Data Structures and Algorithms

Lesson 6. Trees

Alexandra Ciobotaru

24 mar 2022

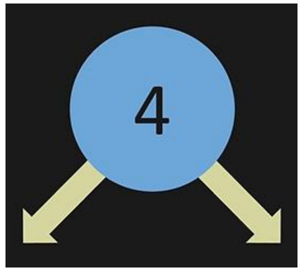
Syllabus

- **Lesson 1. Introduction: Algorithms, Data Structures and Cognitive Science.**
- **Lesson 2. Python Data Types and Structures.**
- **Lesson 3. Principles of Algorithm Design.**
- **Lesson 4. Lists and Pointer Structures.**
- **Lesson 5. Stacks and Queues.**
- **Lesson 6. Trees.**
- Lesson 7. Hashing and Symbol Tables.
- Lesson 8. Graphs and Other Algorithms.
- Lesson 9. Searching.
- Lesson 10. Sorting. Mid-term evaluation quiz (30%)
- Lesson 11. Selection Algorithms.
- Lesson 12. String Algorithms and Techniques.
- Lesson 13. Design Techniques and Strategies.
- Lesson 14. Implementations, Applications and Tools.
- Lab (20%) + Project (50%).

Trees

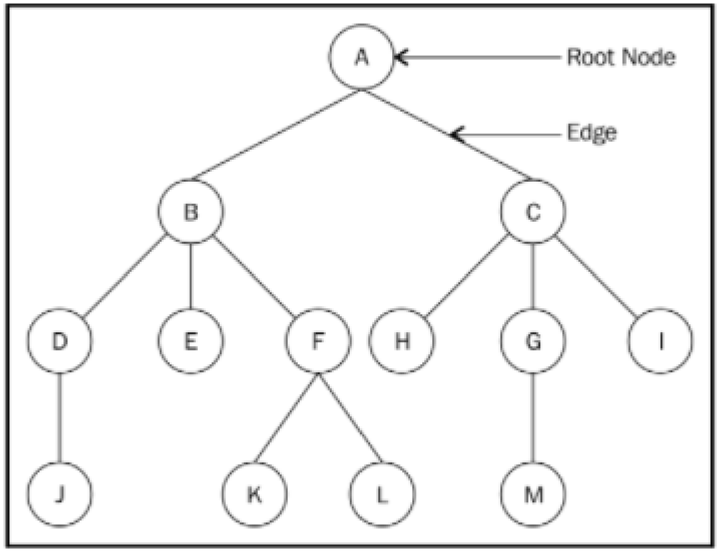
- Trees are hierarchical forms of data structures;
- They are made of linked lists;
- A LL is a tree that doesn’t fork, so a binary tree will have two pointers instead of one

```
{
      "value": 4,
      "left": None,
      "right": None
}
```



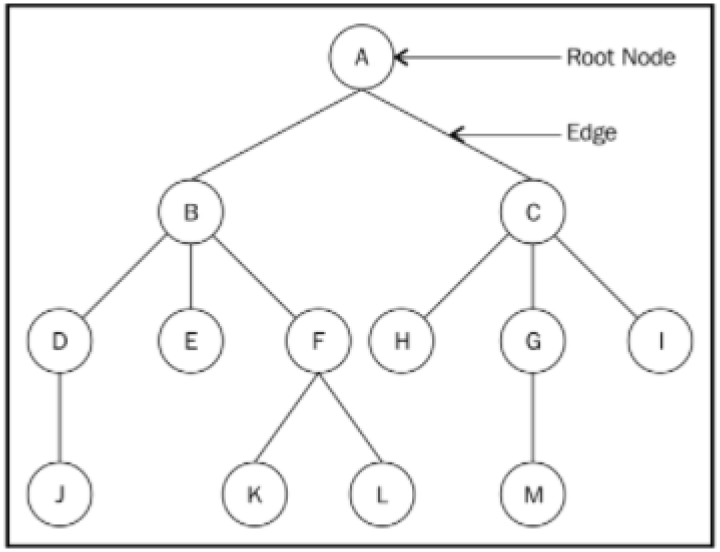
Terminology

- **Nodes** – a node can be any data structure that holds data (A-M)
- **Root** node – the first node from which all other nodes in the tree are attached (A)
- **Sub-tree** - a tree with its nodes being a descendant of some other tree (F-K-L)
- **Degree** – the total number of children of a given node
  - A – degree 2
  - B – degree 3
- **Leaf** node – a node with no children (J, E, K, L, H, M, I)
- **Edge** – connection between nodes
- **Parent** – a node which has a further sub-tree (B for D, E, F)



