Advanced Programming
INFO135

Lecture 9: Greedy Algorithms

Mehdi Elahi

University of Bergen (UiB)

Binary Search Tree

- In the previous lecture we introduced **binary tree** structure.

- A special case of a binary tree that follows a specific **property** is called **Binary Search Tree**.

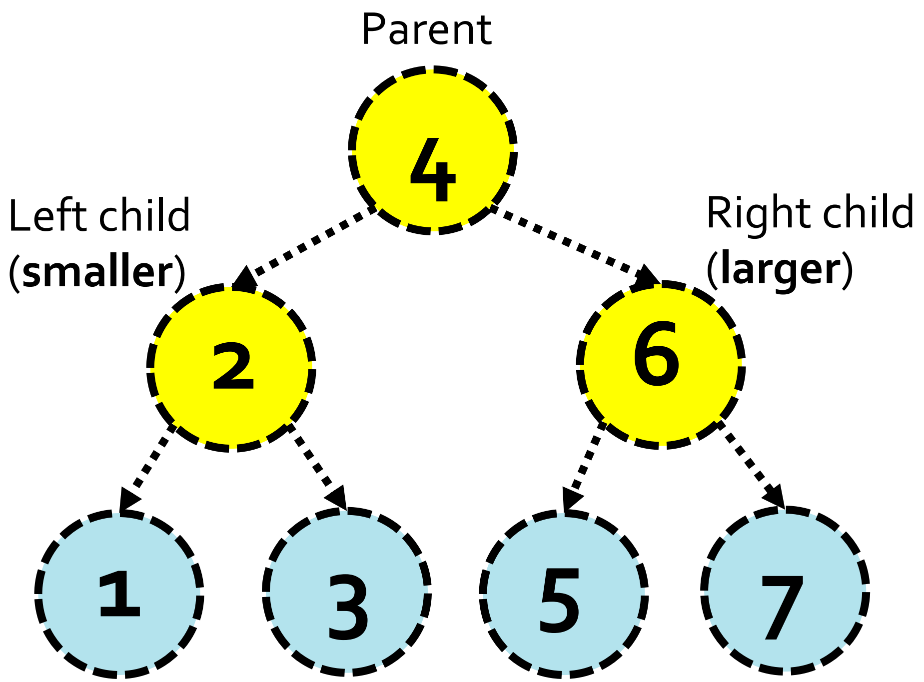
- Data stored in Binary Search Tree is more **efficient to find** (i.e., to search) than in an ordinary binary trees.

Binary Search Tree

- **Binary Search Tree (BST)** is a binary tree where:
  - values that are less than parent node are in **left** subtree
  - values that are greater than parent are in **right** subtree
- This is the **property** of Binary Search Tree. Value stored in a node is also called **Key.**
- Duplicates are **not allowed** in binary search tree (similar to Python Set).

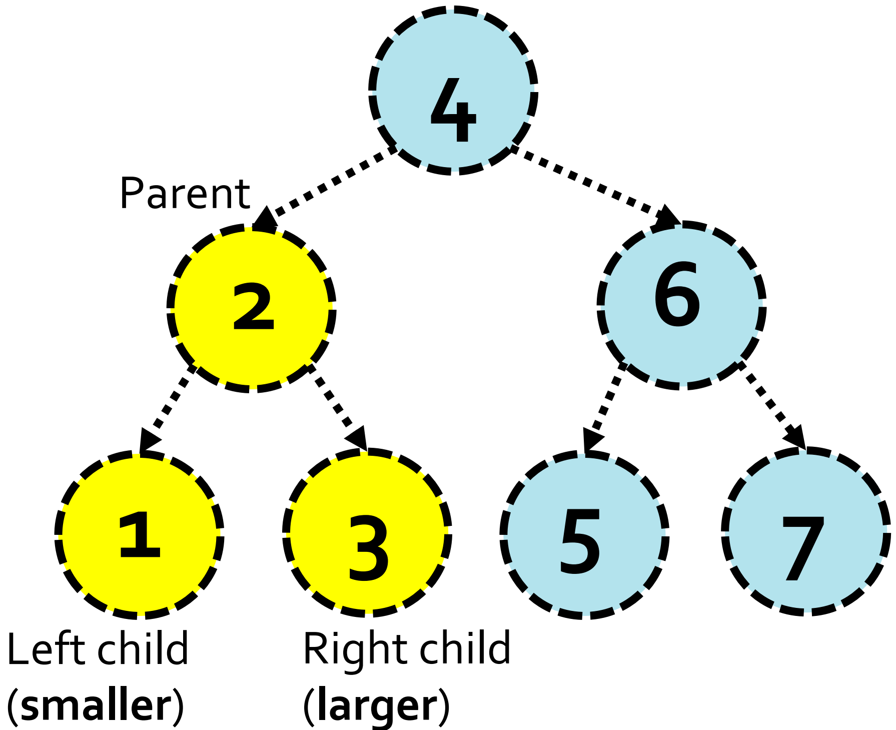
Binary Search Tree

- **Example:**
  - check the highlighted nodes of this binary tree:
    - value of parent (4) **is larger than** left child (2)
    - value of parent (4) **is smaller than** right child (6)



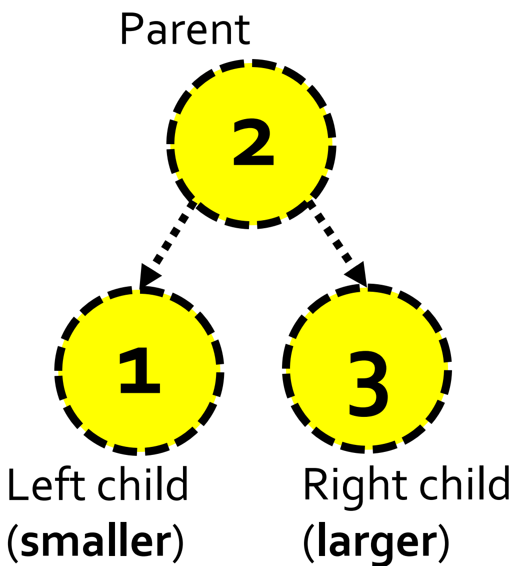
Binary Search Tree

- **Example:**
  - check the highlighted nodes of this binary tree:
    - value of parent (2) **is larger than** left child (1)
    - value of parent (2) **is smaller than** right child (3)



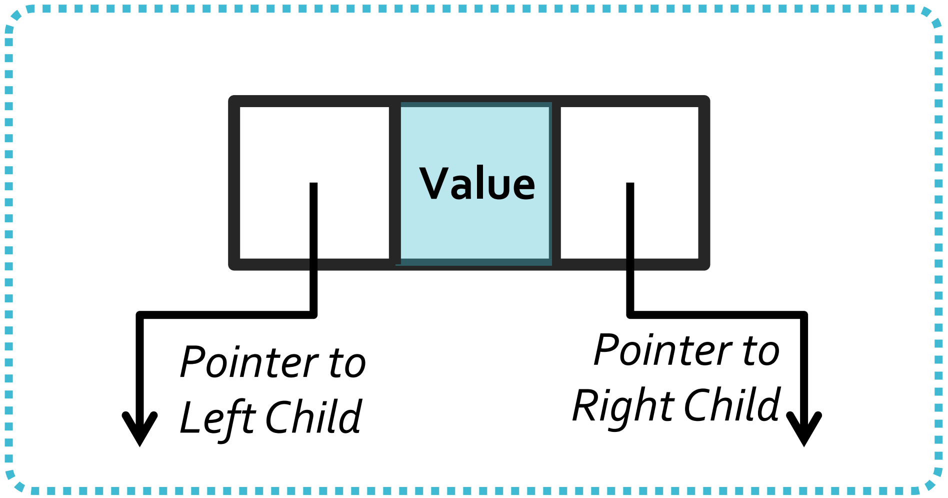
Binary Search Tree

- Lets check how this **Binary Search Tree** can be formed.

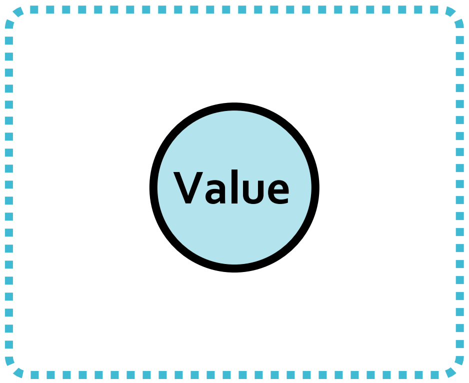


Binary Search Tree

- We **define a Node** structure, that has a value and a "pointer" to the left child & a "pointer" to right child.



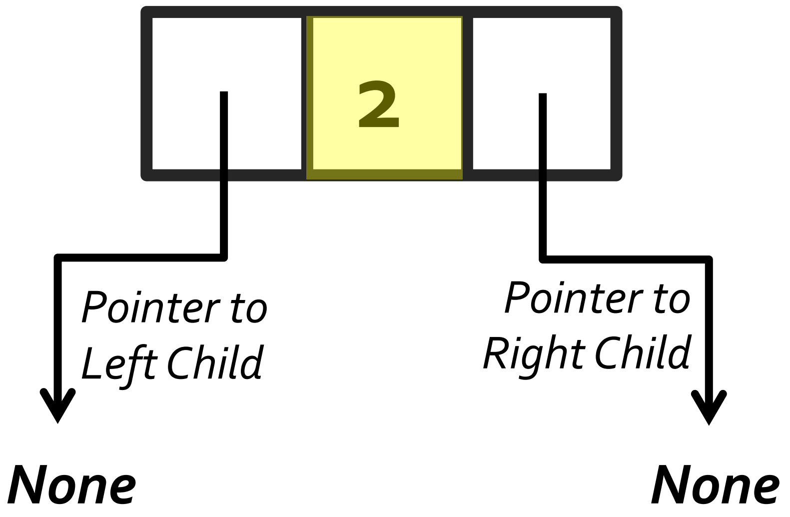
Node

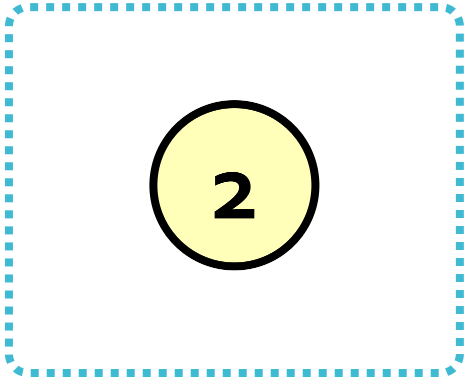


Node

Binary Search Tree

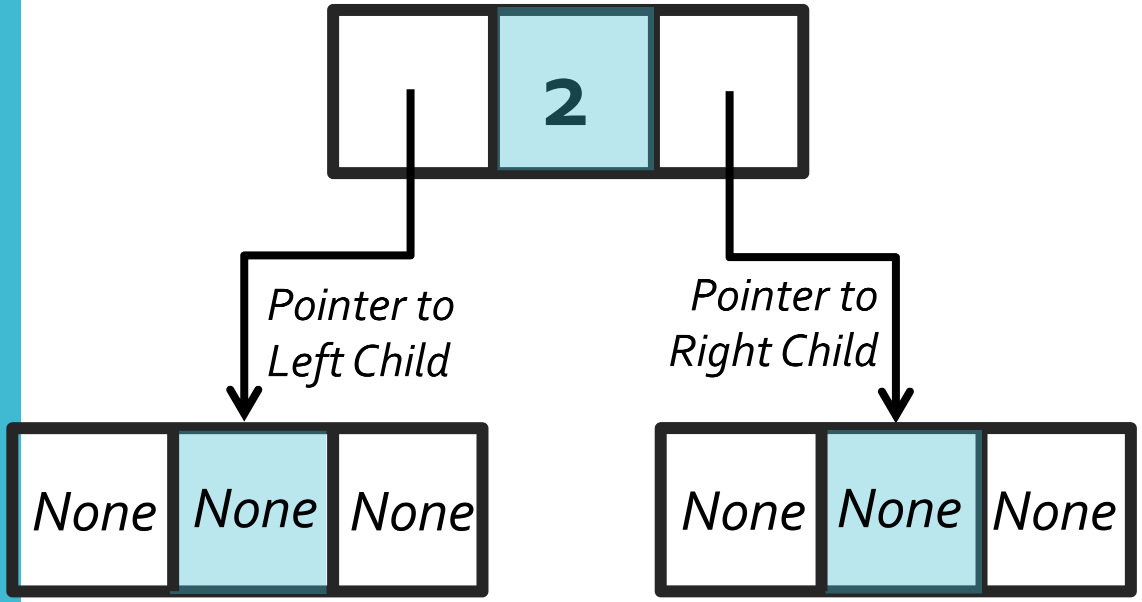
- **First node with value 2** is created, but it does not have any (left or right) child node.

