University of Pittsburgh

Algorithms and Data Structures 2
CS 1501

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
  - Homework 3: this Friday @ 11:59 pm
  - Lab 2: next Monday @ 11:59 pm
  - Assignment 1: Monday Oct 10th @ 11:59 pm
- Please include all instructors when sending private messages on Piazza, if possible
- **Student Support Hours** of the teaching team are posted on the Syllabus page

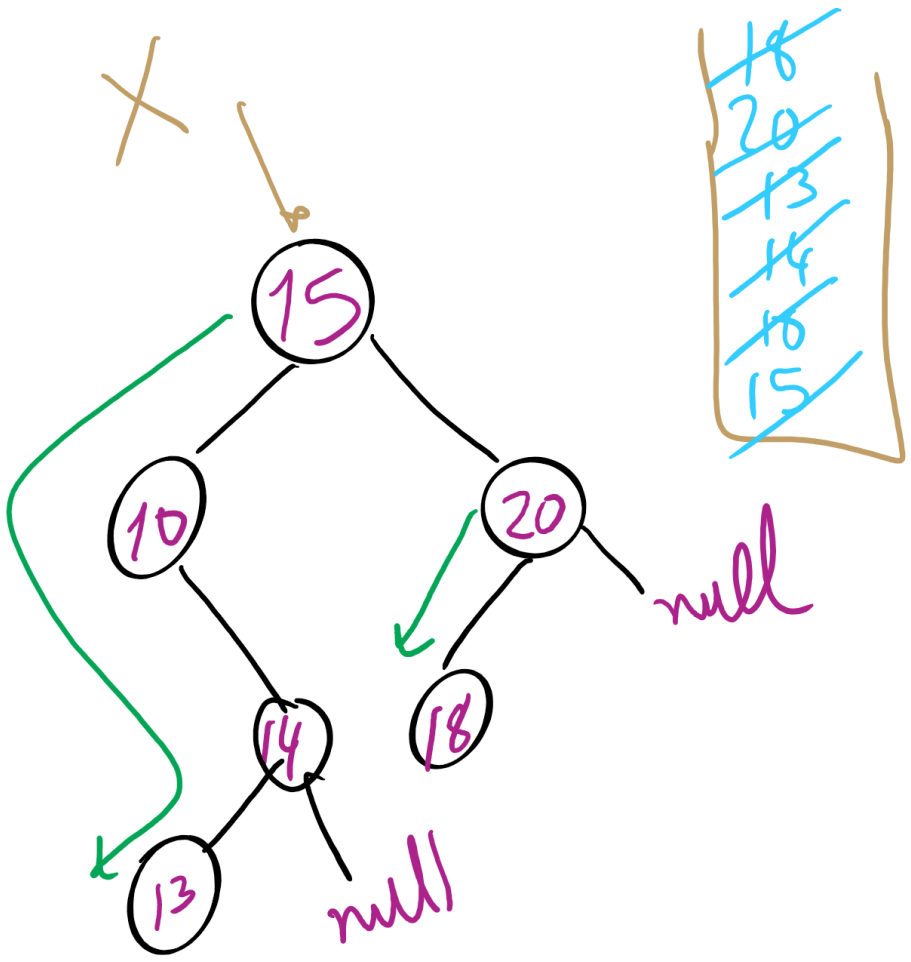
Previous lecture

- Red-Black BST (self-balancing BST)
  - definition and basic operations
  - delete
  - runtime of operations
- Turning recursive tree traversals to iterative

Traversals of a Binary Tree

- Preorder traversal
  - Visit root before we visit root's subtrees
- Inorder traversal
  - Visit root of a binary tree between visiting nodes in root's subtrees.
- Postorder traversal
  - Visit root of a binary tree after visiting nodes in root's subtrees
  - left then right then root