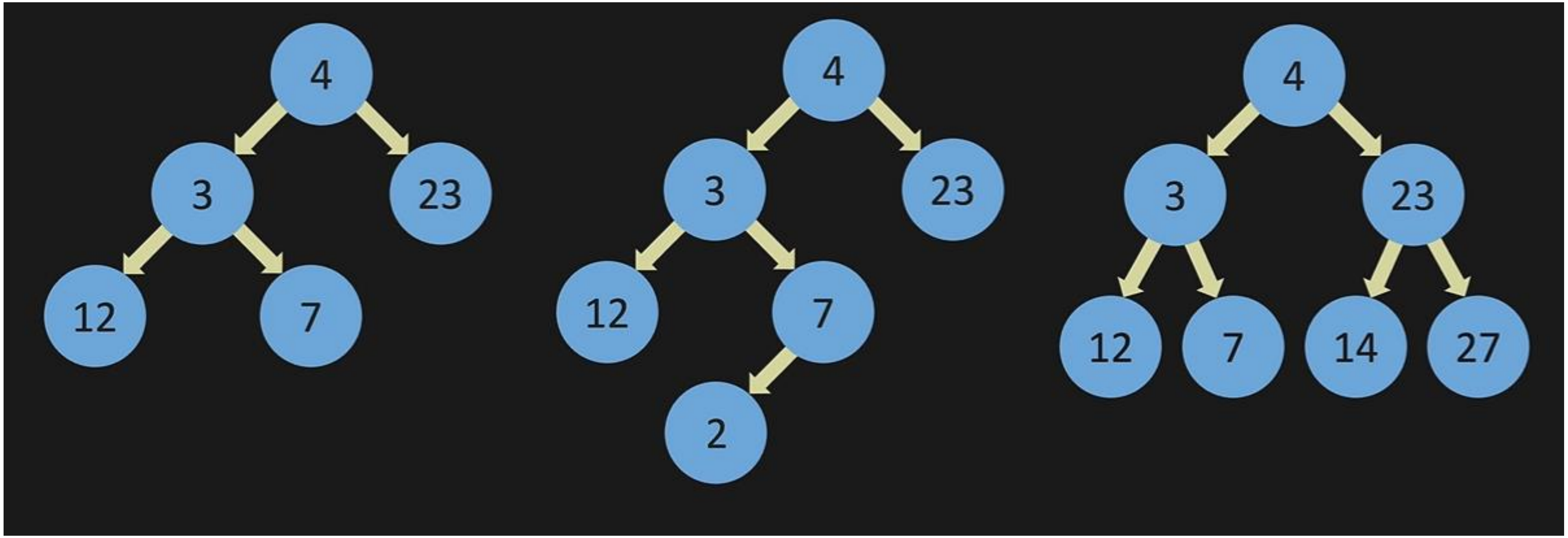
Terminology

- **Child** – a node connected to its parent (B and C for A)
- **Sibling** – nodes with the same parent (D, E, F)
- **Level** – the root node is level 0, its children are level 1, the cldren’s children level 2, and so on (D, E, F, H, G, I are level 2)
- **Height** – the total number of *nodes* in the longest path of the tree (“longest path” in Catan)
- **Depth** of a node – the number of edges from the root of the tree to that node (H is of depth 2)