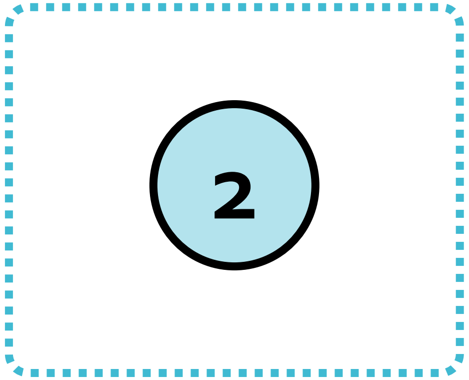




Binary Search Tree

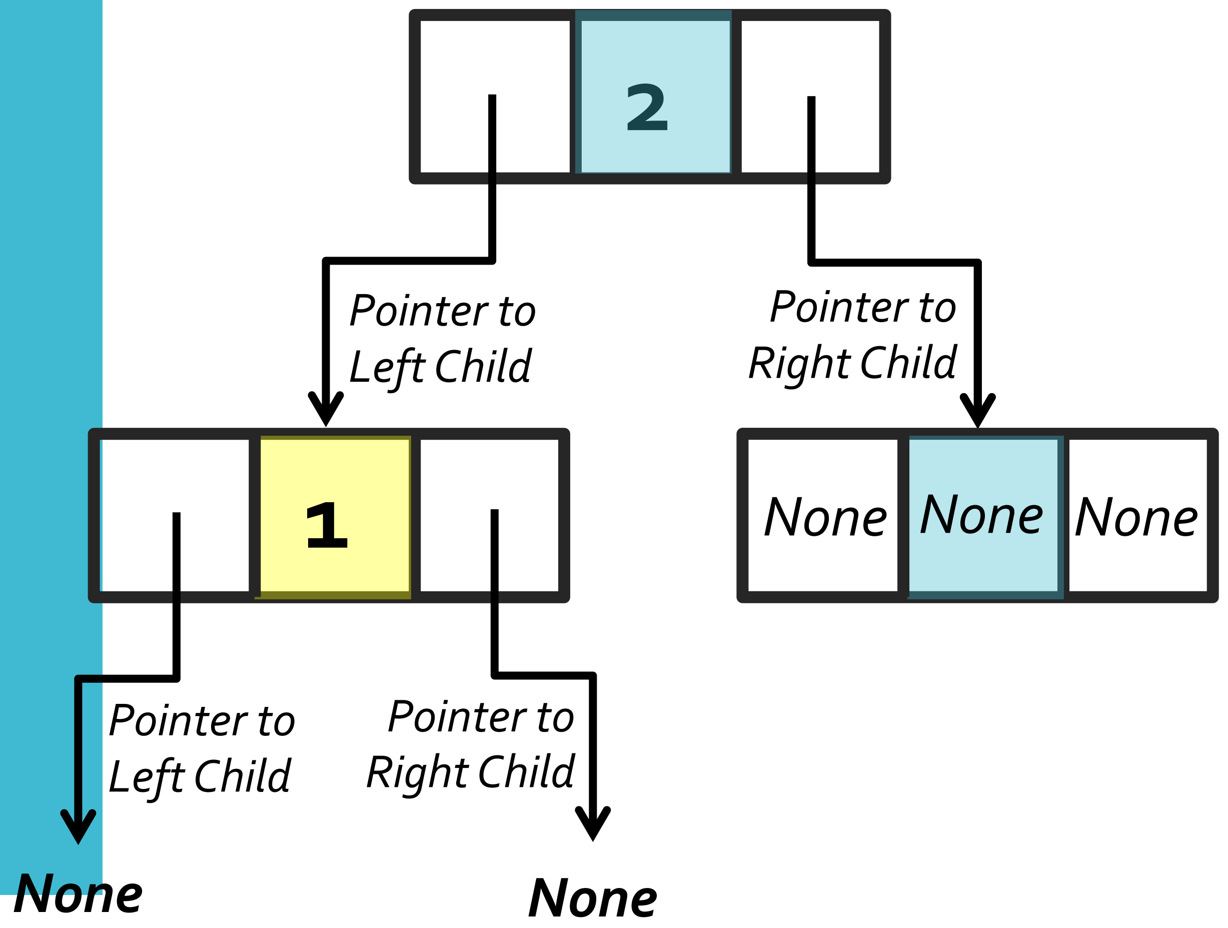
- No child can be represented with an **empty node** where values are *None*.

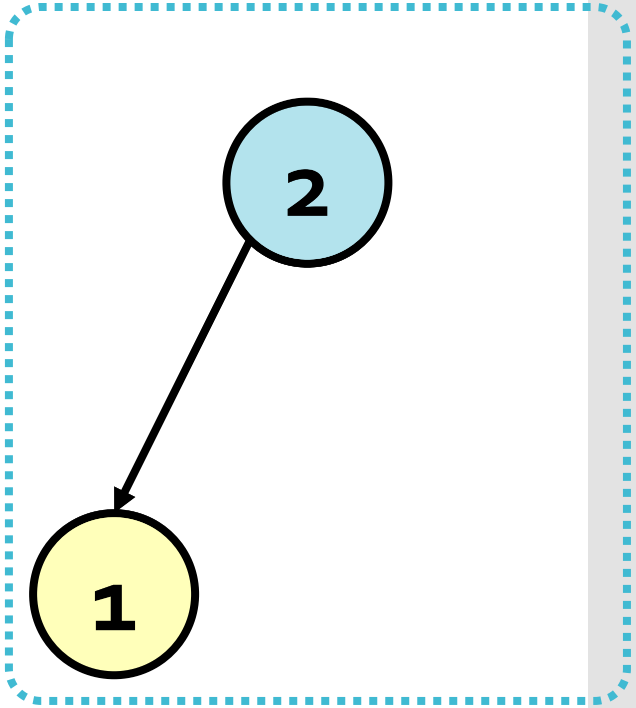




Binary Search Tree

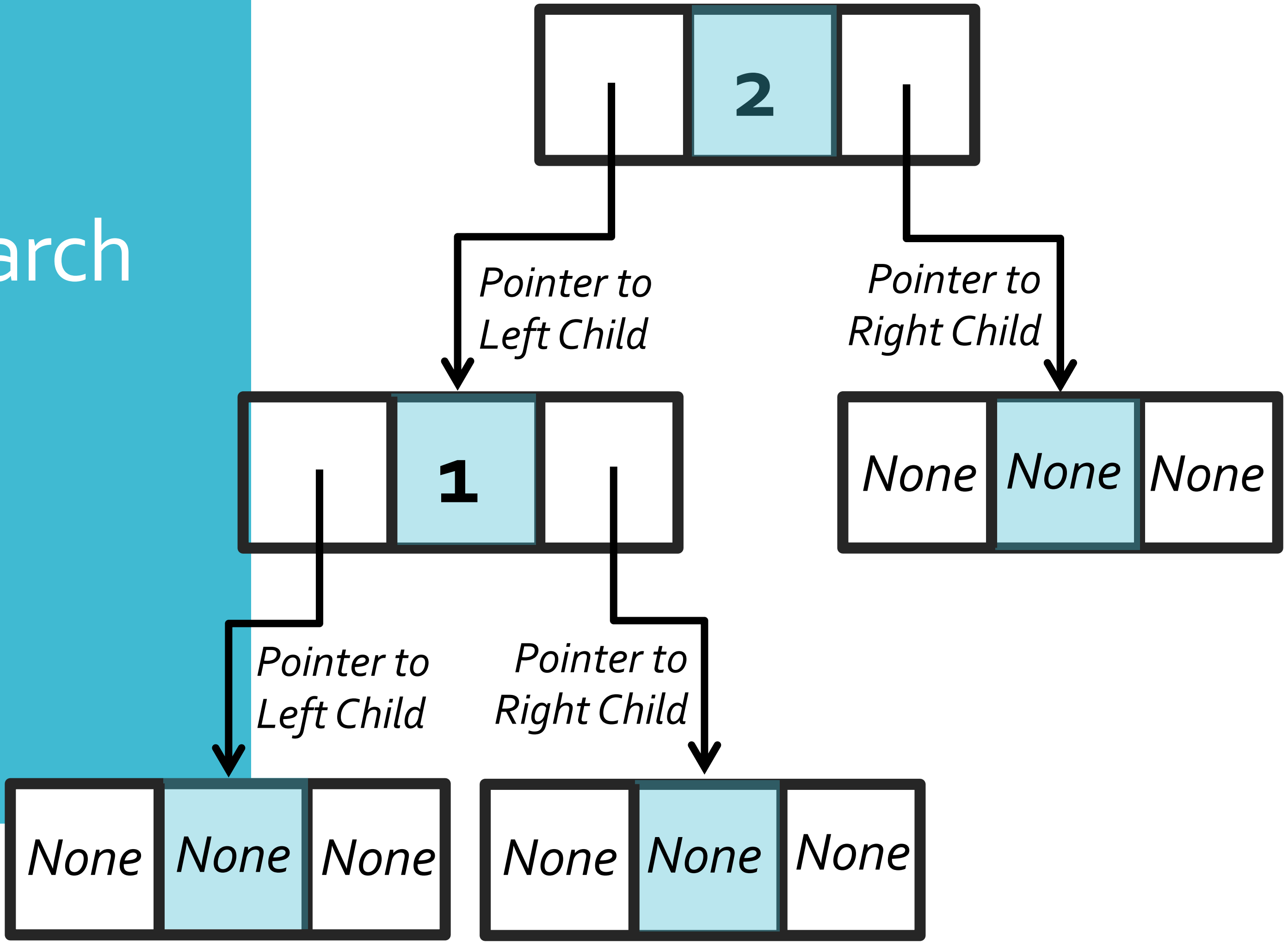
- **Next node with value 1** is created (to insert into the tree). This node has a value **smaller** than the first node and so, it is added as the **left child**.

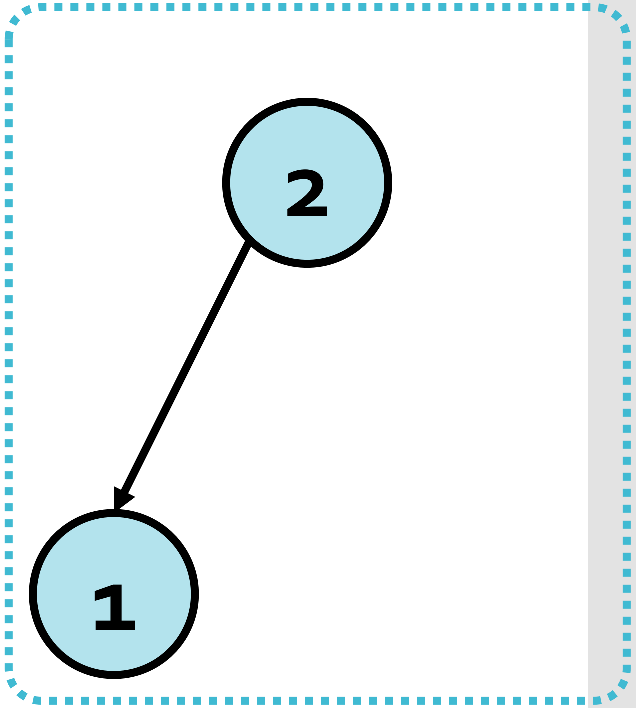




Binary Search Tree

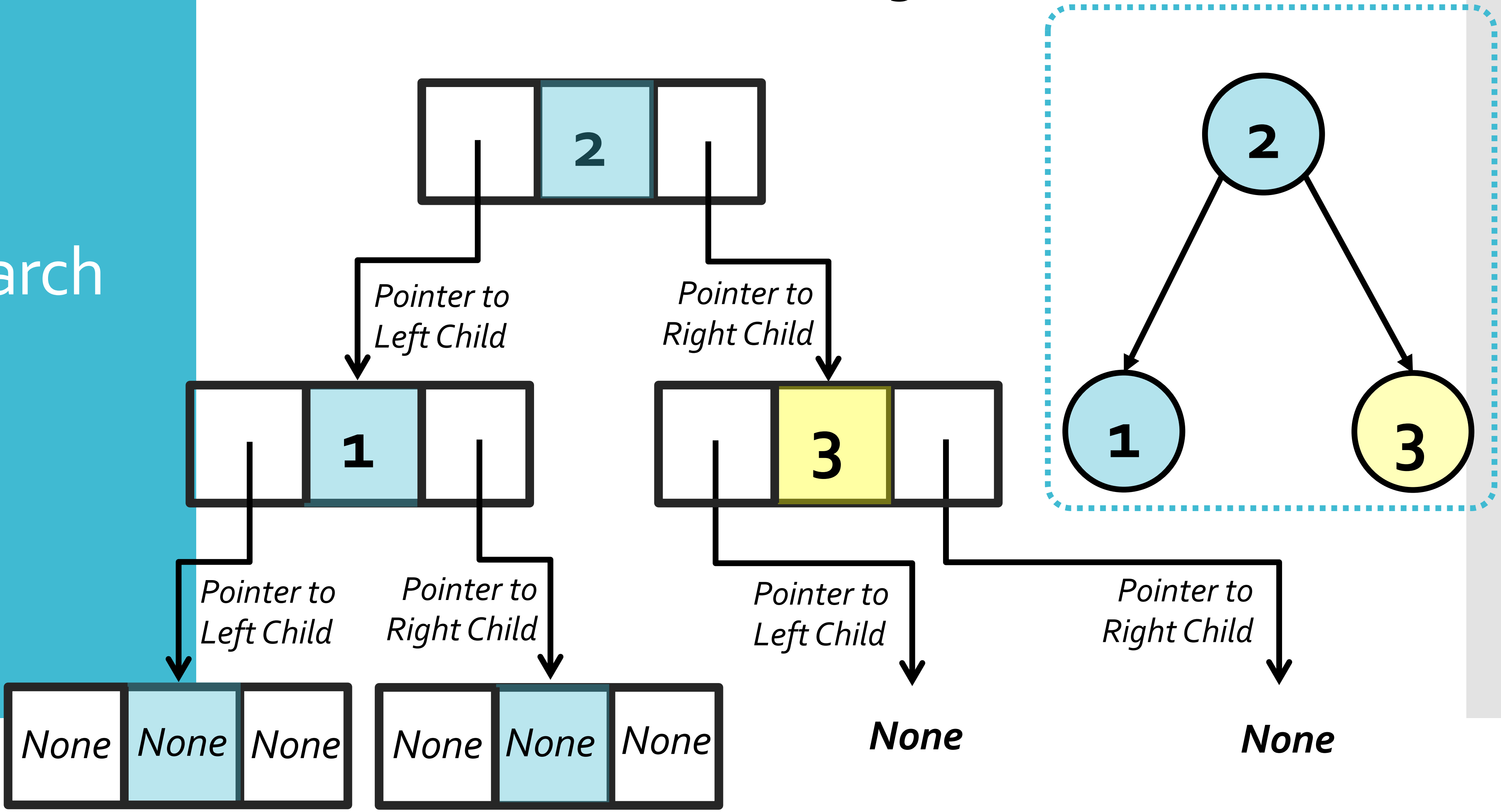
- We create **empty child nodes** with values set to *None*.