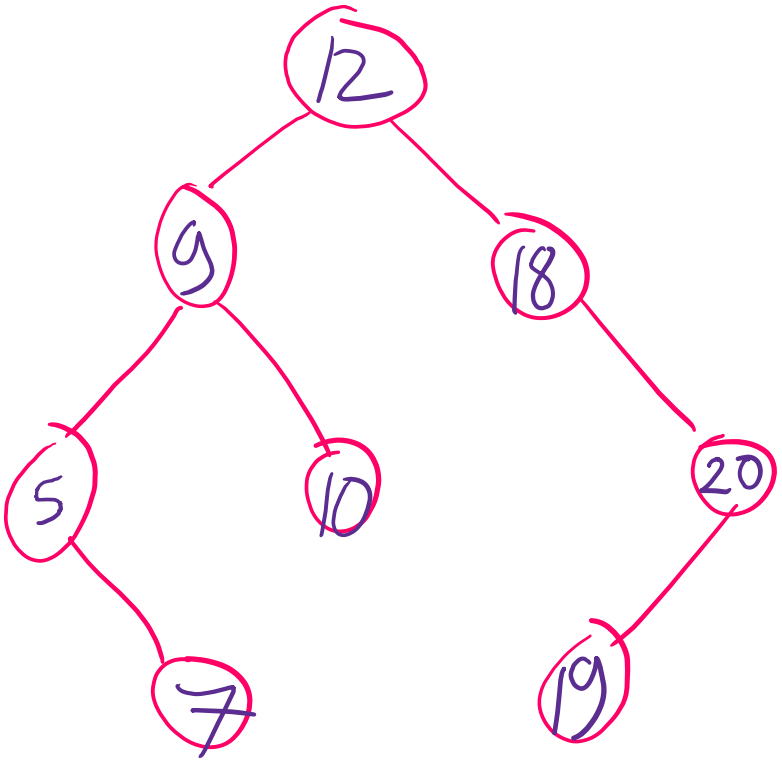
Iterative Post-order Traversal

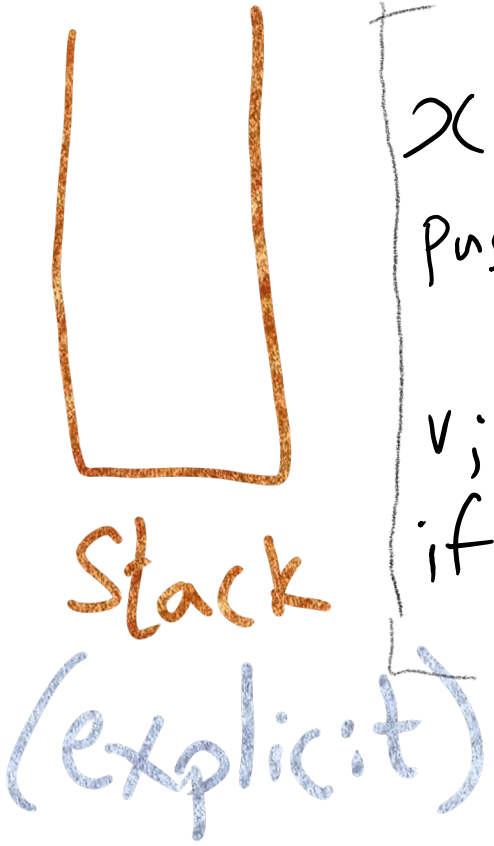


13, 14, 10, 18, 20, 15

```
stack = empty stack
x = root
While( x != null ||
       stack is not empty){
  -find leftmost leaf
  -push nodes as we
     go down the tree
  - x = pop() & visit x
  - if x is a left child
       x = right sibling
  else
       x = null
}
```

Iterative Postorder traversal





repeat until stack empty & x == null

x = leftmost leaf

Push to stack while moving down

visit x

if x is right child

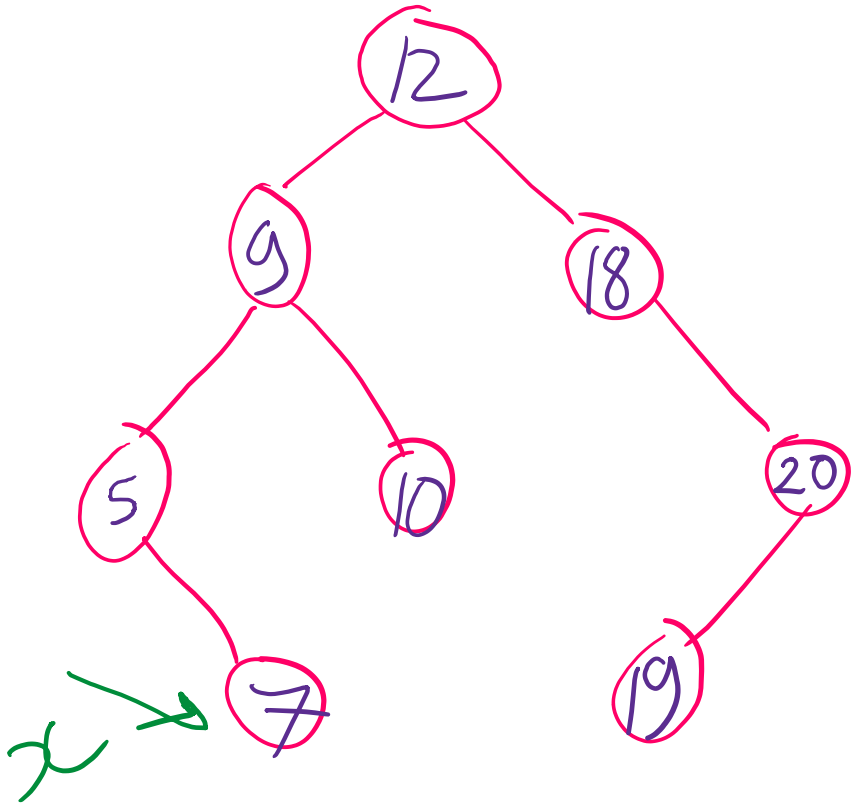
pop & visit

skip leftmost leaf

else x = parent.right

Visit order:

Iterative Postorder traversal



Stack (explicit)

repeat until stack empty & x == null

x = leftmost leaf
Push to stack while moving down

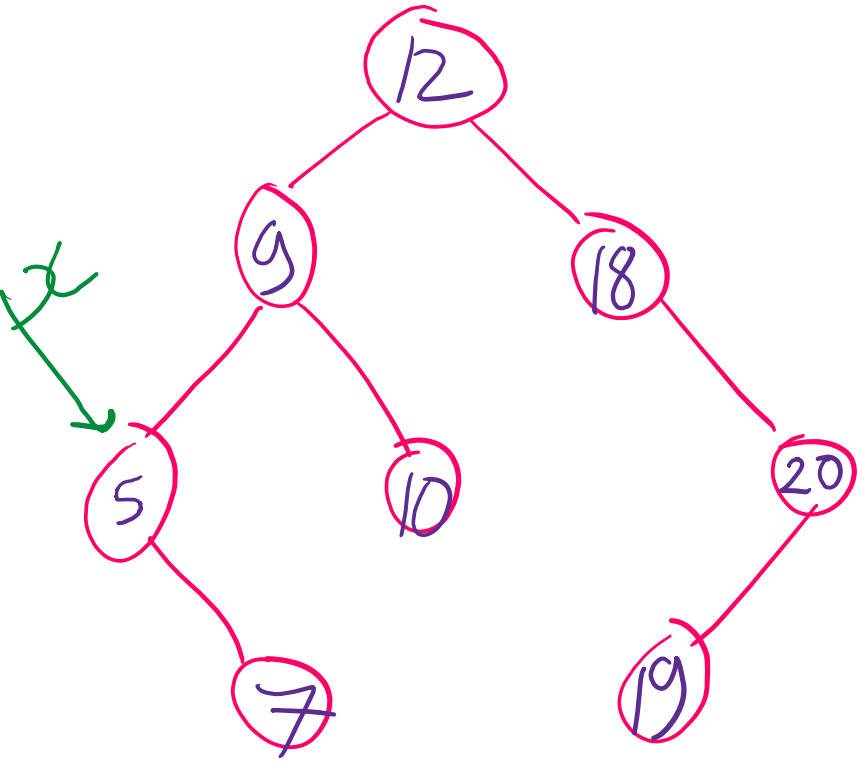
visit x

if x is rightchild
pop & visit
skip leftmost leaf

else x = parent.right

Visit order: 7

Iterative Postorder traversal



Stack (explicit)