Terminology

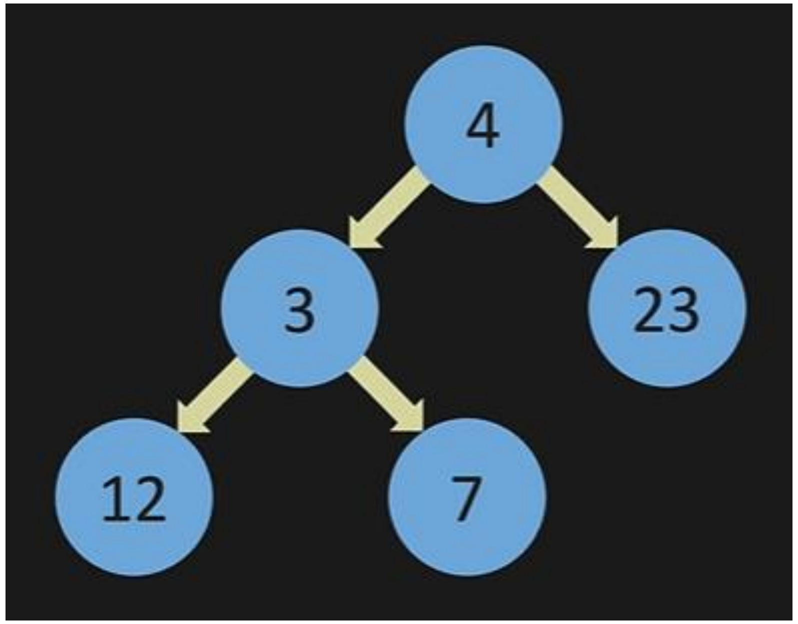
Full Not Full Perfect

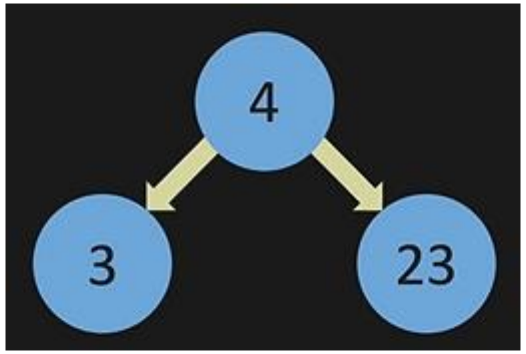


Binary Trees

- You can think at a tree as a dictionary, but it works a little different

```
{
    "value": 4,
    "left": {
        "value": 3,
        "left": None,
        "right": None
    },
    "right": {
        "value": 23,
        "left": None,
        "right": None
    }
}
```

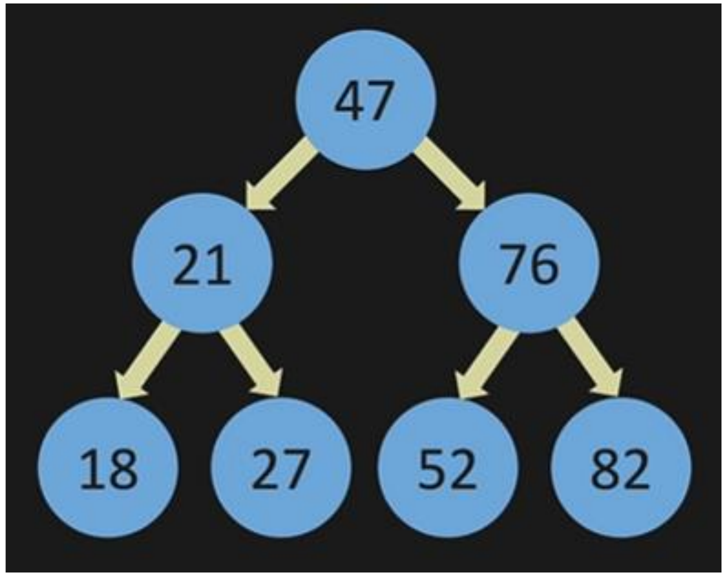




The Binary Search Tree

- The binary search tree is a binary tree, but with its nodes placed out in a particular way
- If the child of the node is greater than the parents value, than the node is going to be placed to the right, eitherwise its going to be placed on the left of the node