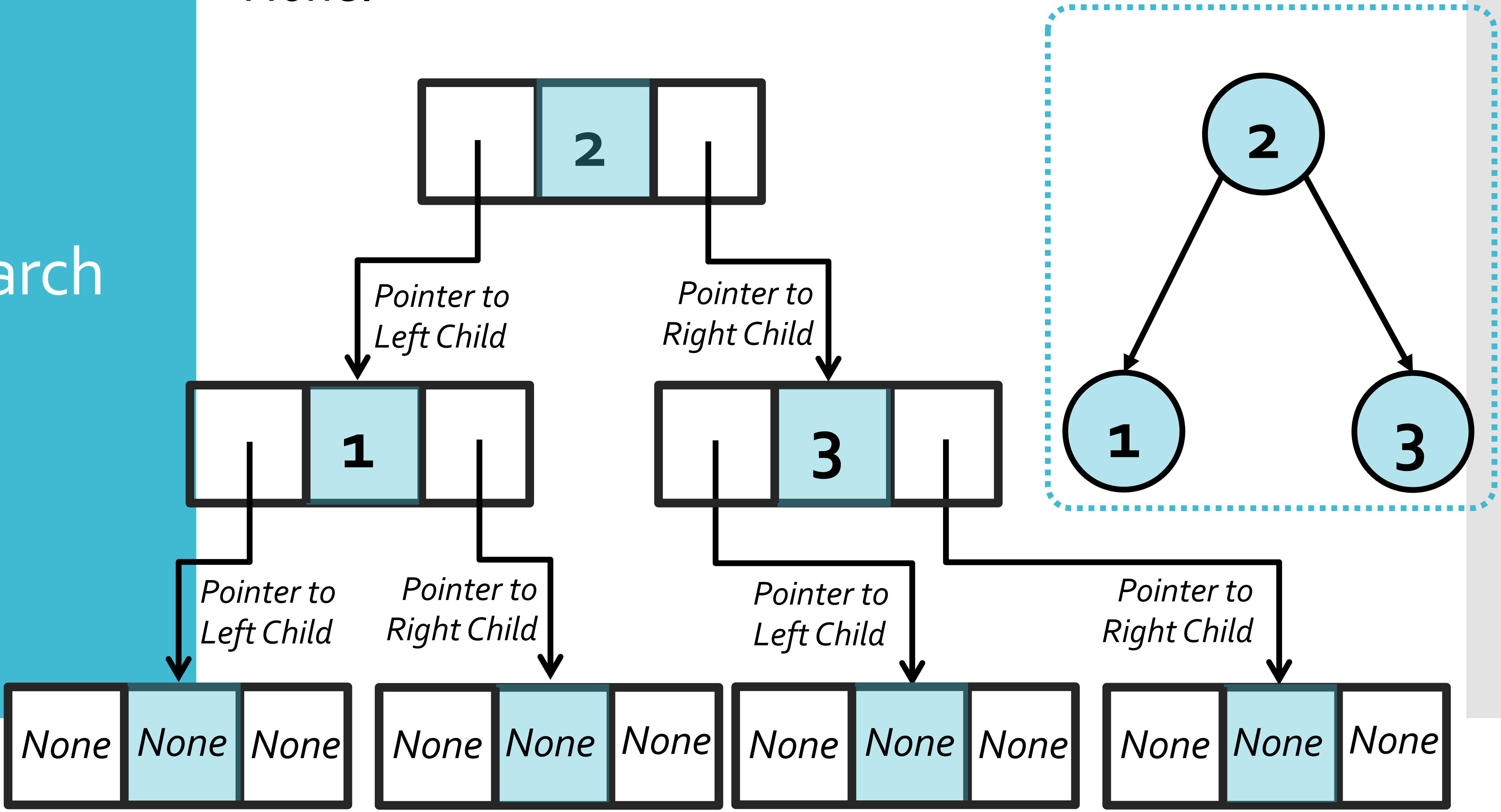
Binary Search Tree

- The next is **value 3** & **we create a node for it** (to insert it into tree). This node has a value **larger** than root node (2) and so, it is added as the **right child**.

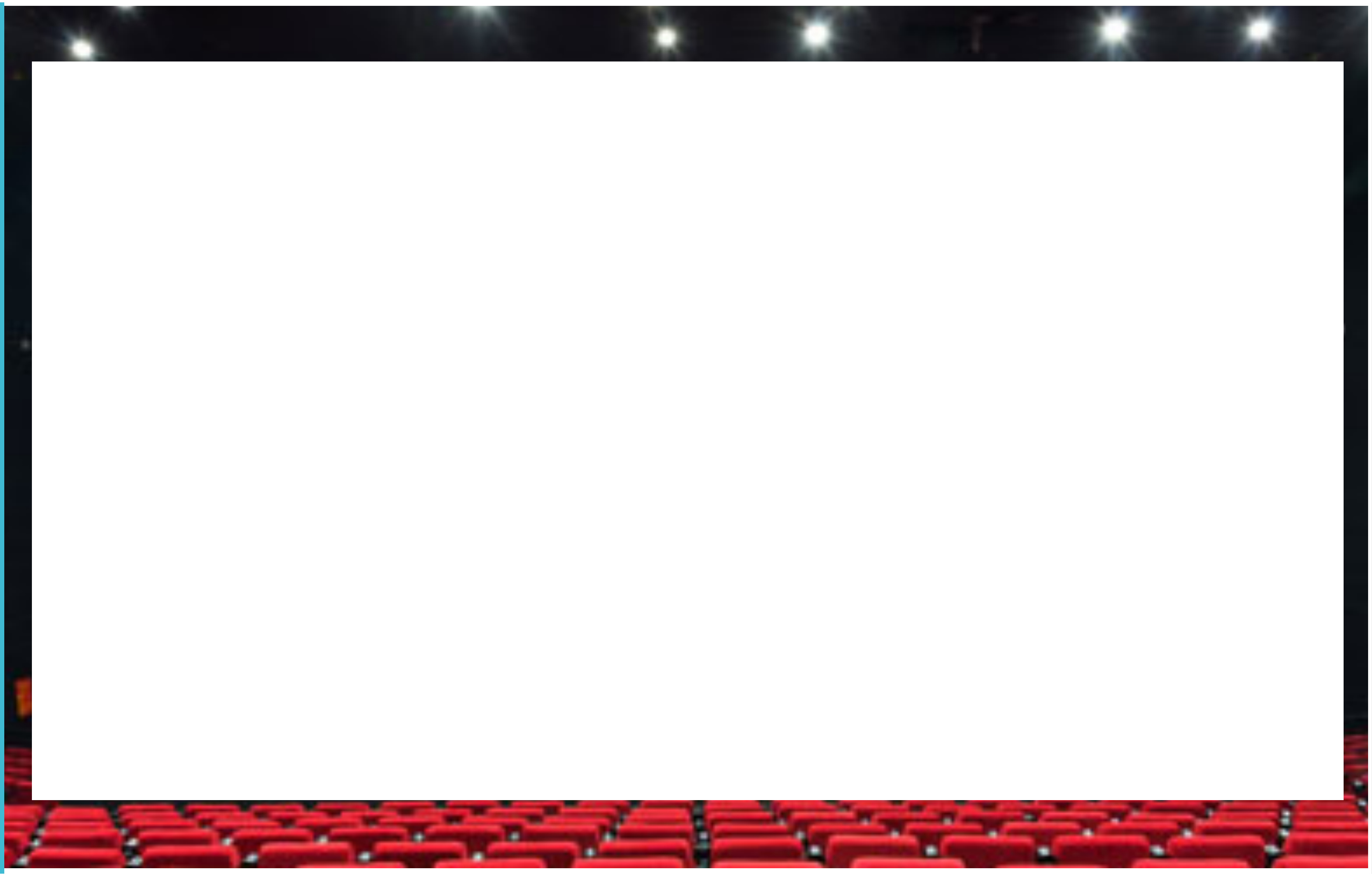


Binary Search Tree

- We create **empty child nodes** with values set to *None*.



Binary Search Tree



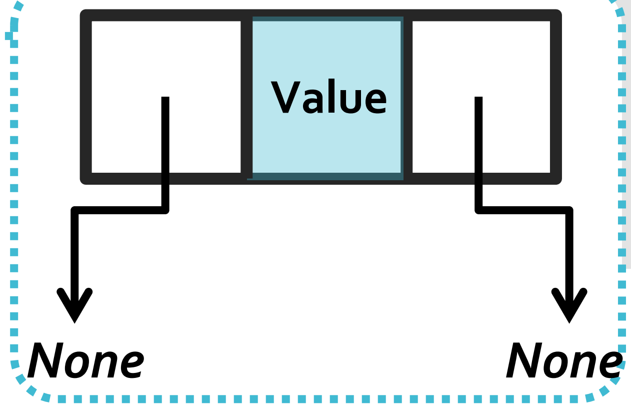
ref : youtu.be/7vw2ildqHlM

Binary Search Tree

Implementing Binary Search Tree class (part 1)

```
class BinarySearchTree:

    def __init__(self, value=None):
        self.value = value
        if self.value:
            self.left_child = BinarySearchTree()
            self.right_child = BinarySearchTree()
        else:
            self.left_child = None
            self.right_child = None
```



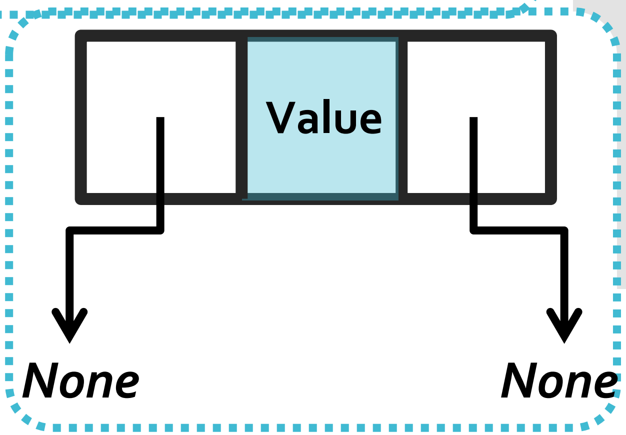
Binary Search Tree

Implementing Binary Search Tree class (part 1)

constructor

```
class BinarySearchTree:

    def __init__(self, value=None):
        self.value = value
        if self.value:
            self.left_child = BinarySearchTree()
            self.right_child = BinarySearchTree()
        else:
            self.left_child = None
            self.right_child = None
```



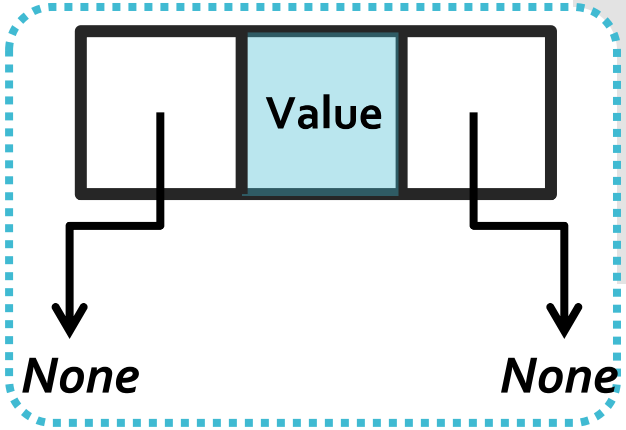
Binary Search Tree

Implementing Binary Search Tree class (part 1)

```
class BinarySearchTree:

    def __init__(self, value=None):
        self.value = value
        if self.value:
            self.left_child = BinarySearchTree()
            self.right_child = BinarySearchTree()
        else:
            self.left_child = None
            self.right_child = None
```

Creates a node with value



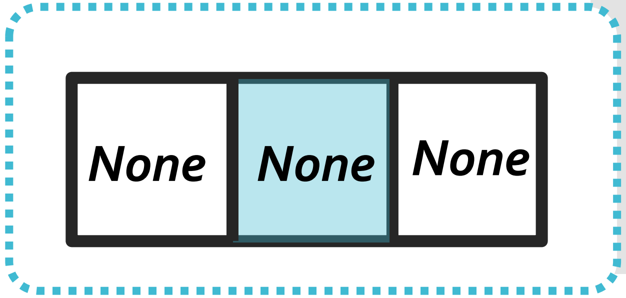
Binary Search Tree

Implementing Binary Search Tree class (part 1)

```
class BinarySearchTree:

    def __init__(self, value=None):
        self.value = value
        if self.value:
            self.left_child = BinarySearchTree()
            self.right_child = BinarySearchTree()
        else:
            self.left_child = None
            self.right_child = None
```

Creates an empty node



Binary Search Tree

Implementing Binary Search Tree class (part 1)

```python
class BinarySearchTree:

    def __init__(self, value=None):
        self.value = value
        if self.value:
            self.left_child = BinarySearchTree()
            self.right_child = BinarySearchTree()
        else:
            self.left_child = None
            self.right_child = None

    def is_empty(self):
        return self.value is None
```

checks if the node is empty

Binary Search Tree

Implementing Binary Search Tree class (part 2)

```
def insert(self, value):
    if self.is_empty():
        self.value = value
        self.left_child = BinarySearchTree()
        self.right_child = BinarySearchTree()

    elif value < self.value:
        self.left_child.insert(value)

    elif value > self.value:
        self.right_child.insert(value)
```

Binary Search Tree

Implementing Binary Search Tree class (part 2)

```
def insert(self, value):
    if self.is_empty():
        self.value = value
        self.left_child = BinarySearchTree()
        self.right_child = BinarySearchTree()

    elif value < self.value:
        self.left_child.insert(value)

    elif value > self.value:
        self.right_child.insert(value)
```

if node is empty, it adds a new node

Binary Search Tree

Implementing Binary Search Tree class (part 2)

```python
def insert(self, value):
    if self.is_empty():
        self.value = value
        self.left_child = BinarySearchTree()
        self.right_child = BinarySearchTree()

    elif value < self.value:
        self.left_child.insert(value)

    elif value > self.value:
        self.right_child.insert(value)
```

if new value is less than current node, insert to left side

Binary Search Tree

Implementing Binary Search Tree class (part 2)

```
def insert(self, value):
    if self.is_empty():
        self.value = value
        self.left_child = BinarySearchTree()
        self.right_child = BinarySearchTree()

    elif value < self.value:
        self.left_child.insert(value)

    elif value > self.value:
        self.right_child.insert(value)
```

if new value is **larger** than current node insert to right side

Binary Search Tree

Implementing Binary Search Tree class (part 3)

```
def in_order(self):
    if self.is_empty():
        return []
    else:
        return self.left_child.in_order() + \
               [self.value] + \
               self.right_child.in_order()

def print_tree(self):
    print(self.in_ord
```

recursively traverses the tree

Binary Search Tree

Implementing Binary Search Tree class (part 3)

```
def in_order(self):
    if self.is_empty():
        return []
    else:
        return self.left_child.in_order() + \
            [self.value] + \
            self.right_child.in_order()

def print_tree(self):
    print(self.in_order())
```

Prints the nodes based on in-order traverse

Binary Search Tree

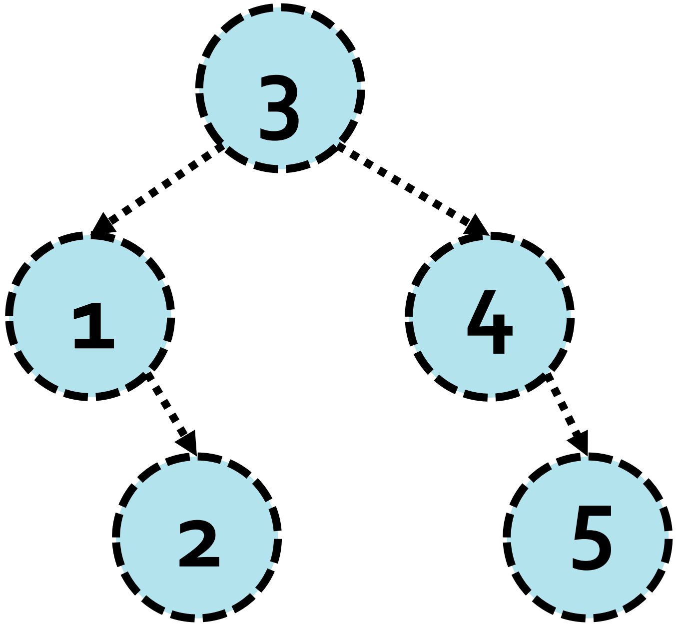
- Testing the implementation.

```
my_tree = BinarySearchTree()
my_tree.insert(3)
my_tree.insert(1)
my_tree.insert(4)
my_tree.insert(2)
my_tree.insert(5)

my_tree.print_tree()
```

[Output:]

```
[1, 2, 3, 4, 5]
```



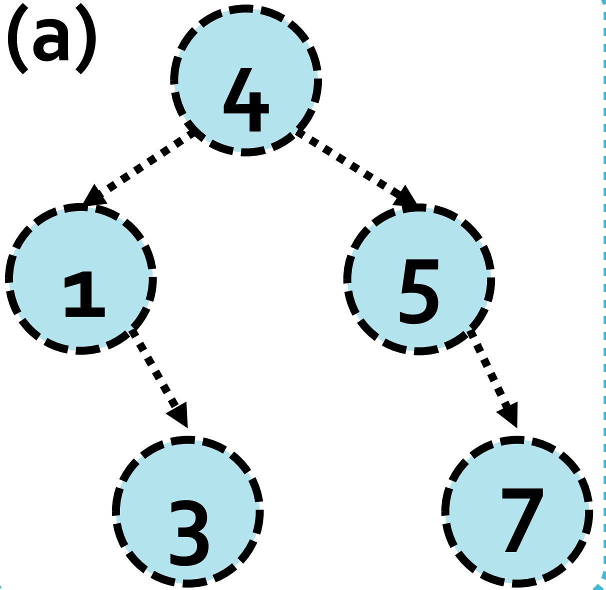
Quiz

❖Which Tree does the following code creates?