repeat until stack empty & x == null

x = leftmost leaf

Push to stack while moving down

visit x

if x is rightchild

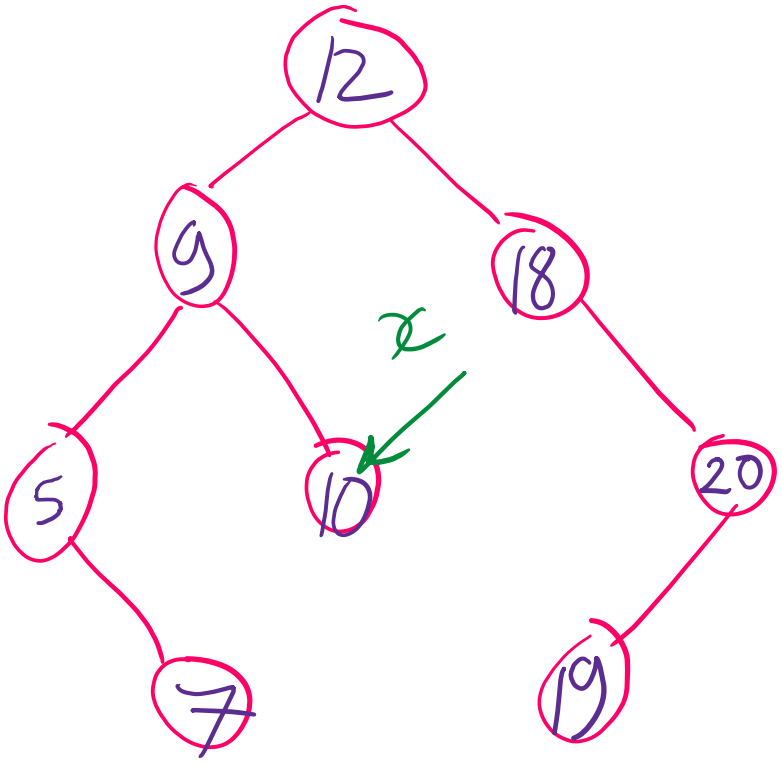
pop & visit

skip leftmost leaf

else x = parent.right

Visit order: 7, 5

Iterative Postorder traversal



Stack (explicit)

repeat until stack empty & x == null

x = leftmost leaf

Push to stack while moving down

visit x

if x is right child

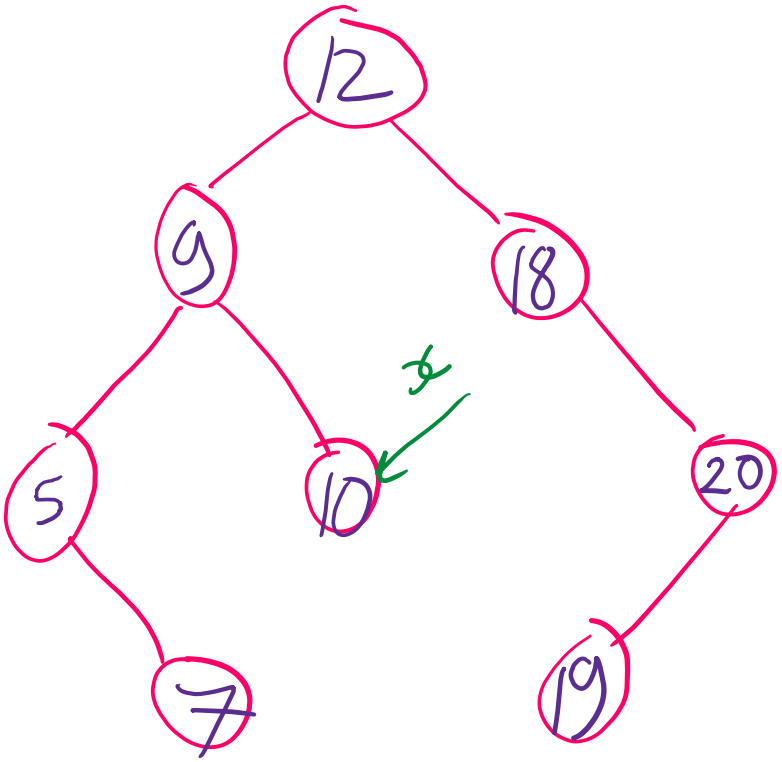
pop & visit

skip leftmost leaf

else x = parent.right

Visit order: 7, 5

Iterative Postorder traversal



Stack (explicit)

repeat until stack empty & x == null

x = leftmost leaf

Push to stack while moving down

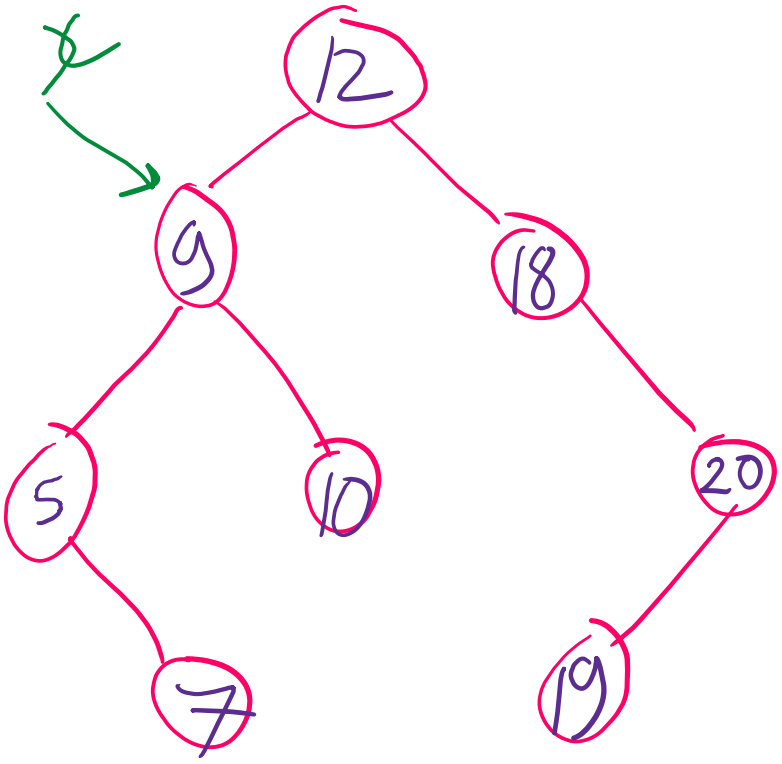
visit x

if x is right child
pop & visit
skip leftmost leaf

else x = parent.right

Visit order: 7, 5, 10

Iterative Postorder traversal



Stack (explicit)