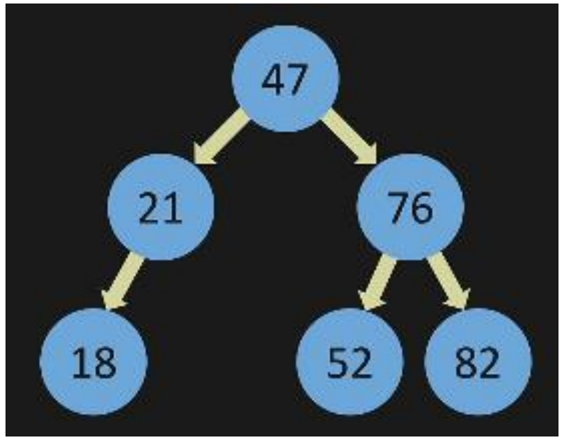
Order: 46, 76, 52, 21, 82, 18, 27



BST Constructor

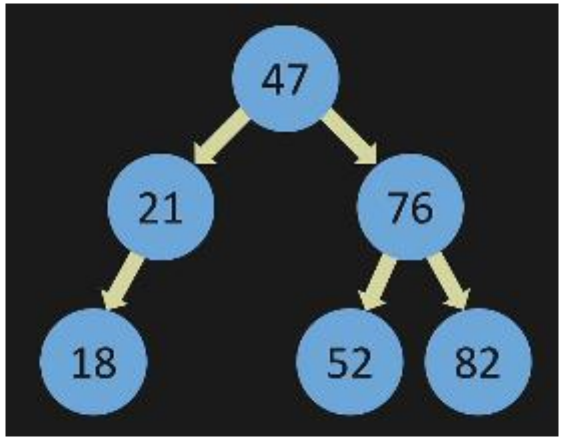
```
class Node:
    def __init__(self, value):
        # the node now points to both left and right
        self.value = value
        self.left = None
        self.right = None

class BinarySearchTree:
    def __init__(self):
        # new_node = Node(value) - we don't create a new node here
        # we create an empty tree and when we add the first item,
we’ll add it with the insert method
        self.root = None

my_tree = BinarySearchTree()
print(my_tree.root)
```

Insert



- Let’s say we want to insert the value 27 in this BST
- First we’ll need to **create a new node**
- Next we’ll compare 27 to the root and if the result is higher we’ll move to the right, otherwise we’ll move to the left
  - **If < left else > right**
- In our example we’ll go to the left
- Now there re two possibilities: either there is a node there, in that spot (like in our example) or there isn’t – if there isn’t, we can just insert the new node there, otherwise we compare again
  - **If None insert new_node else move to next**
- Now we compare our new node to the value found
  - **If < left else > right**
- When we go to the right we have to determine, is that spot empty? If it is we’re going to insert the new node, so again:
  - **If None insert new_node else move to next**

Insert



- So we did the comparison operation, and the None verification twice, when inserting 27 -> these two operations should be inside a loop
- It should be a while loop that iterates for as long as it reaches None
- we'll use a temporary variable to move through the tree, and we will compare new_node to temp
- Edge cases:
  - An empty tree (if root == None then root = new_node)
  - When the new node value equals the node's value (inside the while loop: if new_node==temp return False)