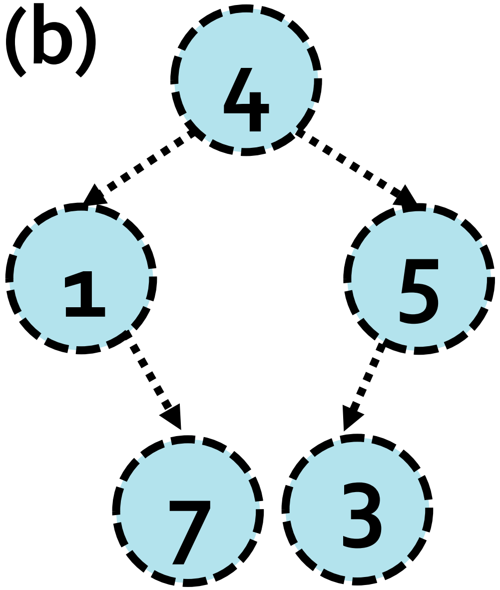
```
my_tree = BinarySearchTree()

my_tree.insert(4)
my_tree.insert(1)
my_tree.insert(5)
my_tree.insert(3)
my_tree.insert(7)
```

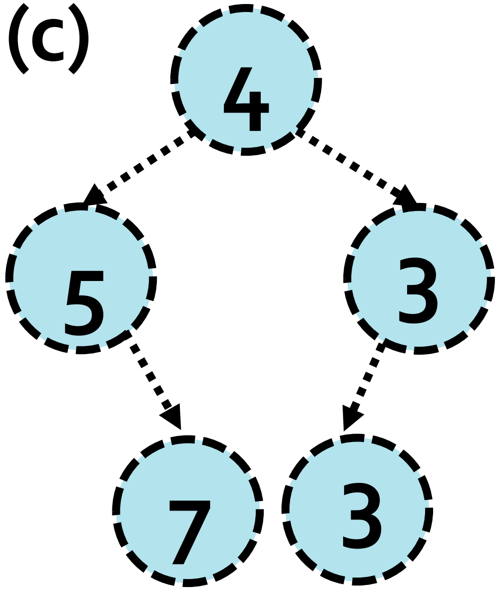
(a)



(b)



(c)

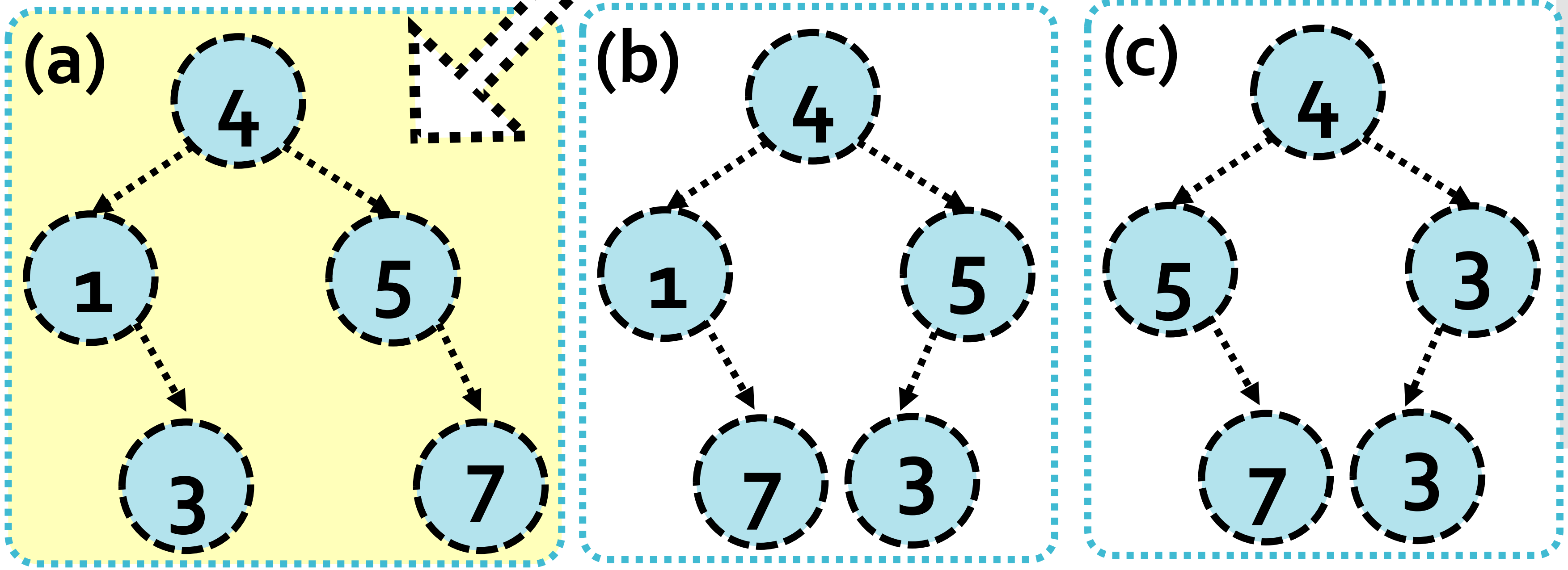


Answer

Which Tree does the following code creates?

```
my_tree = BinarySearchTree()

my_tree.insert(4)
my_tree.insert(1)
my_tree.insert(5)
my_tree.insert(3)
my_tree.insert(7)
```



Binary Search Tree

- We need a **`find(value)`** method that searches the tree [-may be- recursively?] in case whether or not a value is found.

- Then we may ended up with these **conditions**:
  - we may **find** a node with matching value.
  - we may reach a non-matching **leaf node**.

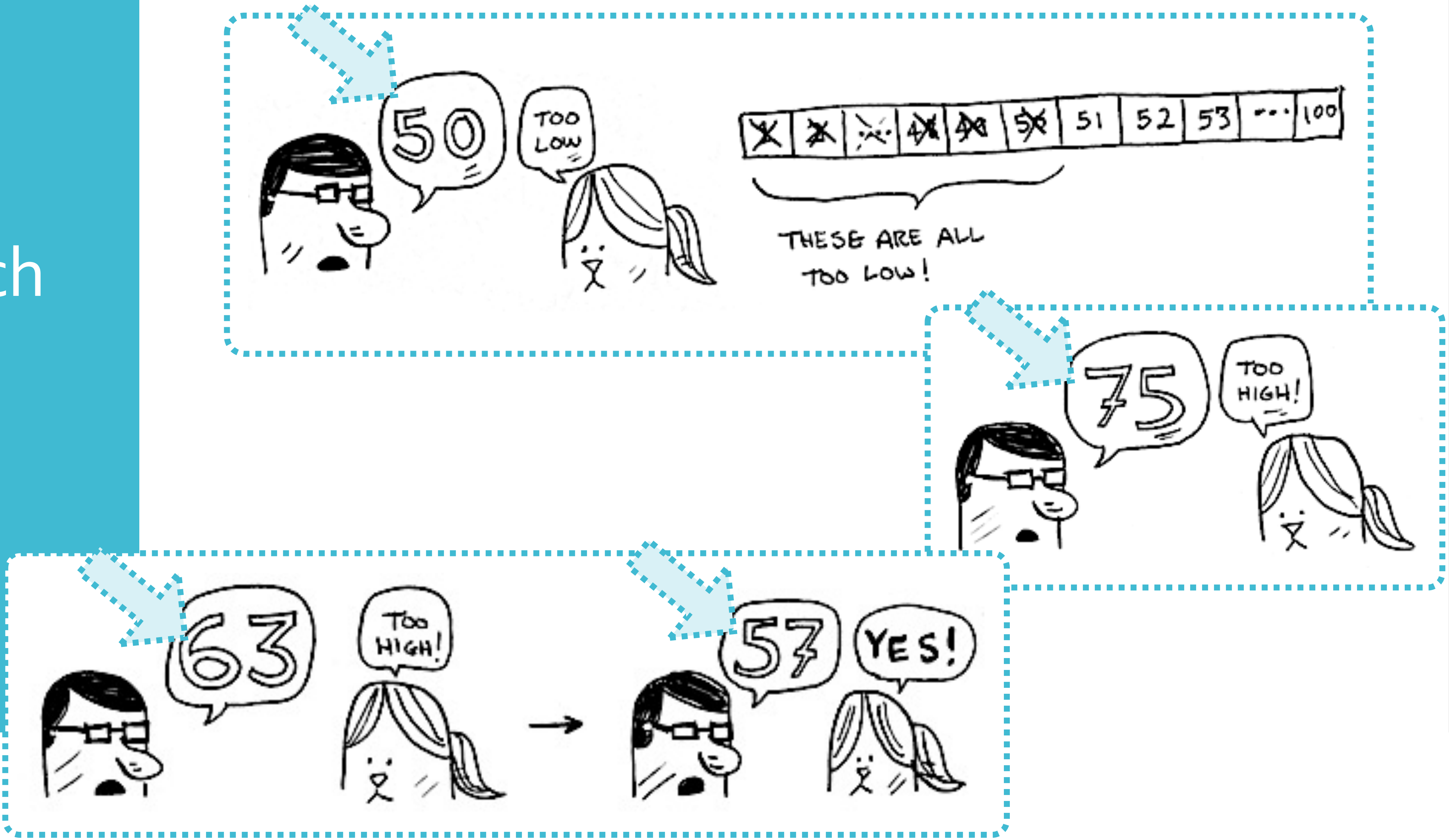
Binary Search Tree

- Suppose we are searching for value ***k***, then we:
  a) check the value of **current node (*v*)**
  b) move to the **left** child if ***k* < *v***
  c) move to the **right** child if ***k* > *v***

- Does it sound **familiar**?

Binary Search Tree

❖ Do you remember the game?



Binary Search Tree

Implementing **Binary Search Tree** class (part 4)

```python
def find(self, value):
    if self.is_empty():
        return False

    elif value == self.value:
        return True

    elif self.value > value:
        return self.left_child.find(value)

    elif self.value < value:
        return self.right_child.find(value)
```

Binary Search Tree

Implementing Binary Search Tree class (part 4)

```python
def find(self, value):
    if self.is_empty():
        return False

    elif value == self.value:
        return True

    elif self.value > value:
        return self.left_child.find(value)

    elif self.value < value:
        return self.right_child.find(value)
```

if current node is empty

Binary Search Tree

Implementing Binary Search Tree class (part 4)

```python
def find(self, value):
    if self.is_empty():
        return False

    elif value == self.value:
        return True

    elif self.value > value:
        return self.left_child.find(value)

    elif self.value < value:
        return self.right_child.find(value)
```

if current value is what we are searching for

Binary Search Tree

Implementing Binary Search Tree class (part 4)

```python
def find(self, value):
    if self.is_empty():
        return False

    elif value == self.value:
        return True

    elif self.value > value:
        return self.left_child.find(value)

    elif self.value < value:
        return self.right_child.find(value)
```

if current value is larger than search value

Binary Search Tree

Implementing Binary Search Tree class (part 4)

```
def find(self, value):
    if self.is_empty():
        return False

    elif value == self.value:
        return True

    elif self.value > value:
        return self.left_child.fi

    elif self.value < value:
        return self.right_child.find(value)
```

if current value is smaller than search value

Binary Search Tree

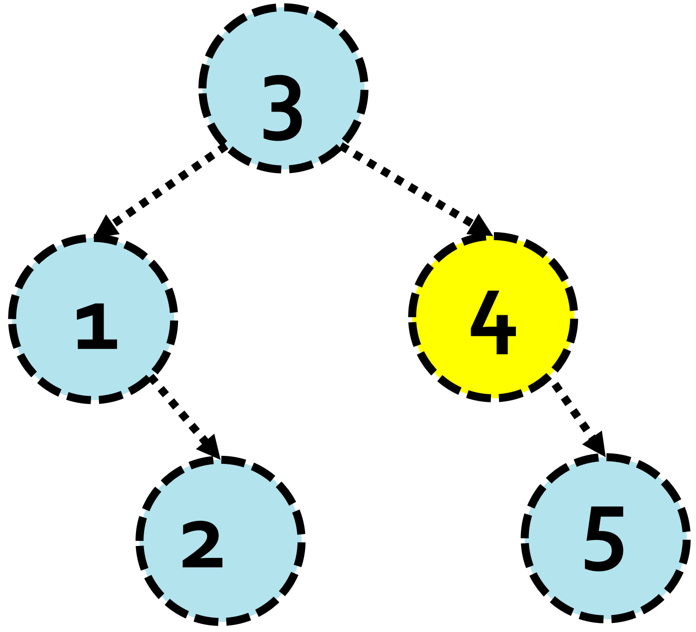
- Testing the implementation.

```
my_tree = BinarySearchTree()
my_tree.insert(3)
my_tree.insert(1)
my_tree.insert(4)
my_tree.insert(2)
my_tree.insert(5)

print('Found 4?')
print(my_tree.find(4))
```

[Output:]

```
Found 4?
True
```



Binary Search Tree

- We can extend the **Binary Search Tree** by implementing more methods.
- May be checking if we reached a **leaf note**.
- We can also implement a **copy method** for copy (for instance coping a child node).
- And also a **delete** method.