repeat until stack empty & x == null

x = leftmost leaf
push to stack while moving down

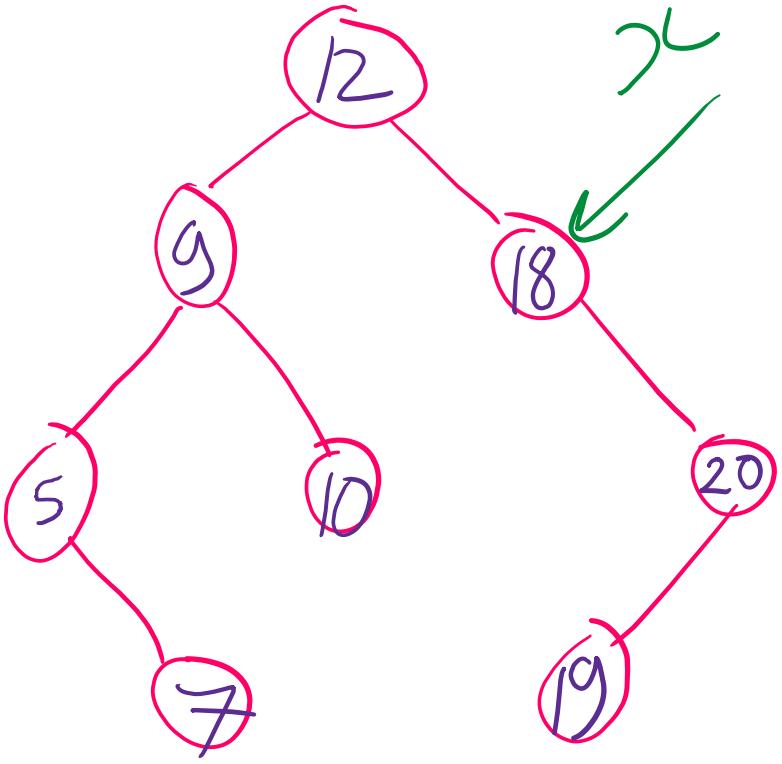
visit x

if x is right child
pop & visit
skip leftmost leaf

else x = parent.right

Visit order: 7, 5, 10, 9

Iterative Postorder traversal



Stack (explicit)

repeat until stack empty & x == null

x = leftmost leaf

Push to stack while moving down

visit x

if x is rightchild

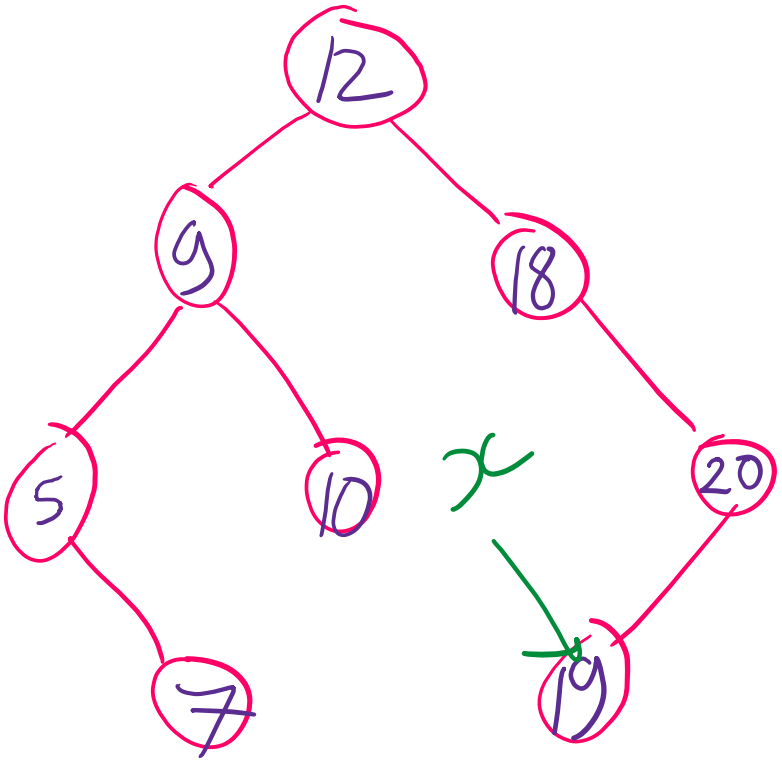
pop & visit

skip leftmost leaf

else x = parent.right

Visit order: 7, 5, 10, 9

Iterative Postorder traversal



Stack (explicit)

repeat until stack empty & x == null

x = leftmost leaf

Push to stack while moving down

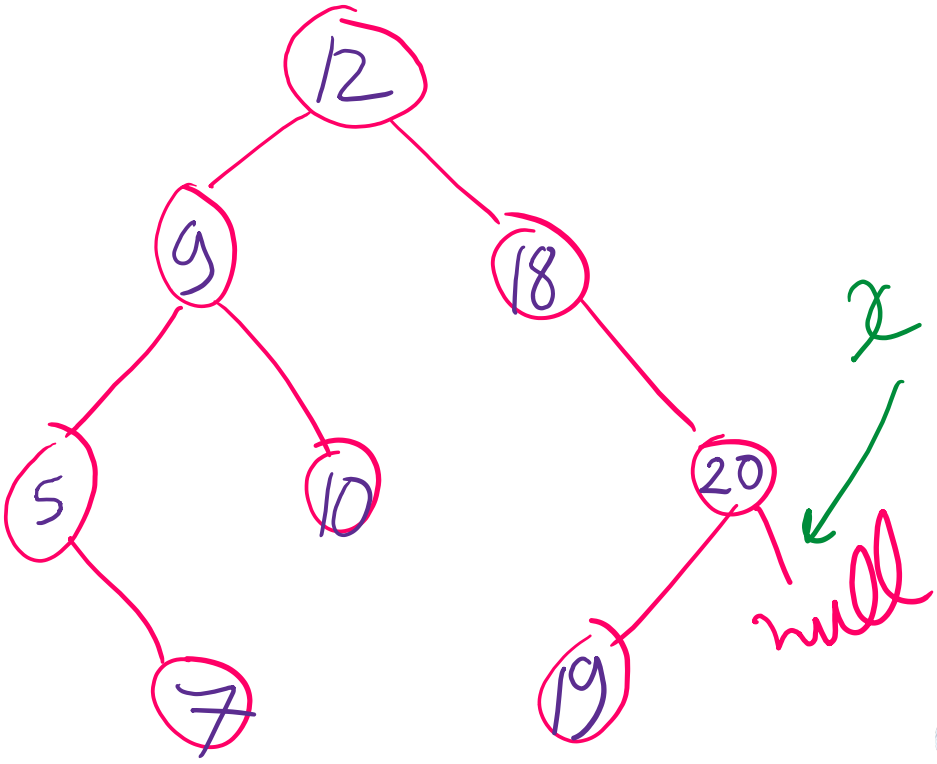
visit x

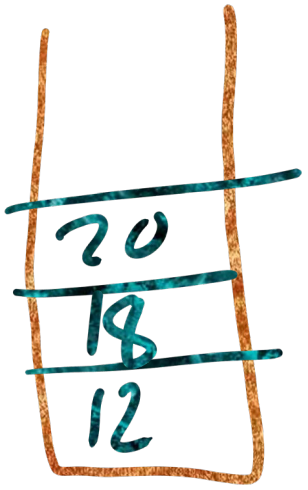
if x is right child
pop & visit
skip leftmost leaf