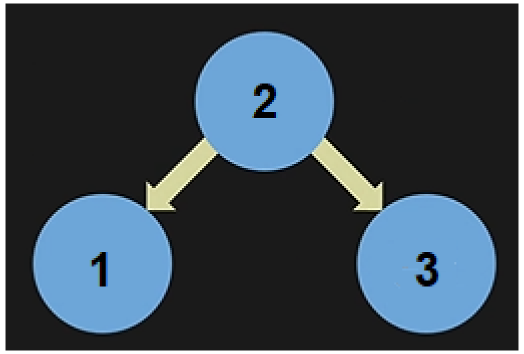
The pseudocode for Insert

- `create new node`
- `if root == None then root = new_node`
- `temp = self.root`
- `while loop`
  - `if new_node == temp return False`
  - `if < left else > right`
  - `if None insert new_node else move to next`

```
class BinarySearchTree:
…
 def insert(self, value):
        new_node = Node(value)   # we create the new node
        if self.root is None:    # the situation when the tree is empty
            self.root = new_node
            return True     # after inserting the method ends by returning True
        temp = self.root    # otherwise we prepare the temp variable
        while (True):
            if new_node.value == temp.value:  # the situation when the new node is
                                              equal to the node in the tree
                return False
            if new_node.value < temp.value:  # the situation when the value is
                                              smaller
                if temp.left is None:  # if the left value is None we place the new
                                node there
                    temp.left = new_node
                    return True
                temp = temp.left   #otherwise we need to compare the new node to
                                next value in the tree, so we move temp down
            else:  # if the value is higher
                if temp.right is None:   # if the right value is None we place it here
                    temp.right = new_node
                    return True
                temp = temp.right  # it loops until we insert the node
```

Let’s test the Insert method

```
my_tree = BinarySearchTree()
my_tree.insert(2)
my_tree.insert(1)
my_tree.insert(3)

print(my_tree.root.value)
print(my_tree.root.left.value)
print(my_tree.root.right.value)
```



Contains (pseudocode)

- See if a tree contains a particular value:
- `if root == None return False`
- `temp = self.root`
- When temp reaches the end of the tree it means the searched value is not contained
- `while temp is not None:`
  - `if < left`
  - `elif > right`
  - `else = return True`
- Return False

Code for Contains Method

```
class BinarySearchTree:
…
   def contains(self, value):
       if self.root is None:   #not actually needed
           return False
      temp = self.root
      while (temp is not None):
          if value < temp.value:
              temp = temp.left
          elif value > temp.value:
              temp = temp.right
          else:
              return True
      return False
```

Let’s test the Contains Method

```
my_tree = BinarySearchTree()
my_tree.insert(47)
my_tree.insert(21)
my_tree.insert(76)
my_tree.insert(18)
my_tree.insert(27)
my_tree.insert(52)
my_tree.insert(82)


print(my_tree.contains(27))
```

