent must be *u*'s successor. If no such node exists, then *u* has no successor. In the example below (slightly different than the above example), we will find D's successor:




```
successor(u): // returns u's successor, or NULL if u does not have a successor
    if u.rightChild != NULL:             // Case 1: u has a right child
        current = u.rightChild
        while current.leftChild != NULL: // find the smallest node in u's right subtree
            current = current.leftChild
        return current
    else:                                // Case 2: u does not have a right child
        current = u
        while current.parent != NULL:    // traverse up until current node is its parent's left
child
            if current == current.parent.leftChild:
                return current.parent
            else:
                current = current.parent
        return NULL // we have reached the root and didn't find a successor, so no successor
exists
```

Step 7

Now that we have determined how to find a given node's successor efficiently, we can perform two important **BST** functions.

The first function we will describe is the **remove** function. To remove a node *u* from a **BST**, we must consider three cases. First, if *u* has no children, we can simply delete *u* (i.e., have *u*'s parent no longer point to *u*). Second, if *u* has a single child, have *u*'s parent point to *u*'s child instead of to *u* (Why does this work?). Third, if *u* has two children, replace *u* with *u*'s successor and remove *u*'s successor from the tree (Why does this work?).

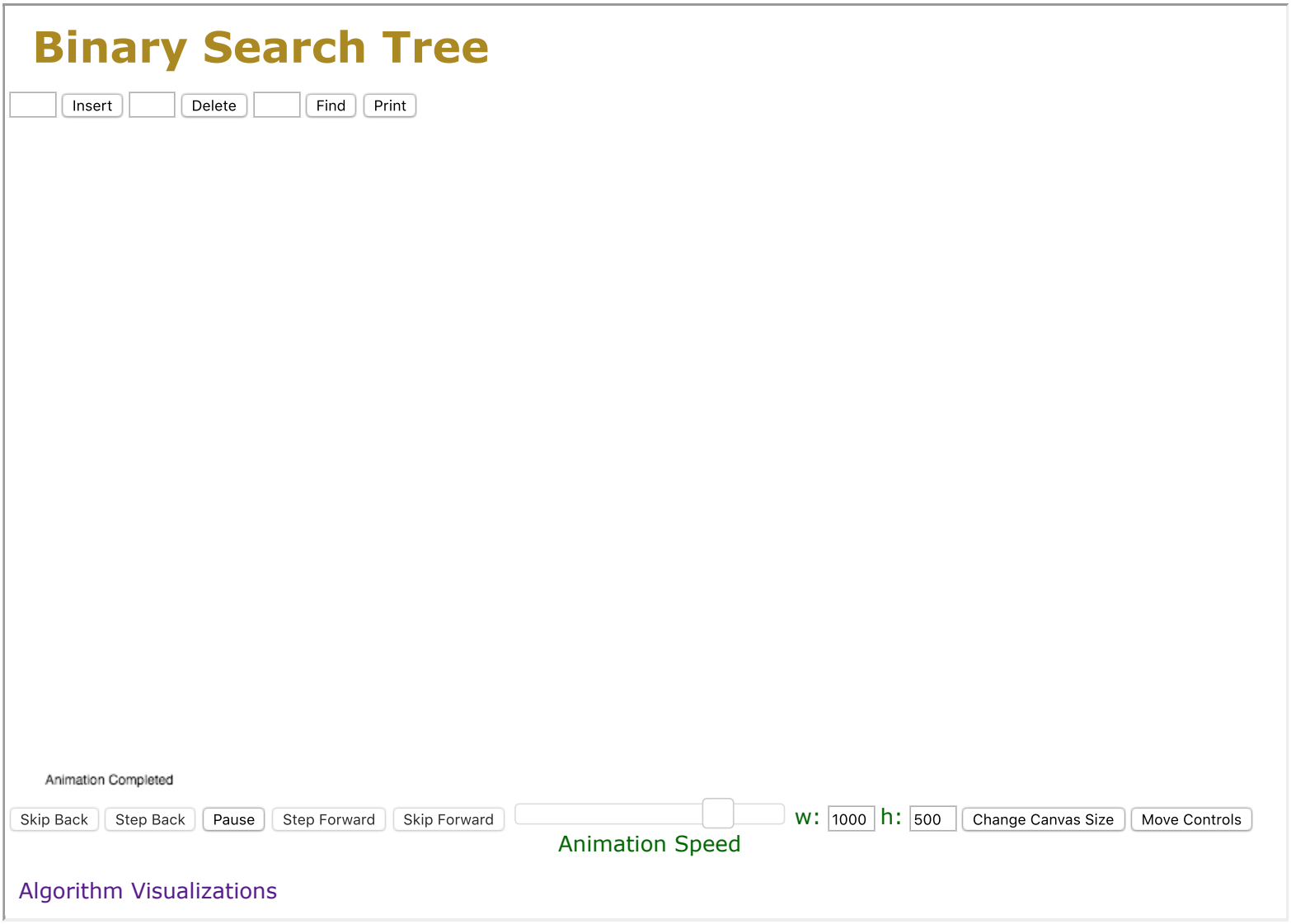
```
remove(element): // removes element if it exists in BST (returns True), or returns False otherwise
    current = root                 // start at the root
    while current != element:
        if element < current:      // if element < current, traverse left
            current = current.leftChild
        else if element > current: // if element > current, traverse right
            current = current.rightChild
        if current == NULL:        // if we traversed and there was no such child, failure
            return False
    // we only reach here if current == element, which means we found element
    if current.leftChild == NULL and current.rightChild == NULL:     // Case 1 (no children)
        remove the edge from current.parent to current
    else if current.leftChild == NULL or current.rightChild == NULL: // Case 2 (one child)
        have current.parent point to current's child instead of to current
    else:                                                            // Case 3 (two children)
        s = current's successor
        replace current with s
        remove the edge from s.parent to s
```