else x = parent.right

Visit order: 7, 5, 10, 9, 19

Iterative Postorder traversal





Stack (explicit)

repeat until stack empty & x == null

x = leftmost leaf

Push to stack while moving down

visit x

if x is right child

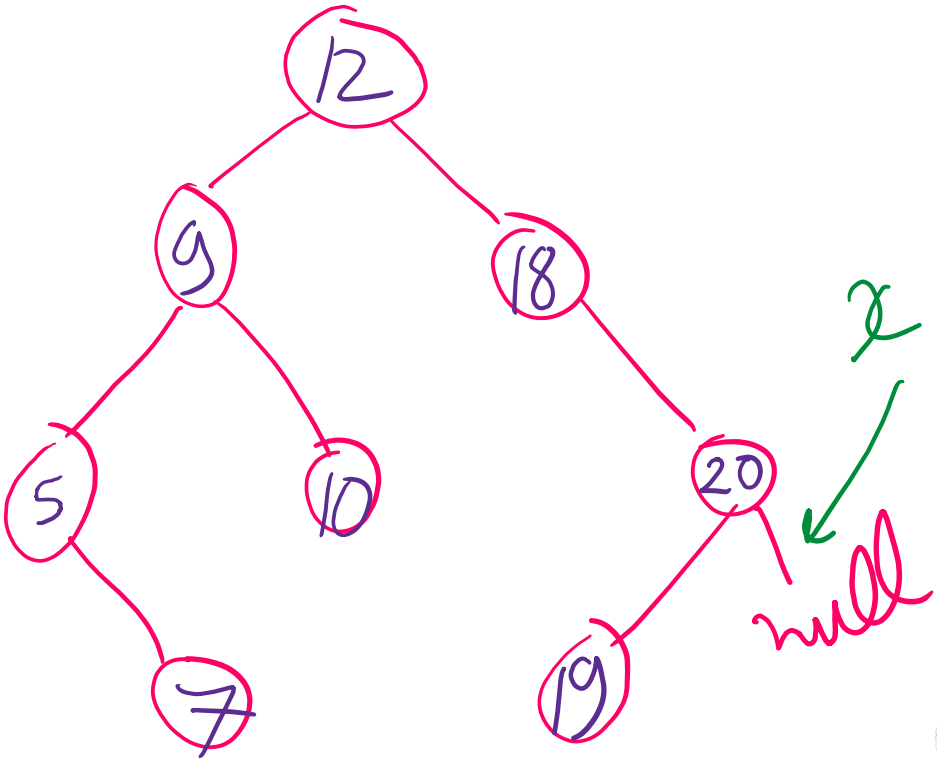
pop & visit

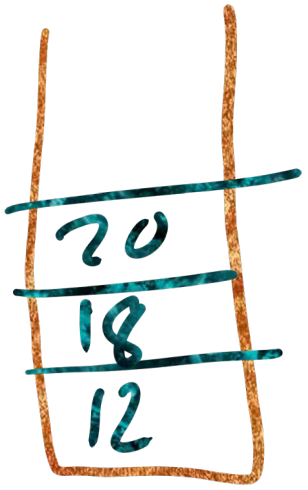
skip leftmost leaf

else x = parent.right

Visit order: 7, 5, 10, 9, 19

Iterative Postorder traversal





Stack (explicit)

repeat until stack empty & x == null

x = leftmost leaf

Push to stack while moving down

visit x

if x is right child

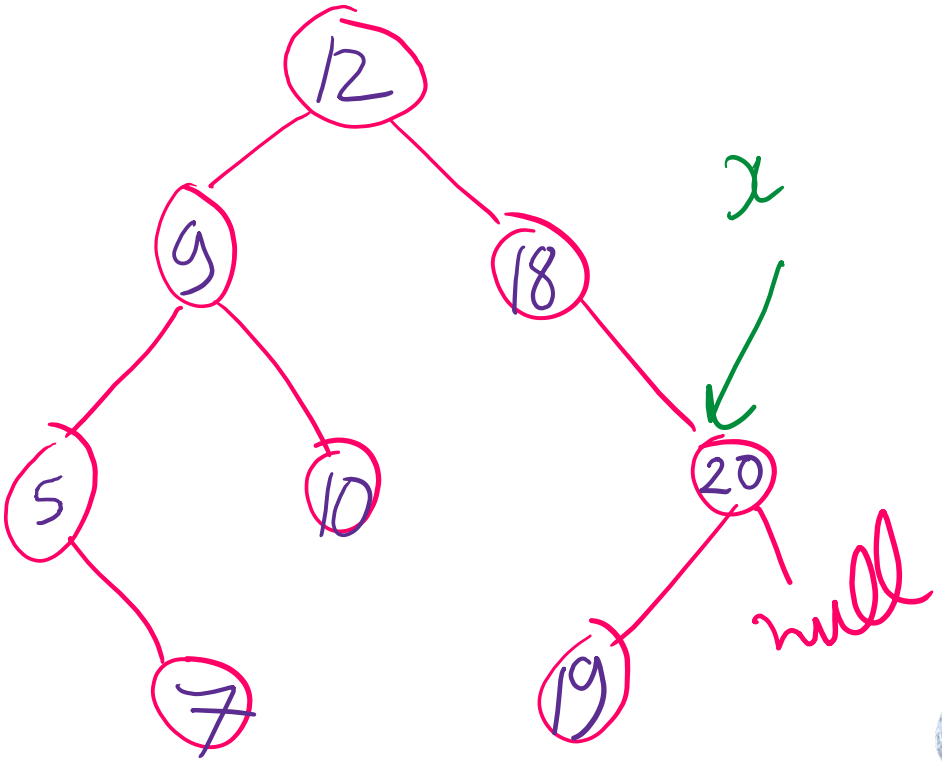
pop & visit

skip leftmost leaf

else x = parent.right

Visit order: 7, 5, 10, 9, 19

Iterative Postorder traversal



Stack (explicit)