Minimum Value

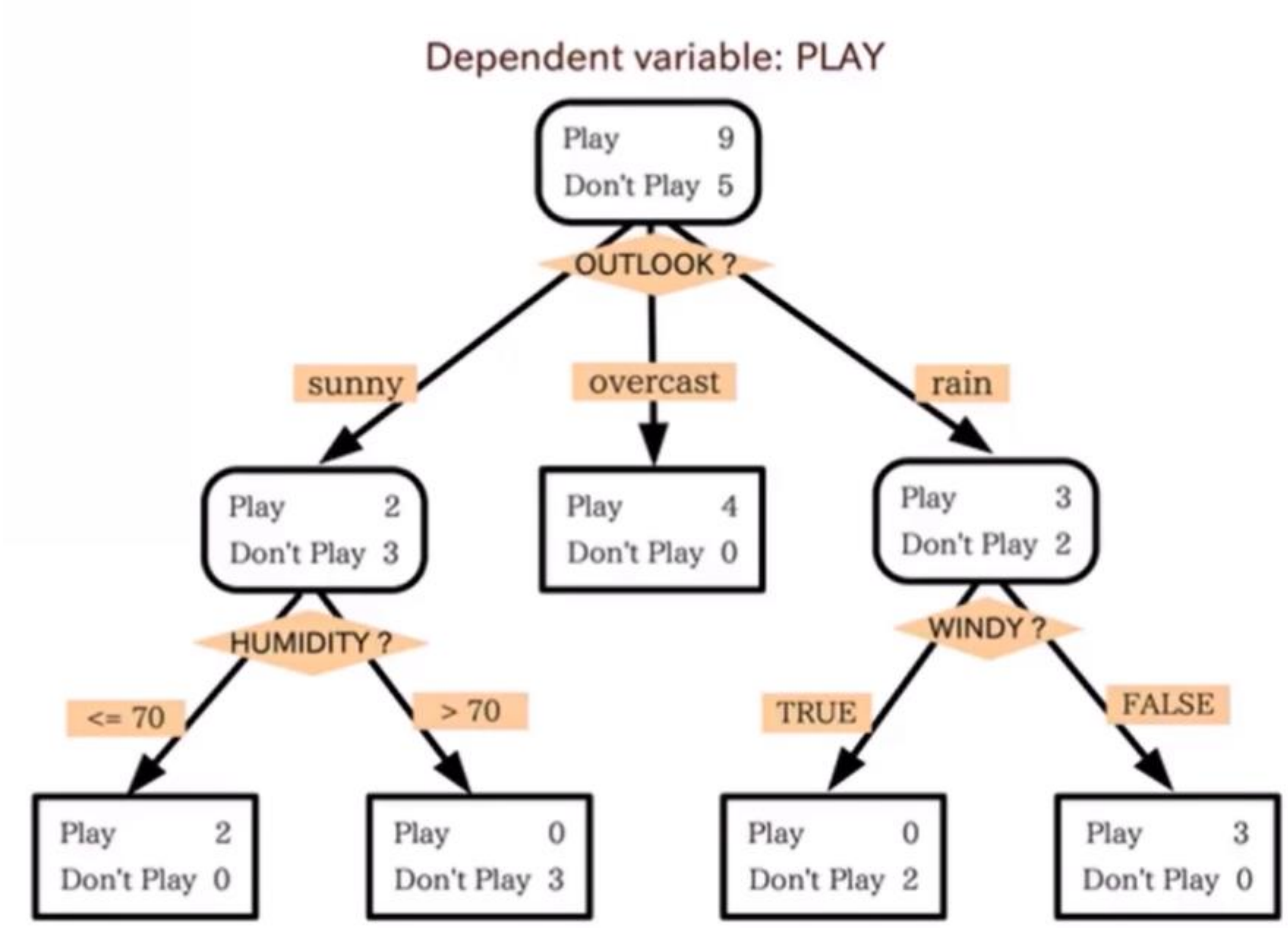
```
 def min_value_node(self, current_node):
        while current_node.left is not None:
            current_node = current_node.left
        return current_node.value
my_tree = BinarySearchTree()
my_tree.insert(47)
my_tree.insert(21)
my_tree.insert(76)
my_tree.insert(18)
my_tree.insert(27)
my_tree.insert(52)
my_tree.insert(82)
print(my_tree.min_value_node(my_tree.root))
```

Decision Tree Algorithm

- This is another machine learning model
- We can create a flowchart out of the training data that works as a “decision tree” to predict future outcomes
- If you have a *classification* task you can use a decision tree to actually look at multiple attributes that you can decide upon at each level in a flow chart
- The decision tree is a form of supervised learning, as the input data is labelled

Should I go out and play outside?

- This is a decision that depends on multiple variables, and a decision tree could be a good choice.
- The decision might have to do with the *temperature*, the *humidity*, wheather it’s *sunny* or not, and so on.
- A decision tree can look at all these different attributes, and decide what are the decisions I need to make regarding each attribute before I arrive to the final decision of whether or not I should go play outside.