Binary Search Tree

Implementing Binary Search Tree class (part 5)

```
def is_leaf(self):
    return self.left_child.is_empty() and \
           self.right_child.is_empty()

def make_empty(self):
    self.value = None
    self.left_child = None
    self.right_child = None

def copy_child(self, child):
    if child == 'left':
        self.value = self.left_child.value
        self.right_child = self.left_child.right_child
        self.left_child = self.left_child.left_child
    elif child == 'right':
        self.value = self.right_child.value
        self.left_child = self.right_child.left_child
        self.right_child = self.right_child.right_child
```

Binary Search Tree

Implementing Binary Search Tree class (part 5)

```python
def is_leaf(self):
    return self.left_child.is_empty() and \
           self.right_child.is_empty()
```

```python
def make_empty(self):
    self.value = None
    self.left_child = None
    self.right_child = None
```

But why?
We will need it later!

```python
def copy_child(self, child):
    if child == 'left':
        self.value = self.left_child.value
        self.right_child = self.left_child.right_child
        self.left_child = self.left_child.left_child
    elif child == 'right':
        self.value = self.right_child.value
        self.left_child = self.right_child.left_child
        self.right_child = self.right_child.right_child
```

Binary Search Tree

Implementing Binary Search Tree class (part 5)

```
def is_leaf(self):
    return self.left_child.is_empty() and \
           self.right_child.is_empty()

def make_empty(self):
    self.value = None
    self.left_child = None
    self.right_child = None

def copy_child(self, child):
    if child == 'left':
        self.value = self.left_child.value
        self.right_child = self.left_child.right_child
        self.left_child = self.left_child.left_child
    elif child == 'right':
        self.value = self.right_child.value
        self.left_child = self.right_child.left_child
        self.right_child = self.right_child.right_child
```

Binary Search Tree

Implementing Binary Search Tree class (part 6)

```
def delete(self, value):
    if self.is_empty():
        print('Binary tree is empty.')

    elif value < self.value:
        self.left_child.delete(value)

    elif value > self.value:
        self.right_child.delete(value)

    elif value == self.value:
        if self.is_leaf():
            self.make_empty()
        elif self.left_child.is_empty():
            self.copy_child('right')
        else:
            self.value = self.left_child.delete_max()
```

Binary Search Tree

Implementing Binary Search Tree class (part 6)

```
def delete(self, value):
    if self.is_empty():
        print('Binary tree is empty.')

    elif value < self.value:
        self.left_child.delete(value)

    elif value > self.value:
        self.right_child.delete(value)

    elif value == self.value:
        if self.is_leaf():
            self.make_empty()
        elif self.left_child.is_empty():
            self.copy_child('right')
        else:
            self.value = self.left_child.delete_max()
```

Binary Search Tree

Implementing Binary Search Tree class (part 6)

```python
def delete(self, value):
    if self.is_empty():
        print('Binary tree is empty.')

    elif value < self.value:
        self.left_child.delete(value)

    elif value > self.value:
        self.right_child.delete(value)

    elif value == self.value:
        if self.is_leaf():
            self.make_empty()
        elif self.left_child.is_empty():
            self.copy_child('right')
        else:
            self.value = self.left_child.delete_max()
```

Binary Search Tree

Implementing Binary Search Tree class (part 6)

```python
def delete(self, value):
    if self.is_empty():
        print('Binary tree is empty.')

    elif value < self.value:
        self.left_child.delete(value)

    elif value > self.value:
        self.right_child.delete(value)

    elif value == self.value:
        if self.is_leaf():
            self.make_empty()
        elif self.left_child.is_empty():
            self.copy_child('right')
        else:
            self.value = self.left_child.delete_max()
```

Binary Search Tree

Implementing Binary Search Tree class (part 7)

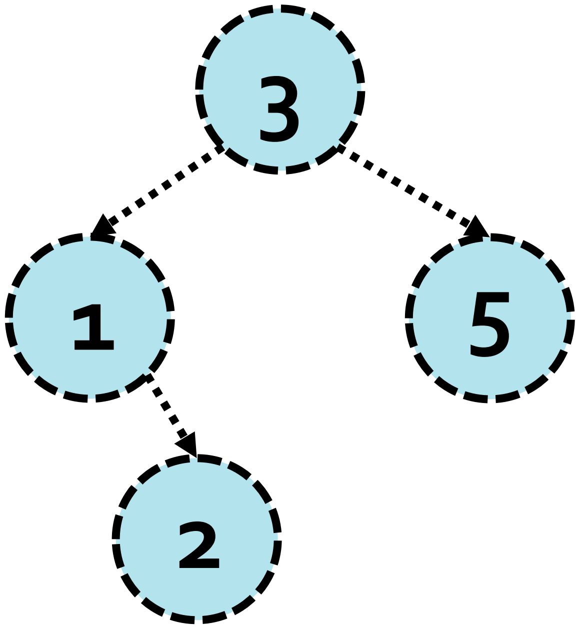
```python
def delete_max(self):
    if self.right_child.is_empty():
        max_val = self.value
        if self.left_child.is_empty():
            self.make_empty()
        else:
            self.copy_child('left')
        return max_val
    else:
        return self.right_child.delete_max()
```

Binary Search Tree

❖ Testing the implementation.

```
my_tree.delete(4)
my_tree.print_tree()

print('Found 4?')
print(my_tree.find(4))
```

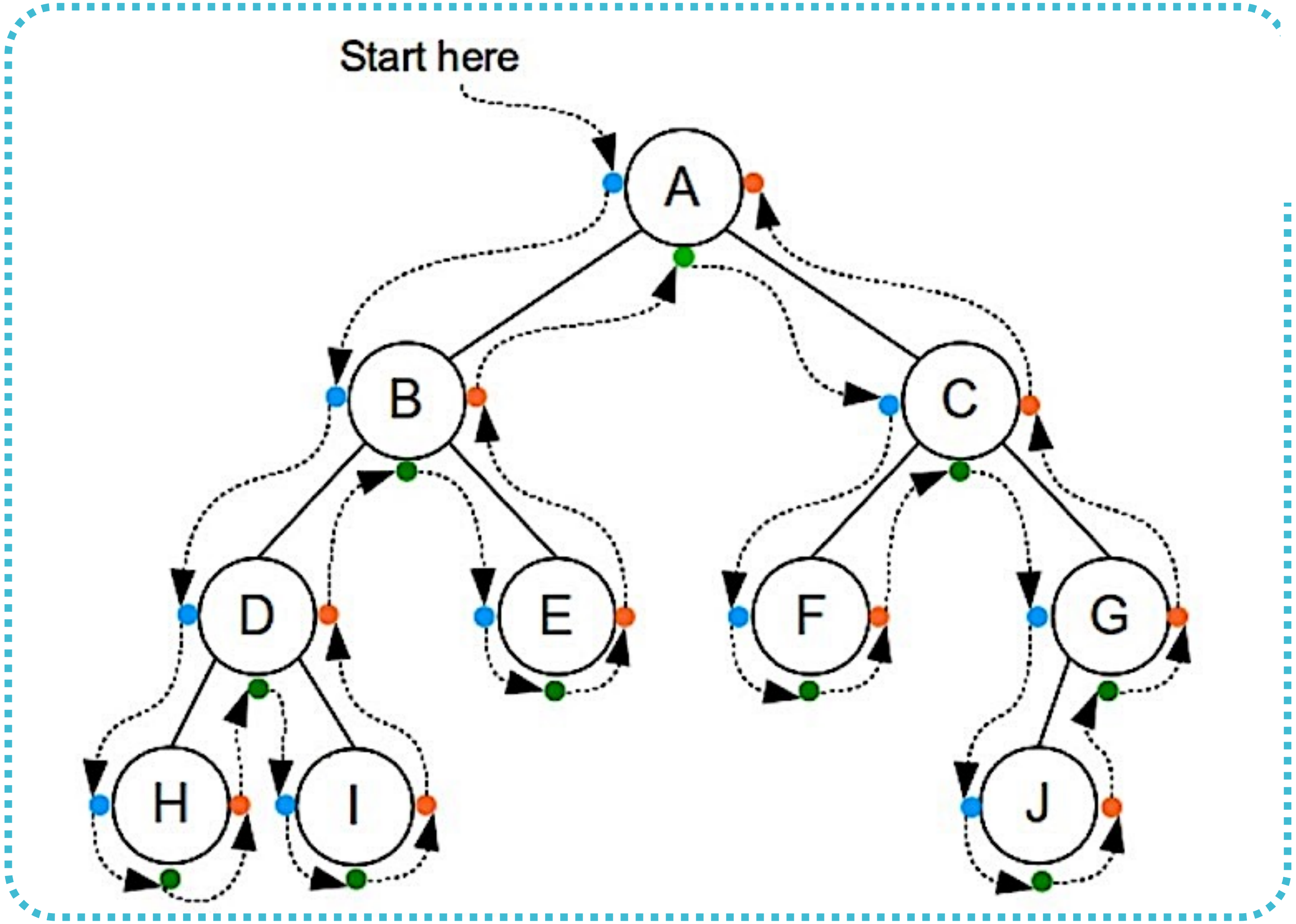


[Output:]
```
[1, 2, 3, 5]
Found 4?
False
```

Preorder, Inorder, Postorder

- This implementation has one traverse method, i.e., **inorder**!
- **Inorder** recursively do a (inorder) traversal on the left subtree, then visit the root node, and finally do a recursive (inorder) traversal of the right subtree.
- But we can have **more** ways of traverse!

Quiz

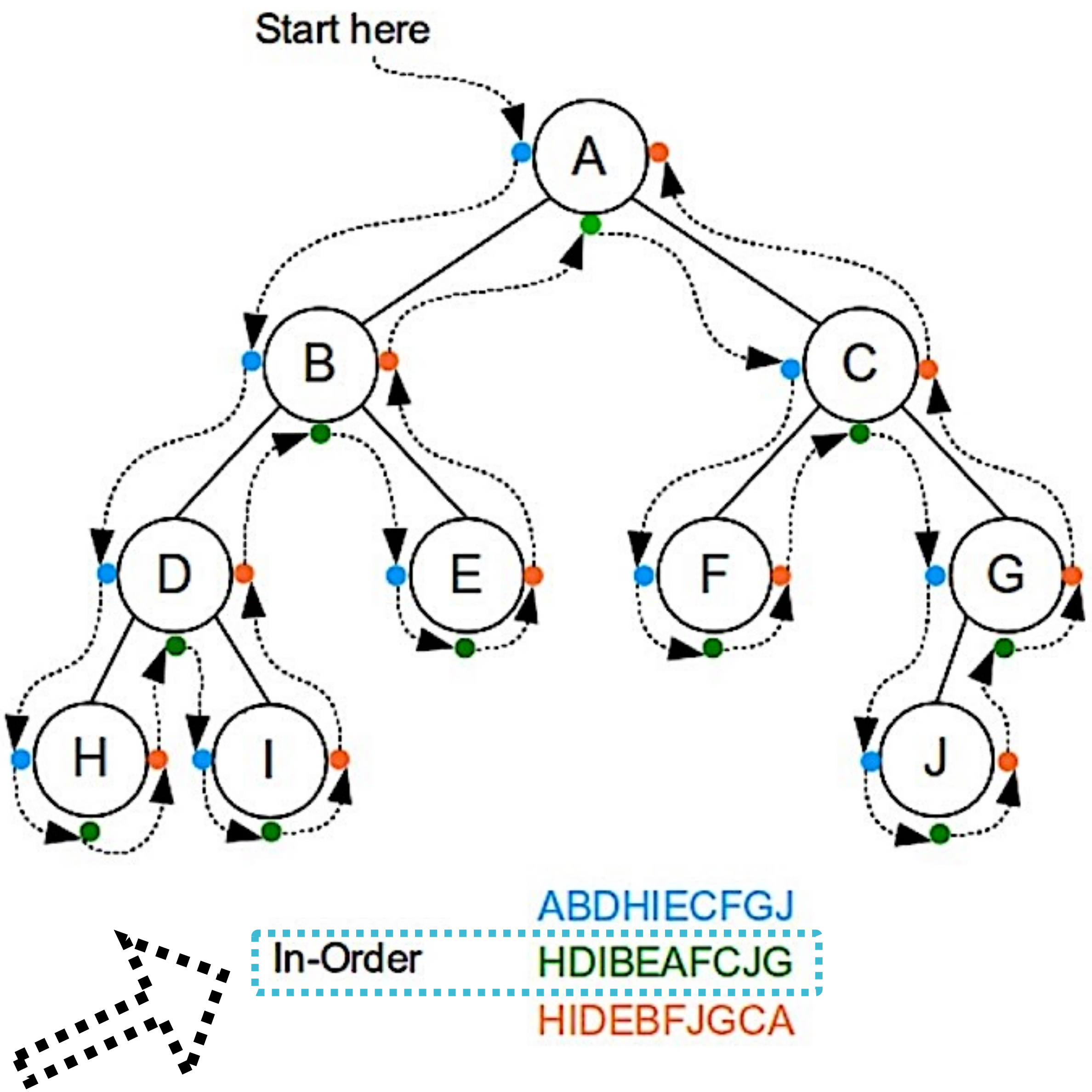


ABDHIECFGJ
HDIBEAFCJG
HIDEBFJGCA

?