repeat until stack empty & x == null

x = leftmost leaf

Push to stack while moving down

visit x

if x is right child

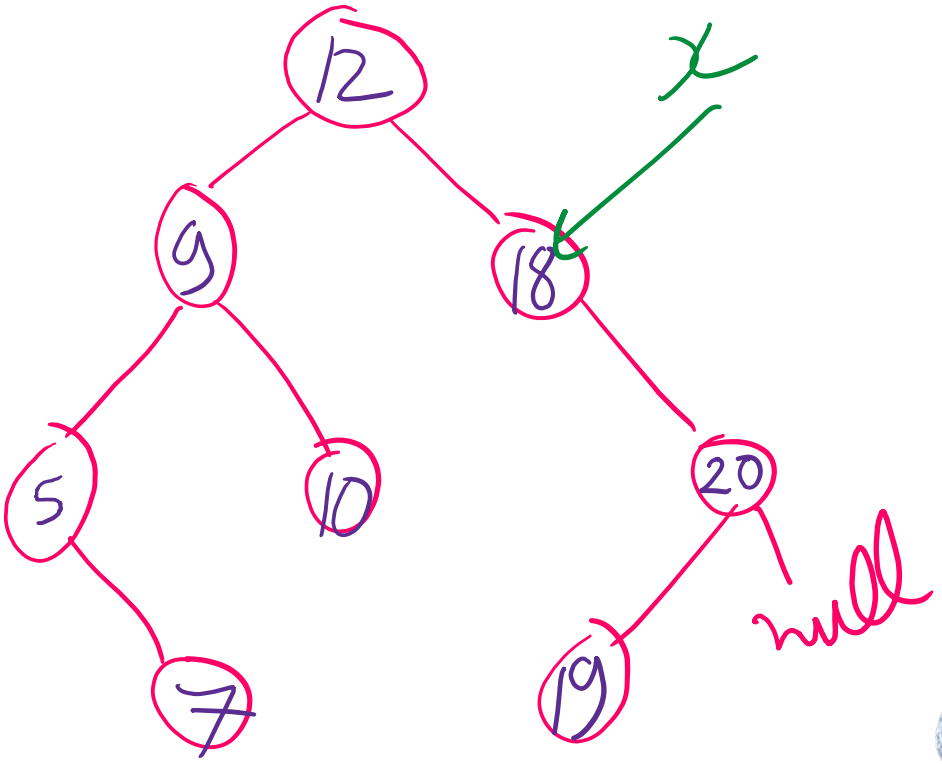
pop & visit

skip leftmost leaf

else x = parent.right

Visit order: 7, 5, 10, 9, 19, 20

Iterative Postorder traversal



Stack (explicit)

repeat until stack empty & x == null

x = leftmost leaf
push to stack while moving down

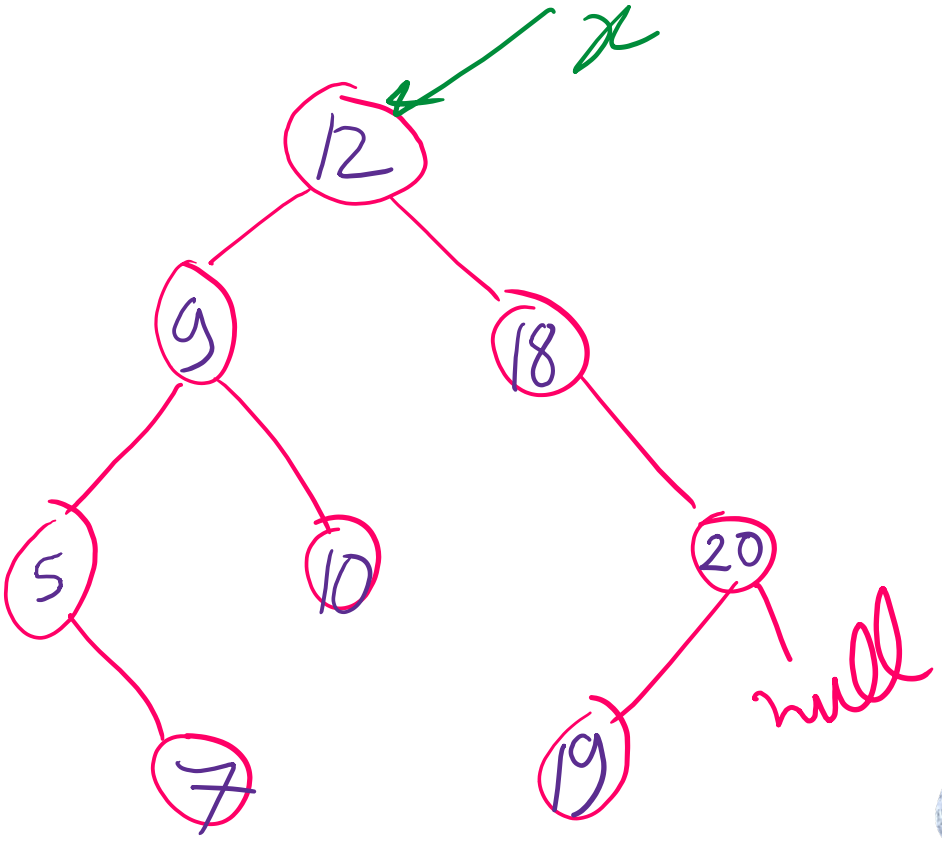
visit x

if x is right child
pop & visit
skip leftmost leaf

else x = parent.right

Visit order: 7, 5, 10, 9, 19, 20, 18

Iterative Postorder traversal



Stack (explicit)