The second function we will describe is the **in-order traversal** function. An in-order traversal of a **BST** starts at the root and, for any given node, traverses left, then "visits" the current node, and then traverses right. However, what if we want to iterate through the elements of a **BST** in sorted order with the freedom of starting at any node that we want? Doing a full-fledged in-order traversal starting at the root would be inefficient. Instead, we can simply start at whichever node we want and repeatedly call the successor function until we reach a point where no more successors exist.

```
inOrder(): // perform an in-order traversal over the elements of BST using successor()
    current = the left-most element of BST
    while current != NULL:
        output current
        current = successor(current)
```

Step 8

Below is a visualization of a **Binary Search Tree**, created by David Galles at the University of San Francisco. Note that the delete algorithm the visualization tool uses differs from that which we cover in the next few steps, so focus mainly on insert and find.



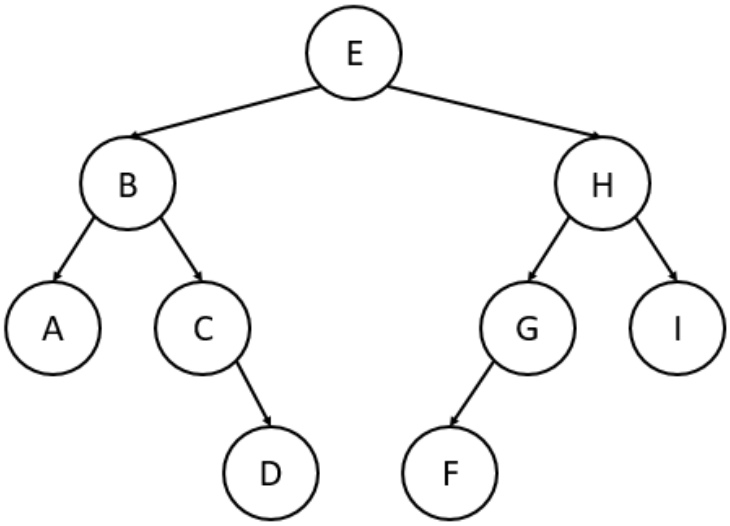
Step 9

To be able to analyze the performance of a **Binary Search Tree**, we must first define some terms to describe the tree's shape.

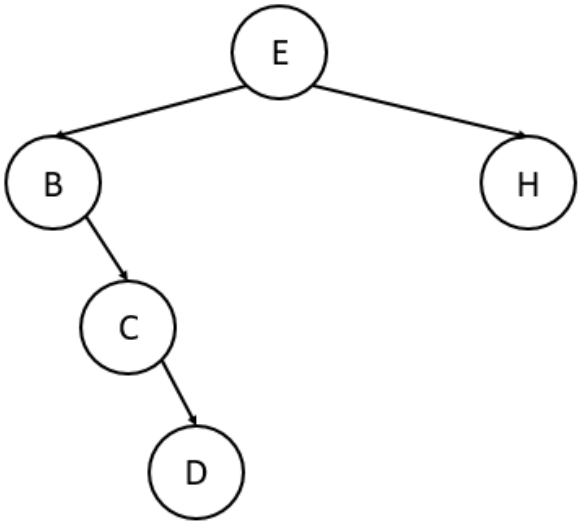
A tree's **height** is the longest distance from the root of the tree to a leaf, where we define "distance" in this case as the number of edges (not nodes) in a path. A tree with just a root has a height of 0, a tree with a root and one child has a height of 1, etc. For the sake of consistency, we say that an empty tree has a height of -1.