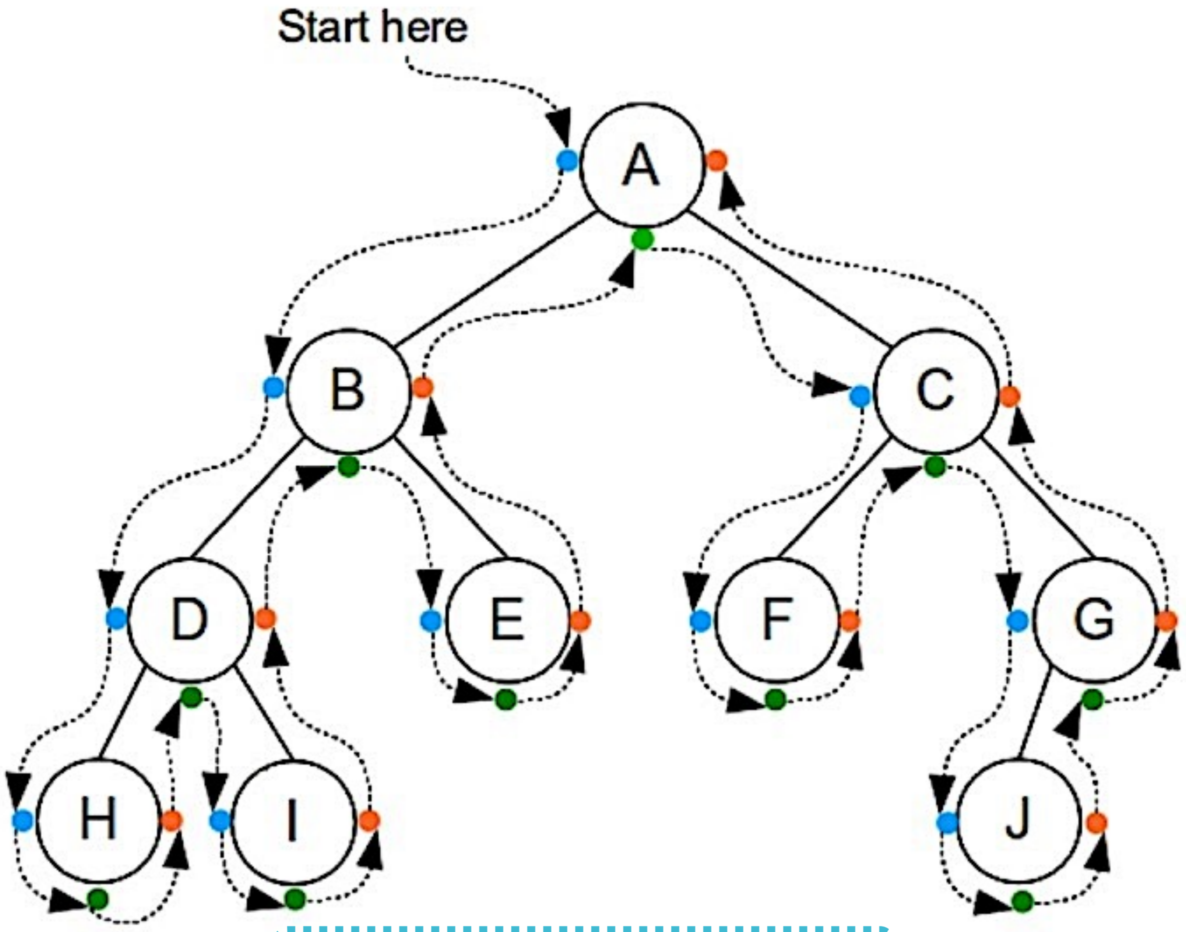
Question: Which Color indicates In-order traverse?

Answer



But what about the **other colors**?

Answer



Pre-Order	ABDHIECFGJ
In-Order	HDIBEAFCJG
Post-Order	HIDEBFJGCA

Preorder, Inorder, Postorder

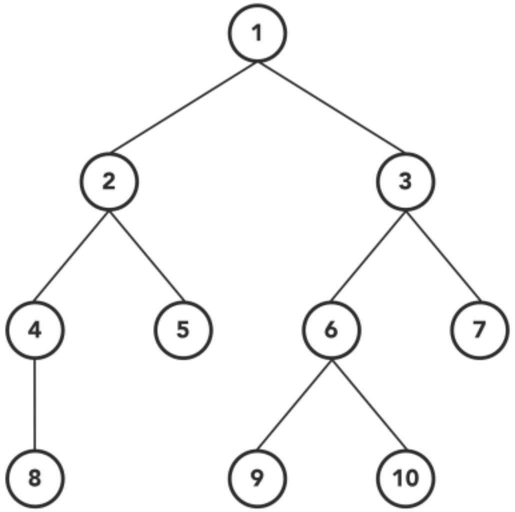
- **Preorder:** visits the root node first, then recursively do a preorder traversal of the left subtree, followed by a recursive preorder traversal of the right subtree.

- **Postorder:** recursively do a postorder traversal of the left subtree and the right subtree followed by a visit to the root node.

- Now lets see them **traversing** an example tree!

Preorder, Inorder, Postorder

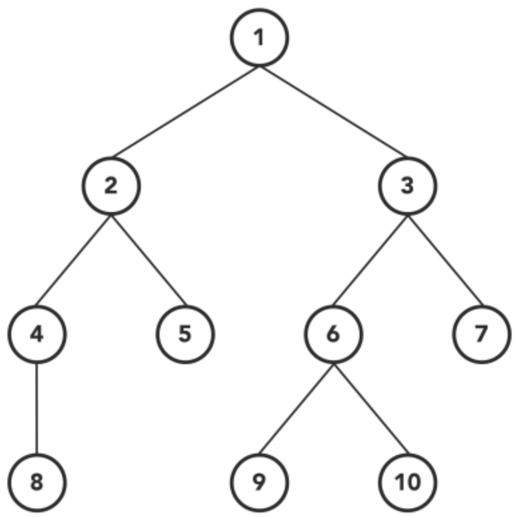
- **Preorder:**
  - visit root node,
  - go to left-subtree,
  - go to right-subtree



ref:Towardsdatascience.com

Preorder, Inorder, Postorder

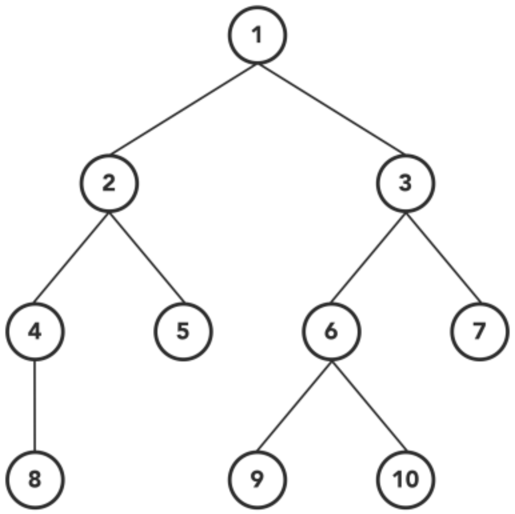
- **Inorder:**
  - go to left-subtree,
  - visit root node,
  - go to right-subtree



ref:Towardsdatascience.com

Preorder, Inorder, Postorder

- **Postorder:**
  - go to left-subtree,
  - go to right-subtree,
  - visit root node



ref:Towardsdatascience.com

Preorder, Inorder, Postorder

Implementation of Preorder and Postorder

```
class BinarySearchTree:

    def pre_order(self):
        if self.is_empty():
            return []
        else:
            return [self.value] + \
                    self.left_child.pre_order() + \
                    self.right_child.pre_order()
```

Preorder, Inorder, Postorder

Implementation of Preorder and Postorder

```
class BinarySearchTree:

    def pre_order(self):
        if self.is_empty():
            return []
        else:
            return [self.value] + \
                   self.left_child.pre_order() + \
                   self.right_child.pre_order()
```

Preorder, Inorder, Postorder

Implementation of Preorder and Postorder

```
class BinarySearchTree:

    def pre_order(self):
        if self.is_empty():
            return []
        else:
            return [self.value] + \
                   self.left_child.pre_order() + \
                   self.right_child.pre_order()
```

Preorder, Inorder, Postorder

Implementation of Preorder and Postorder

```
class BinarySearchTree:

    def pre_order(self):
        if self.is_empty():
            return []
        else:
            return [self.value] + \
                   self.left_child.pre_order() + \
                   self.right_child.pre_order()
```

Preorder,
Inorder,
Postorder

Implementation of Preorder and Postorder

```
class BinarySearchTree:

    def pre_order(self):
        if self.is_empty():
            return []
        else:
            return [self.value] + \
                   self.left_child.pre_order() + \
                   self.right_child.pre_order()
```

Implementation of Preorder and Postorder

Preorder, Inorder, Postorder

```
def post_order(self):
    if self.is_empty():
        return []
    else:
        return self.left_child.post_order() + \
               self.right_child.post_order() + \
               [self.value]
```

Preorder, Inorder, Postorder

❖Implementation of Preorder and Postorder

```
def post_order(self):
    if self.is_empty():
        return []
    else:
        return self.left_child.post_order() + \
               self.right_child.post_order() + \
               [self.value]
```

Preorder, Inorder, Postorder

❖Implementation of Preorder and Postorder

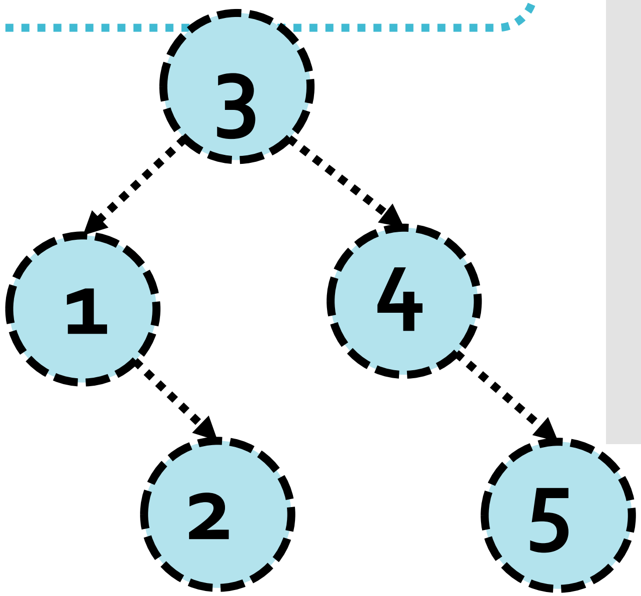
```
def post_order(self):
    if self.is_empty():
        return []
    else:
        return self.left_child.post_order() + \
               self.right_child.post_order() + \
               [self.value]
```

Preorder, Inorder, Postorder

Testing our implement

```python
my_tree = BinarySearchTree()
my_tree.insert(3)
my_tree.insert(1)
my_tree.insert(4)
my_tree.insert(2)
my_tree.insert(5)

print("Pre-order  traversal:", my_tree.pre_order())
print("In-order   traversal:", my_tree.in_order())
print("Post-order traversal:", my_tree.post_order())
```

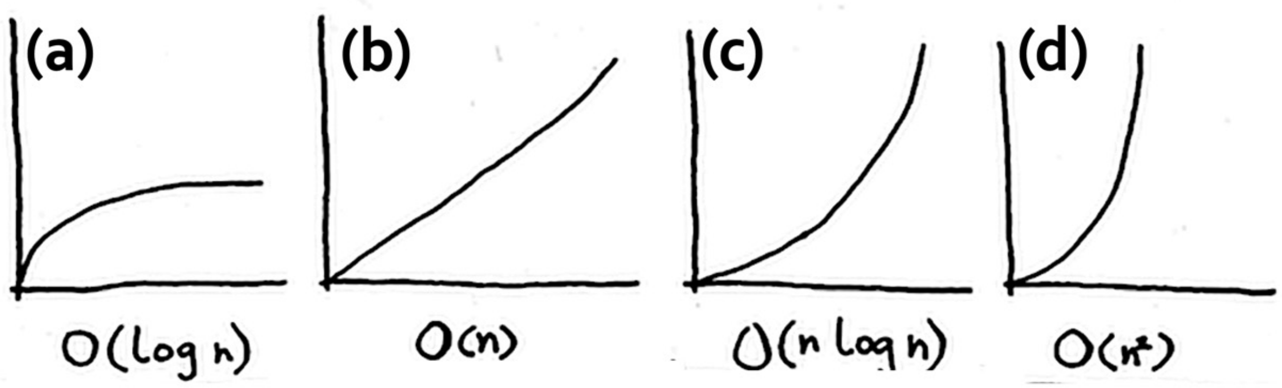


[Output:]

```
Pre-order  traversal: [3, 1, 2, 4, 5]
In-order   traversal: [1, 2, 3, 4, 5]
Post-order traversal: [2, 1, 5, 4, 3]
```

Quiz

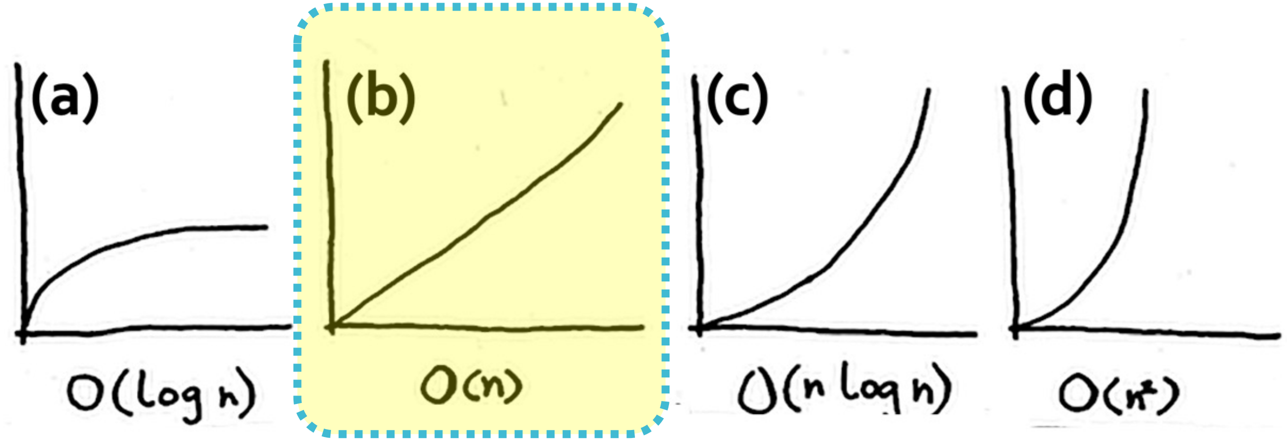
1) Which of the images represent the Big O notation for **Preorder, Inorder, Postorder**?



2) Which of the the tree traverse could be better:
   - for making a **copy of a tree**
   - for **deleting a tree** (from leaf to root)

Answer

1) Which of the images represent the Big O notation for **Preorder, Inorder, Postorder**?



- Because we traverse each node **only once**.

Answer

- Which of the the tree traverse could be better:
  - for making a **copy of a tree** --> Preorder
  - for **deleting a tree** (from leaf to root) --> Postorder

Greedy Algorithms

- **Greedy Algorithms** follow an effective **strategy** to solve problems.
- **Greedy Algorithms:** make choices that seem to be the best solution at a moment (i.e., locally-optimal) hoping that it will lead to globally-optimal solution.
- If there is a (objective) function that **needs to be optimized** (such as cost to be minimized):
  - Greedy algorithm makes choices step-by-step to ensure that the objective function is optimized.