repeat until stack empty & x == null

x = leftmost leaf
push to stack while moving down

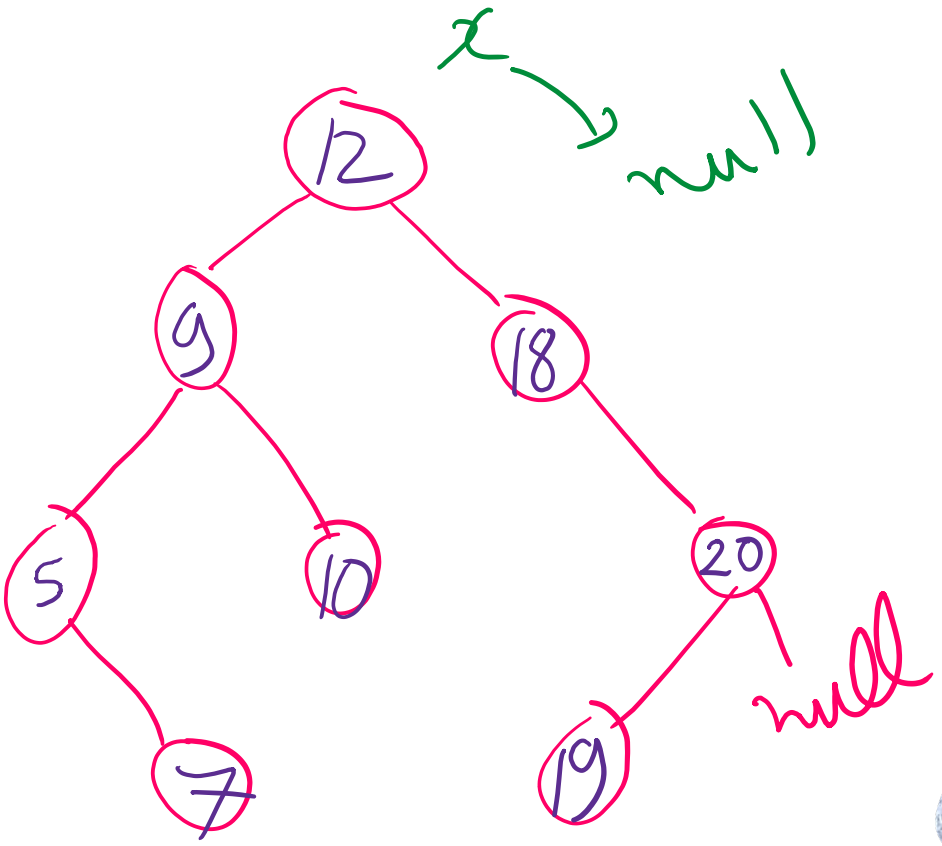
visit x

if x is right child
pop & visit
skip leftmost leaf

else x = parent.right

Visit order: 7, 5, 10, 9, 19, 20, 18, 12

Iterative Postorder traversal



Stack (explicit)

repeat until stack empty & x == null

x = leftmost leaf
Push to stack while moving down

visit x

if x is right child
pop & visit
skip leftmost leaf

else x = parent.right

Visit order: 7, 5, 10, 9, 19, 20, 18, 12

Muddiest Points

- **Q: So, just to recap: When we add a new node to a Red-Black BST, if that leads to a violation we need to fix that violation and all others back up to the tree's root before we can say we're done?**
- Yep!

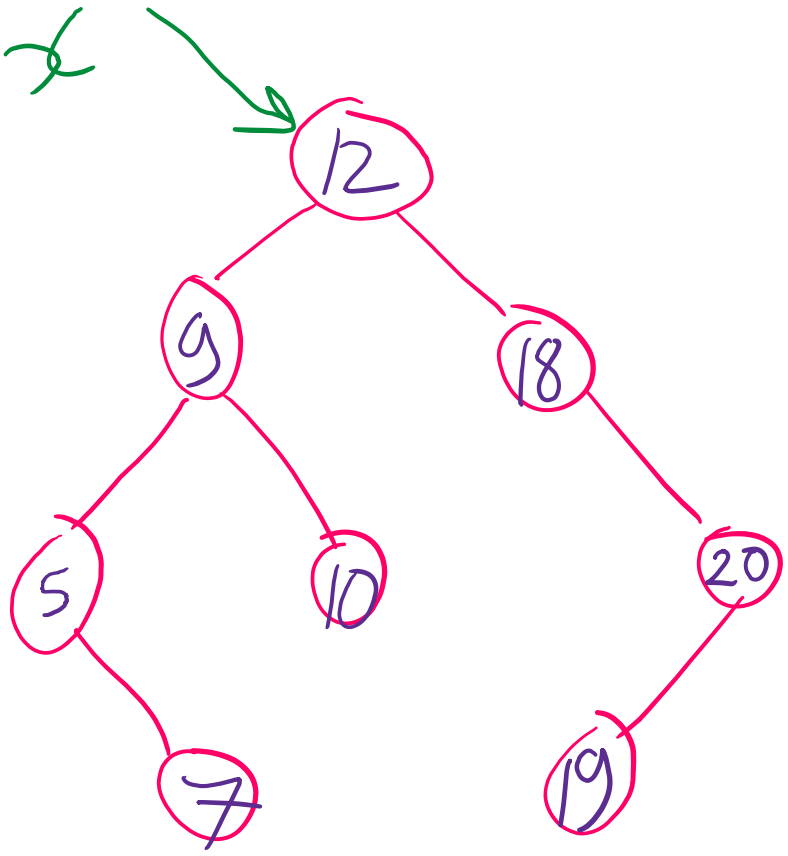
Muddiest Points

- **Q: I think there may be an overlap in which sections muddiest points you are displaying**
- Yes. But I made sure to explain the concept before going over the muddies points