Notice that the worst-case time complexity of inserting, finding, and removing elements of a **BST** is proportional to the its *height*: in the tree-traversal step of any of the three aforementioned operations (i.e., traversing left or right to either find an element or to find the insertion site for a new element), we perform a constant-time comparison operation to see if we have to continue left or right, or to see if we are finished. Recall that we perform this constant-time comparison operation once for each level of the **BST**, so as the number of levels in a **BST** grows (i.e., the *height* of the **BST** grows), the number of comparison operations we must perform grows proportionally.

A **balanced** binary tree is one where many (or most) internal nodes have exactly two children. A *perfectly balanced* binary tree is one where *all* internal nodes have exactly two children. The term "balanced" can be generalized to any *k*-ary tree (e.g. binary, ternary, 4-ary, etc.): a balanced *k*-ary tree is one where many (or most) internal nodes have exactly *k* children. Below is an example of a balanced **BST**:



An **unbalanced** binary tree is one where many internal nodes have exactly one child. A *perfectly unbalanced* binary tree is one where *all* internal nodes have exactly one child. The term "unbalanced" can be generalized to any *k*-ary tree: an unbalanced *k*-ary tree is one where many internal nodes have less than *n* children (the more extreme deviation from *n* children, the more unbalanced the tree). Below is an example of an unbalanced **BST**:



It should be clear that, with regard to balance, the two extreme shapes a tree can have are *perfectly balanced* and *perfectly unbalanced*.

If a tree is *perfectly unbalanced*, each internal node has exactly one child, so the tree would have a height of ***n*-1** (we would have *n* nodes down one path, and that path will have *n*-1 edges: one edge between each pair of nodes along the path).

If a binary tree is *perfectly balanced*, each internal node has exactly two children (or *k* children, if we generalize to *k*-ary trees), so a *perfectly balanced* binary tree with *n* elements has a height of $\mathbf{\log_2(n+1)-1}$. This equation looks complicated, but the $\log_2(n)$ portion, which is the important part, has some simple intuition behind it: every time you add another level to the bottom of a *perfectly balanced* binary tree, you are roughly doubling the number of elements in the tree. If a tree has one node, adding another row adds 2 nodes. If a tree has 3 nodes, adding another row adds 4 nodes. If a tree has 7 nodes, adding another row adds 8 nodes. Since the height of a *perfectly balanced* binary tree effectively tells you "how many doublings" occurred with respect to a one-node *perfectly balanced* binary tree, the number of nodes in the tree would be roughly $n = 2^h$ (where *h* is the height of the tree), so the height of the tree would be roughly $h = \log_2(n)$.

STOP and Think: What previously-discussed data structure does a binary tree transform into when it becomes *perfectly unbalanced*?

Step 10

EXERCISE BREAK: What is the worst-case time complexity of the `find` operation of a **Binary Search Tree** with *n* elements?

To solve this problem please visit https://stepik.org/lesson/28727/step/10

Step 11

EXERCISE BREAK: What is the worst-case time complexity of the `find` operation of a *balanced* **Binary Search Tree** with *n* elements?

To solve this problem please visit https://stepik.org/lesson/28727/step/11

Step 12

As you might have noticed, although **Binary Search Trees** are relatively efficient when they are balanced, they can become unbalanced fairly easily depending on the order in which the elements are inserted.

STOP and Think: If we were to insert elements into a **BST** in sorted order (i.e., insert the smallest element, then insert the next smallest, etc.), what would the resulting tree look like?

We found out that a **Binary Search Tree** performs pretty badly in the worst case, but that it performs pretty well when it is well-balanced. How does the **BST** perform on average (i.e., what is its average-case time complexity)? Although this is a seemingly simple question, it requires formal mathematical exploration to answer, which we will dive into in the next section.