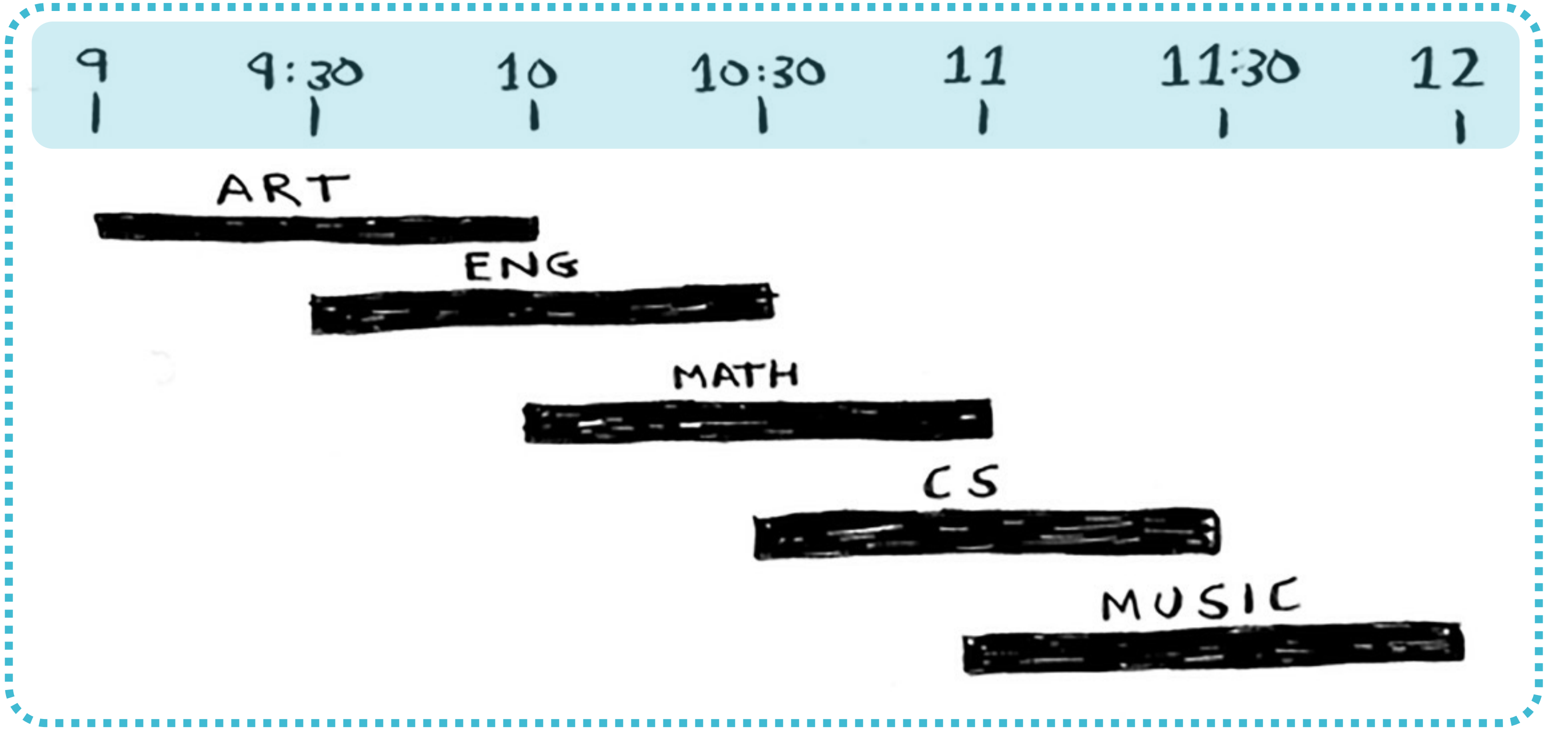
Greedy Algorithms

- Suppose you have a classroom and want to hold as **many classes here as possible**.
- This is the list of possible classes.

CLASS	START	END
ART	9 AM	10 AM
ENG	9:30 AM	10:30 AM
MATH	10 AM	11 AM
CS	10:30 AM	11:30 AM
MUSIC	11 AM	12 PM

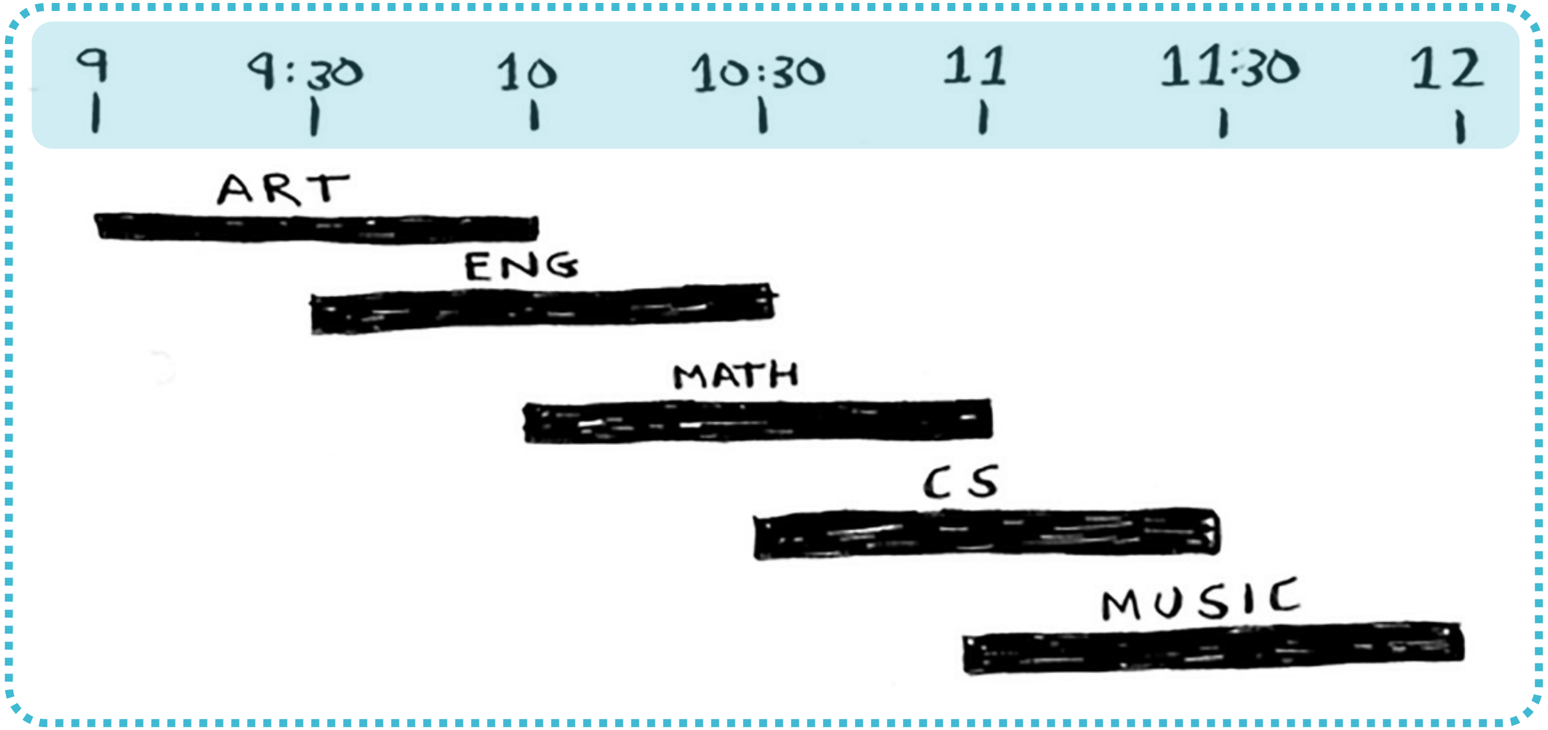
Greedy Algorithms

- You **can't hold all** of these classes in there, because some of them overlap.



Greedy Algorithms

- How do you pick what set of classes to hold, so that you get **the biggest set of classes** possible?



Greedy Algorithms

- The **algorithm** is the following:
  1. First, pick the class that **ends the soonest**. This is the first class you'll hold.

     Art ends the soonest (at 10:00 am) and we **pick it**.

CLASS	START	END	
ART	9 AM	10 AM	✓
ENG	9:30 AM	10:30 AM	
MATH	10 AM	11 AM	
CS	10:30 AM	11:30 AM	
MUSIC	11 AM	12 PM	

Greedy Algorithms

- The algorithm is the following:
    2. Then, pick a class that **starts after the first class** and ends the **soonest**. This is the second class you'll hold.

English is out because it conflicts with Art, but Math works.

CLASS	START	END	
ART	9 AM	10 AM	✓
ENG	9:30 AM	10:30 AM	✗
MATH	10 AM	11 AM	✓
CS	10:30 AM	11:30 AM	
MUSIC	11 AM	12 PM	

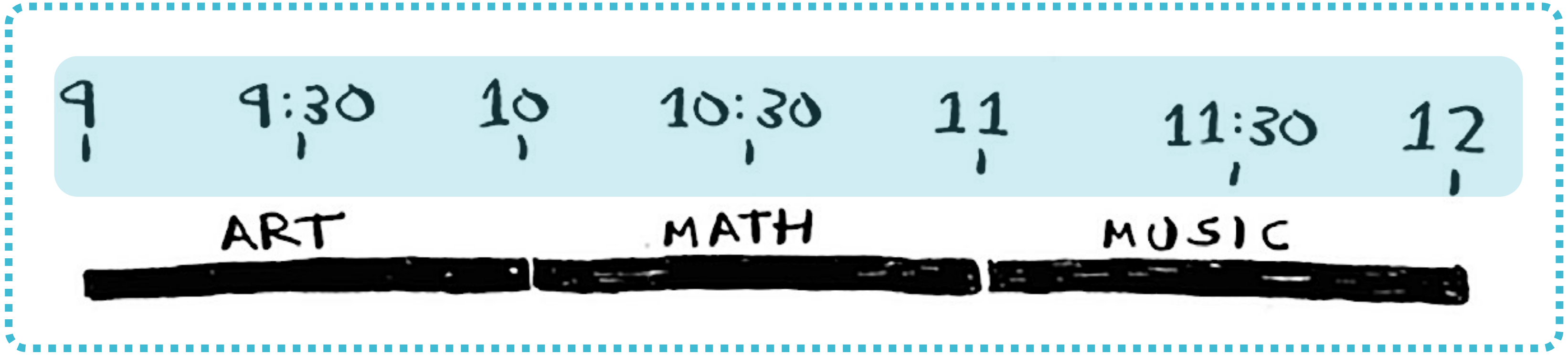
Greedy Algorithms

- The algorithm is the following:
  3. Finally, CS conflicts with Math, but Music works.

CLASS	START	END	
ART	9 AM	10 AM	✓
ENG	9:30 AM	10:30 AM	✗
MATH	10 AM	11 AM	✓
CS	10:30 AM	11:30 AM	✗
MUSIC	11 AM	12 PM	✓

Greedy Algorithms

❖ So these are the **three classes** you'll hold in this classroom.



Greedy Algorithms

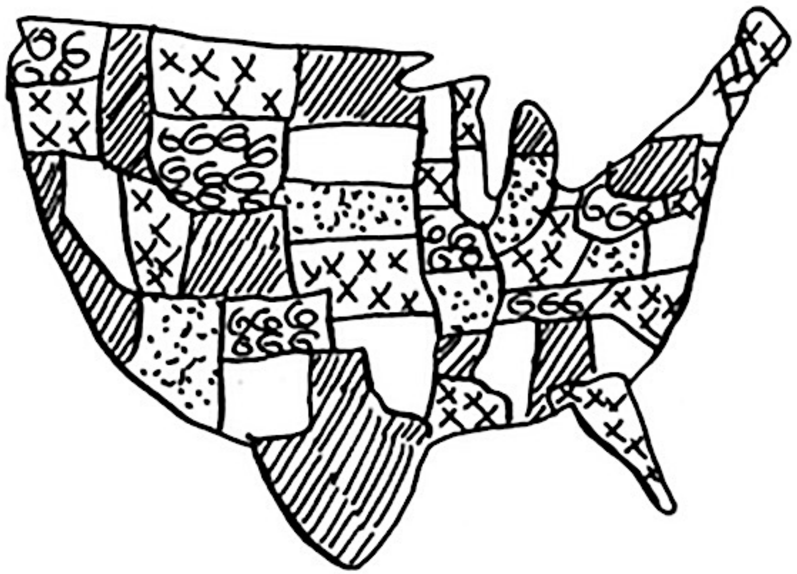
- This was a **Greedy Algorithm**, since:
  - at each step, it picks an **optimal move.**
  - in this case, it picks a class that **ends the soonest**.

- In technical terms:
  - at each step, it picks a **locally optimal** solution.
  - at the end, it reaches **globally optimal** solution.

Greedy Algorithms

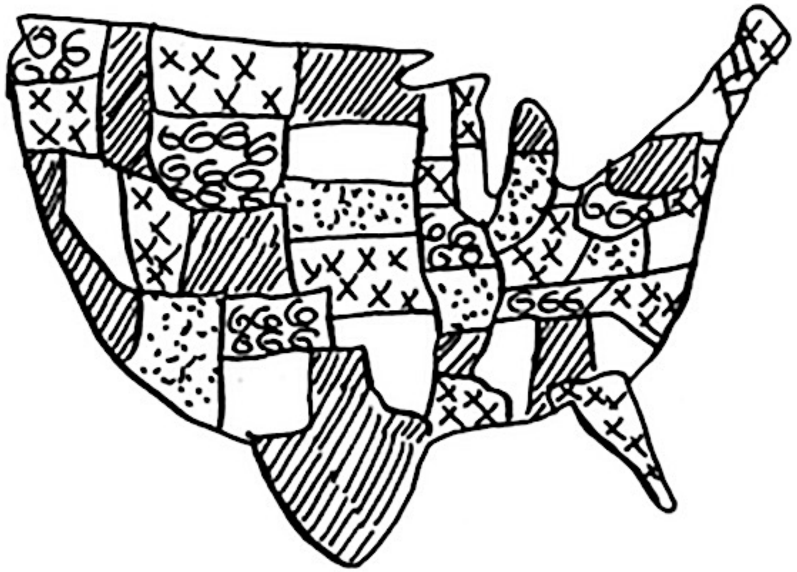
Suppose:

- You're starting a **radio show in USA.**
- You want to reach listeners in **all states.**
- But a station in a new state **will cost you.**
- You try to **minimize the cost** by choosing stations you will play on.



Greedy Algorithms

- Each station **covers a region**, and there's overlap. How to figure out the **smallest set of stations** you can play on to **cover all states**?
- Lets see how you can do it.