Example: filter out resumes

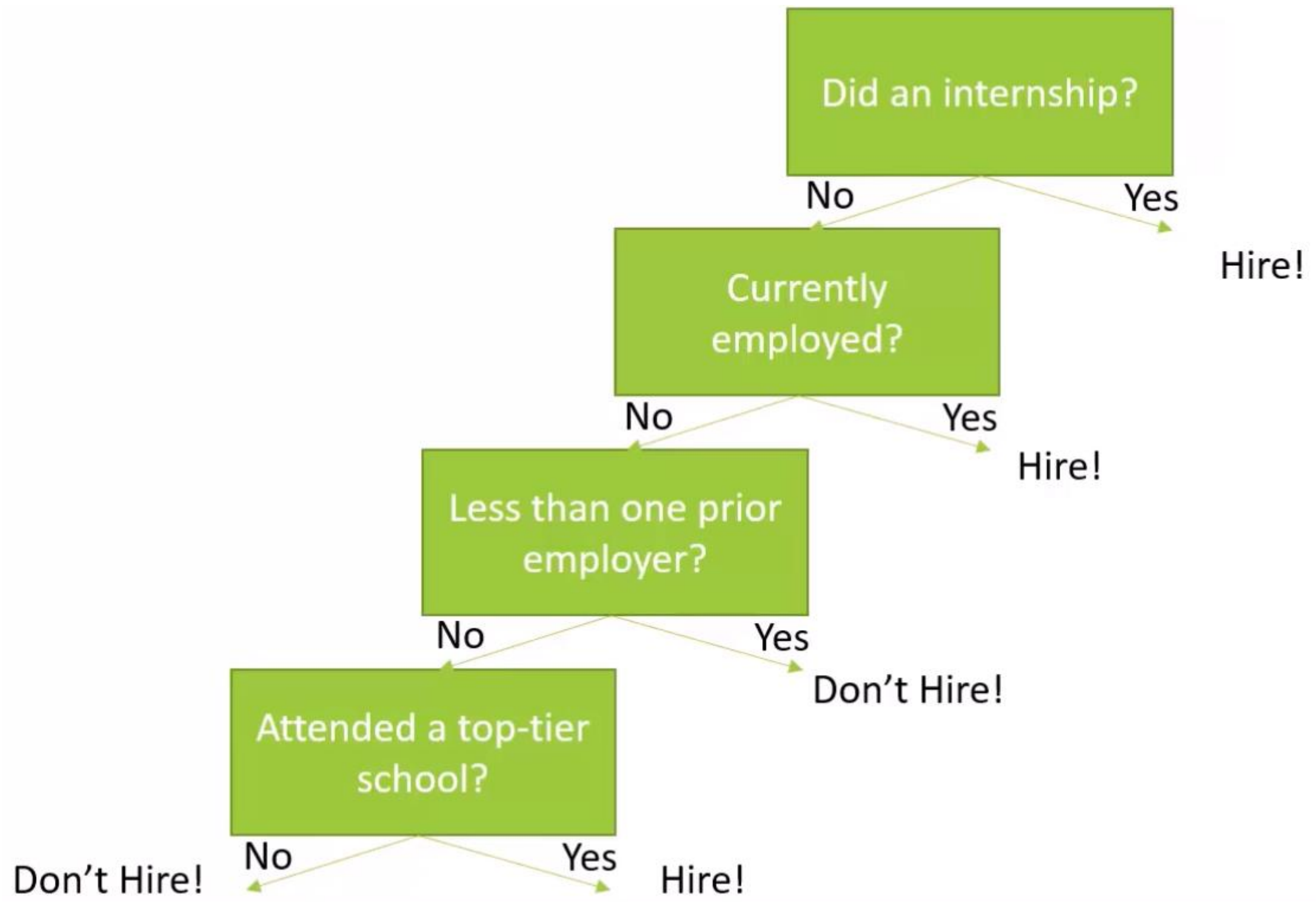
- We train data about past candidates and use the model to predict future candidates
- Some fabricated hiring data:

Candidate ID	Years Experience	Employed?	Previous employers	Level of Education	Top-tier school	Interned	Hired
0	10	1	4	0	0	0	1
1	0	0	0	0	1	1	1
2	7	0	6	0	0	0	0
3	2	1	1	1	1	0	1
4	20	0	2	2	1	0	0

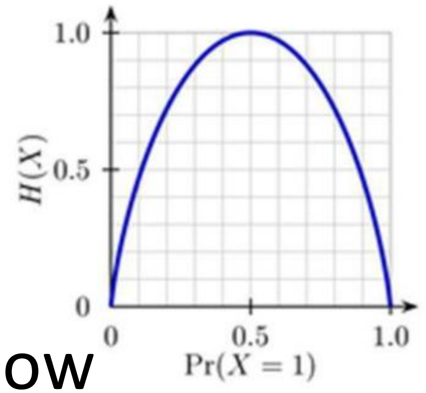
The dependent variable

Algorithm

- At each step, find the attribute we can use to partition the dataset in order to minimize the *entropy* of the data at the next step
- This is an ID3 (Iterative Dichotomiser 3) algorithm
- Is a greedy algorithm – as it goes down the tree it picks the attribute that will minimize the entropy at that point



What is entropy?