Greedy Algorithms

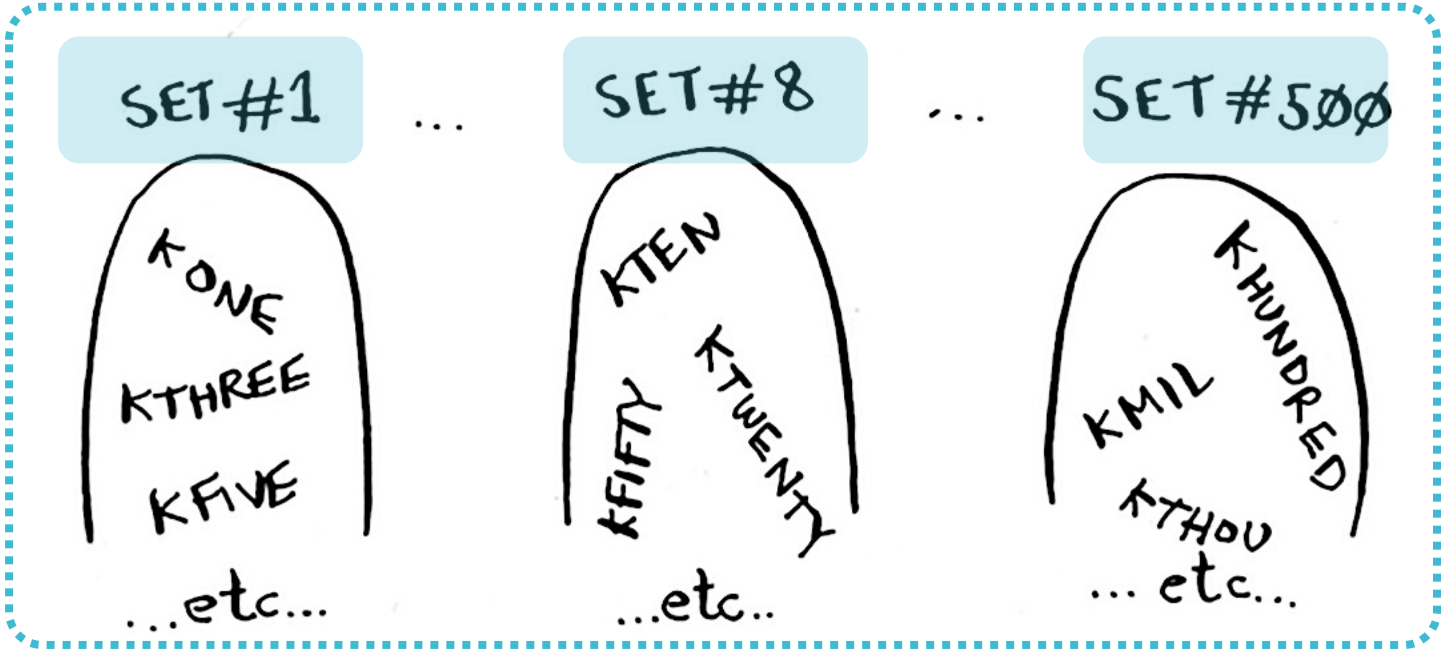
1) List every possible subset of stations. This is called **the power set**.

RADIO STATION	AVAILABLE IN
KONE	ID, NV, UT
KTWO	WA, ID, MT
KTHREE	OR, NV, CA
KFOUR	NV, UT
KFIVE	CA, AZ

...etc...

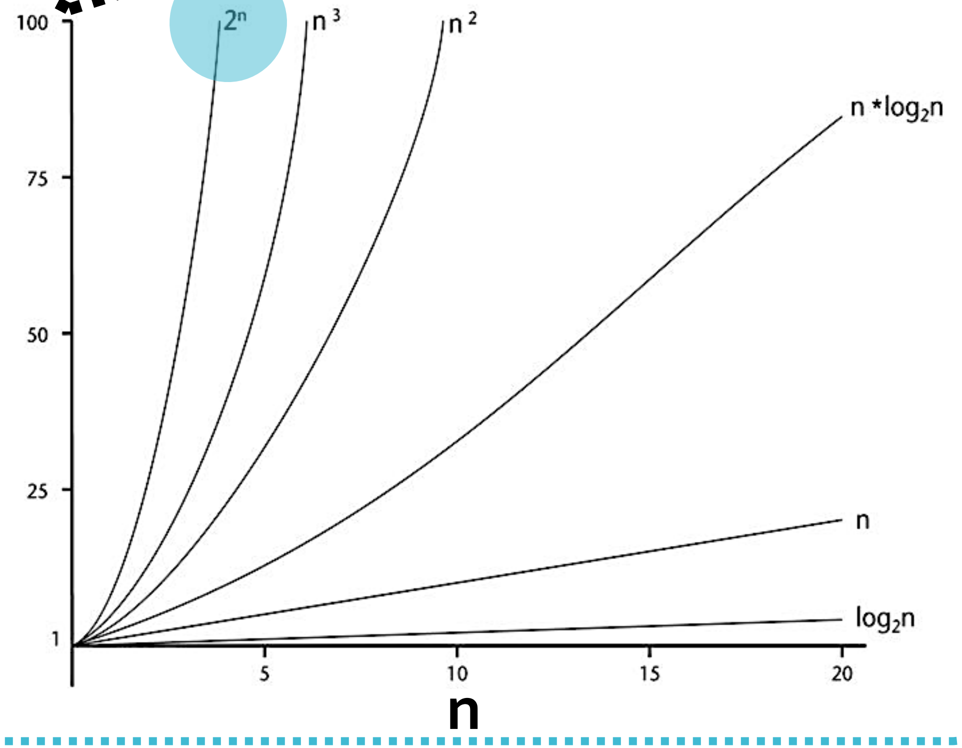
Greedy Algorithms

2) Then, pick the set with the **smallest number** of stations that **covers all states.**



Greedy Algorithms

- Can you say **how many subsets** we may have?
- There are 2^n possible subsets (**$=2^n$ stations**)!
- It will take **$O(2^n)$ time**. This really big.
- But how big?



Greedy Algorithms

- **$O(2^n)$ time**
- If you can calculate 10 subsets per second.

NUMBER OF STATIONS	TIME TAKEN
5	3.2 sec
10	102.4 sec
32	13.6 years
100	4×10^{21} years

Greedy Algorithms

- **Greedy algorithms** can be the rescue, again:
  1) pick the station that covers **the most** states, that haven't been covered yet.
  2) repeat until all the states are covered.
- This algorithm takes **$O(n^2)$ time**, where **n is the number of radio stations**.
- It is an ***approximation*** algorithm.
- If computing the exact solution takes too much time, such **approximation** algorithm **can help**.

Greedy Algorithms

- **Approximation** algorithms are judged by
  - how **fast** they are?
  - how **close** they are to the optimal solution?

- **Greedy algorithms** are good choices because:
  - they are **simple** to come up with.
  - they usually run **fast**.

Greedy Algorithms

- Lets **implement our Greedy Algorithm** to find the states, step-by-step.

Greedy Algorithms

- First, we build a **python dictionary** of stations:
  - each **key** is a **station** name
  - each **values** is the **set of states** that station covers
- **Example:** "k1" station covers:
  - `{"id", "nv", "ut"}`
  - that is Idaho (id), Nevada (nv), and Utah (ut).

```python
states_needed = {"mt", "wa", "or", "id", "nv", "ut", "ca", "az"}

stations = dict()
stations["k1"] = {"id", "nv", "ut"}
stations["k2"] = {"wa", "id", "mt"}
stations["k3"] = {"or", "nv", "ca"}
stations["k4"] = {"nv", "ut"}
stations["k5"] = {"ca", "az"}
```

Greedy Algorithms

```python
def find_states(states_needed, stations):
    final_stations = set()
    while states_needed:
        best_station = None
        states_covered = set()
```

`final_stations()`

- is a Python Set that keeps the final set of stations.

Greedy Algorithms

```
def find_states(states_needed, stations):
    final_stations = set()
    while states_needed:
        best_station = None
        states_covered = set()
```

`while` loop

- will go through all states and pick the station that covers **the most** uncovered states.
- we call this **`best_station`**

Greedy Algorithms

```
def find_states(states_needed, stations):
    final_stations = set()
    while states_needed:
        best_station = None
        states_covered = set()
```

`best_station`

- is initially set to **`None`**

`states_covered`

- is the set of all states that **this station covers** and haven't yet been covered.

Greedy Algorithms

`for` loop

- will go through all station to check which one is the **best station**.

```
for station, states_for_station in stations.items():
    covered = states_needed & states_for_station

    if len(covered) > len(states_covered):
        best_station = station
        states_covered = covered
```

Greedy Algorithms

`covered`

- is the set of uncovered states that this station covers and it is the **intersection** of two sets:
- **`states_needed`** set
- **`states_for_station`** set

```
for station, states_for_station in stations.items():
    covered = states_needed & states_for_station

    if len(covered) > len(states_covered):
        best_station = station
        states_covered = covered
```

Greedy Algorithms

`if` clause checks whether **`covered`** station has more states than the current **`best_station`**

```
for station, states_for_station in stations.items():
    covered = states_needed & states_for_station

    if len(covered) > len(states_covered):
        best_station = station
        states_covered = covered
```

Greedy Algorithms

`if` so, then:

`covered` station is the new **`best_station`**.

```
for station, states_for_station in stations.items():
    covered = states_needed & states_for_station

    if len(covered) > len(states_covered):
        best_station = station
        states_covered = covered
```

Greedy Algorithms

states_needed

- is updated by removing the states that **aren't needed** anymore.

```
    states_needed -= states_covered
    final_stations.add(best_station)

print('Final stations are:', final_stations)
```

Greedy Algorithms

when **for** loop is over:

- we add **best_station** to the final list of stations, called **final_stations**
- and print the **final_stations**

```
    states_needed -= states_covered
    final_stations.add(best_station)

print('Final stations are:', final_stations)
```

Greedy Algorithms

- **Full implementation** of the Greedy Algorithm that can find the states to cover.

```python
def find_states(states_needed, stations):
    final_stations = set()
    while states_needed:
        best_station = None
        states_covered = set()
```

Part 1

```python
        for station, states_for_station in stations.items():
            covered = states_needed & states_for_station

            if len(covered) > len(states_covered):
                best_station = station
                states_covered = covered
```

Part 2

```python
        states_needed -= states_covered
        final_stations.add(best_station)

    print('Final stations are:', final_stations)
```

Part 3

Greedy Algorithms

- Testing our implementation.

```
stations = dict()

stations["k1"] = {"id", "nv", "ut"}
stations["k2"] = {"wa", "id", "mt"}
stations["k3"] = {"or", "nv", "ca"}
stations["k4"] = {"nv", "ut"}
stations["k5"] = {"ca", "az"}

find_states(states_needed, stations)
```

[Output:]

```
Final stations are: {'k1', 'k3', 'k5', 'k2'}
```

Greedy Algorithms

- Greedy Algorithm vs Exact Algorithm

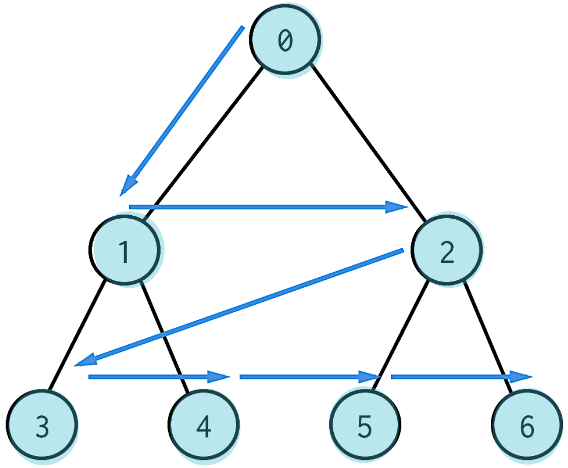
NUMBER OF STATIONS	O(n!) EXACT ALGORITHM	$O(n^2)$ GREEDY ALGORITHM
5	3.2 sec	2.5 sec
10	102.4 sec	10 sec
32	13.6 yrs	102.4 sec
100	4×10^{21} yrs	16.67 min

Quiz

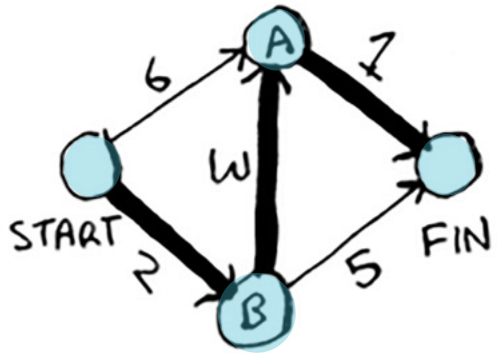
- Which of the following algorithms we discussed before, are **Greedy Algorithms**.

  a) Breadth-first search (BFS)
  b) Dijkstra's algorithm

Breadth-first search (BFS)



Dijkstra's

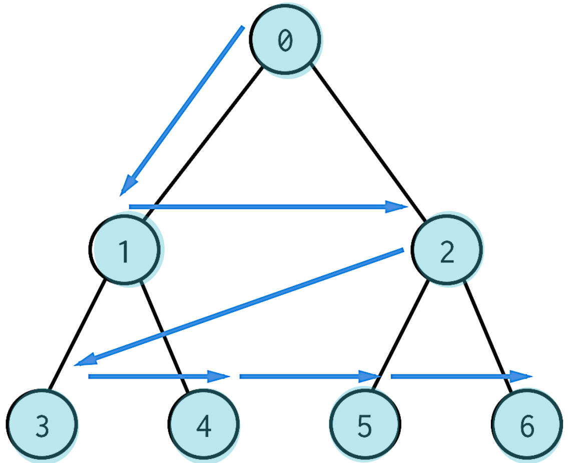


Answer

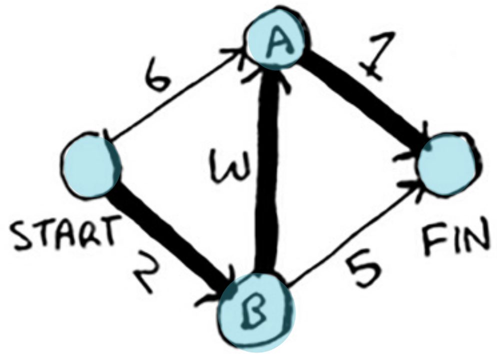
- Which of the following algorithms we discussed before, are Greedy Algorithms.

- Answer: Both of them.

Breadth-first search (BFS)



Dijkstra's



Next Lesson

- Dynamic Programming