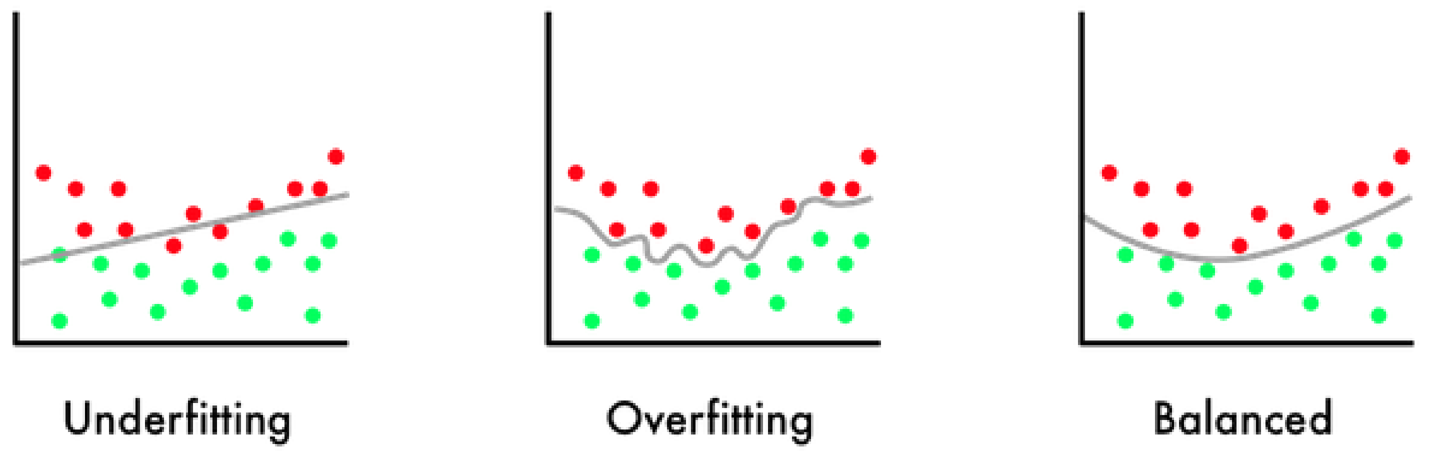


- Entropy is a measure of a dataset's disorder – how same or how different my data is.
- If we classify a dataset into N different classes:
  - The entropy is 0 if all classes are equal (we have just one class)
  - The entropy is high if all classes are different (there is more disorder in my dataset)
- Mathematically, the entropy can be computed using:
  - $H(s) = -p_1 \ln p_1 - \ldots - p_n \ln p_n$
  - For each class in my data I will have one of those *p* terms (the proportion of the data that in that class)
- More on the subject: https://youtu.be/0GCGaw0QOhA

Disadvantage - overfitting

- Decision trees are prone to **overfitting**
- The algorithm may be good on the trained data, but might work bad on real data
- To fight that, we can use random forests – we construct several alternate decision trees and let them "vote" on the final classification (this technique of combining classifiers is called ensemble learning)
- We have multiple trees (a forest), each that uses a random sub-sample of the data we have for training, and the trees vote for the final result (democracy!)
    - Bootstrap aggregating / bagging = randomly re-sampling the input data for each tree
    - Randomize a subset of the attributes each step
- Another thing random forests can do is to restrict the number of attributes it can chose between, at each stage, when trying to minimize the entropy as it goes
- -> variation from tree to tree

Extra: Overfitting



Source: https://towardsdatascience.com/8-simple-techniques-to-prevent-overfitting-4d443da2ef7d

Thank you!