Muddiest Points

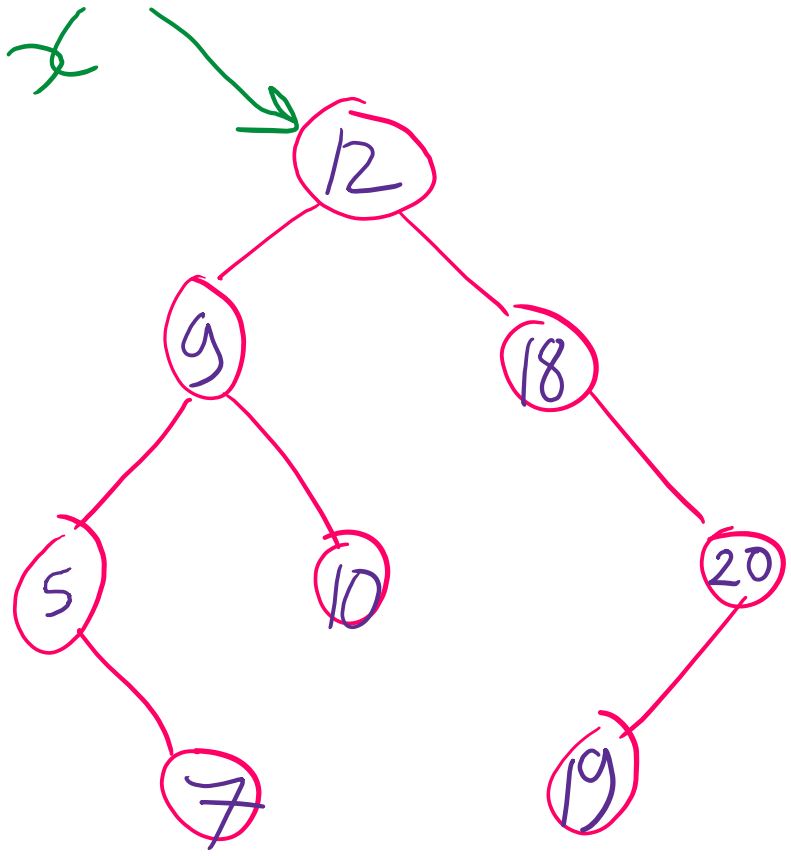
- **Q: how does a tree map work? a hashmap works effectively by 'converting' a string to an int by using a hash code to map a key to a value, what is a treemap and how does it do that?**
- Both TreeMap and HashMap implement the Map or Dictionary interface
- Nothing in the Map interface requires the conversion into an integer
- TreeMap uses a Binary Search Tree instead of a hash table to perform add, search, delete, etc.

BST search using iteration




```
x = root
while(x != null){
    if equal break;
    if < x = x.left
    if > x = x.right
}
```

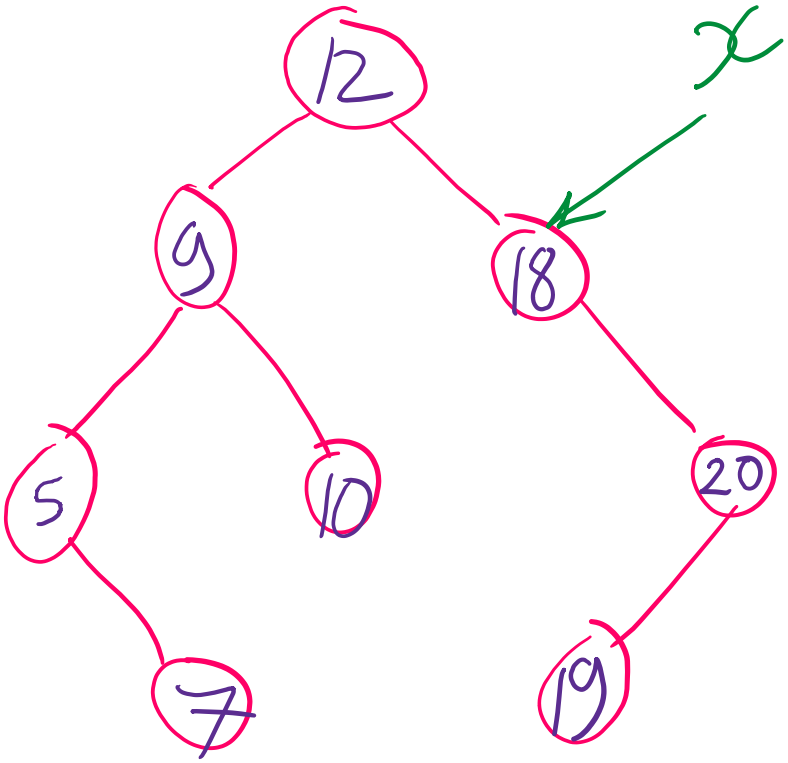
BST search using iteration




```
x = root
while(x != null){
    if equal break;
    if < x = x.left
    if > x = x.right
}
```

Search for 19

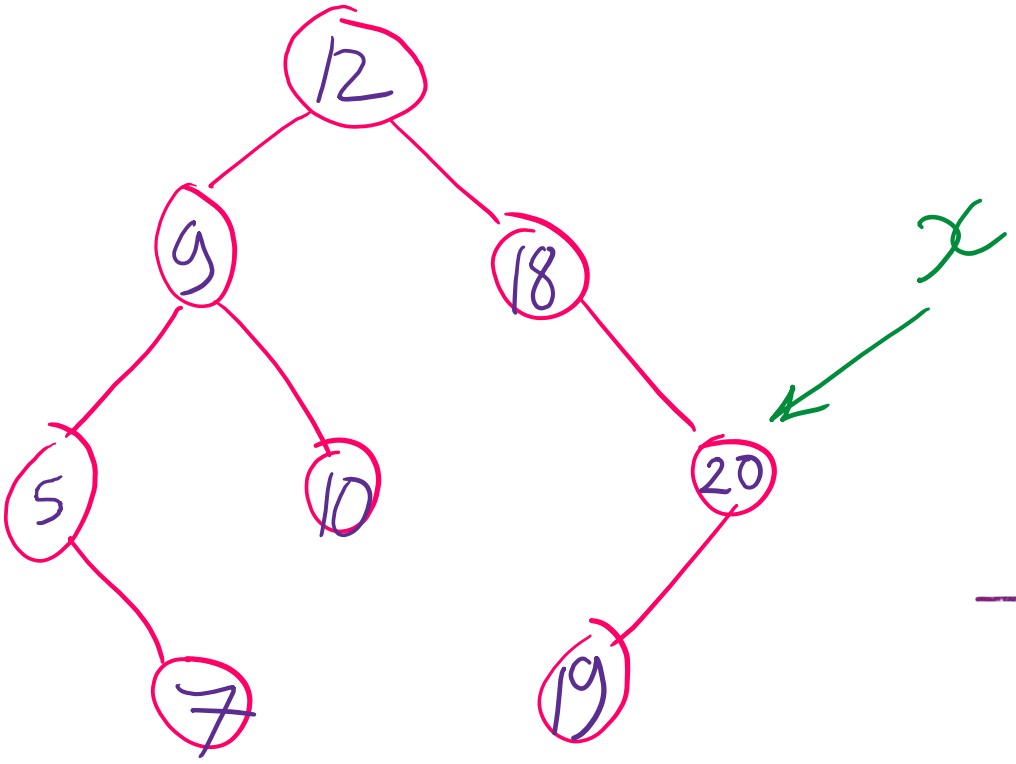
BST search using iteration




```
x = root
while(x != null){
    if equal break;
    if < x = x.left
    if > x = x.right
}
```

Search for 19

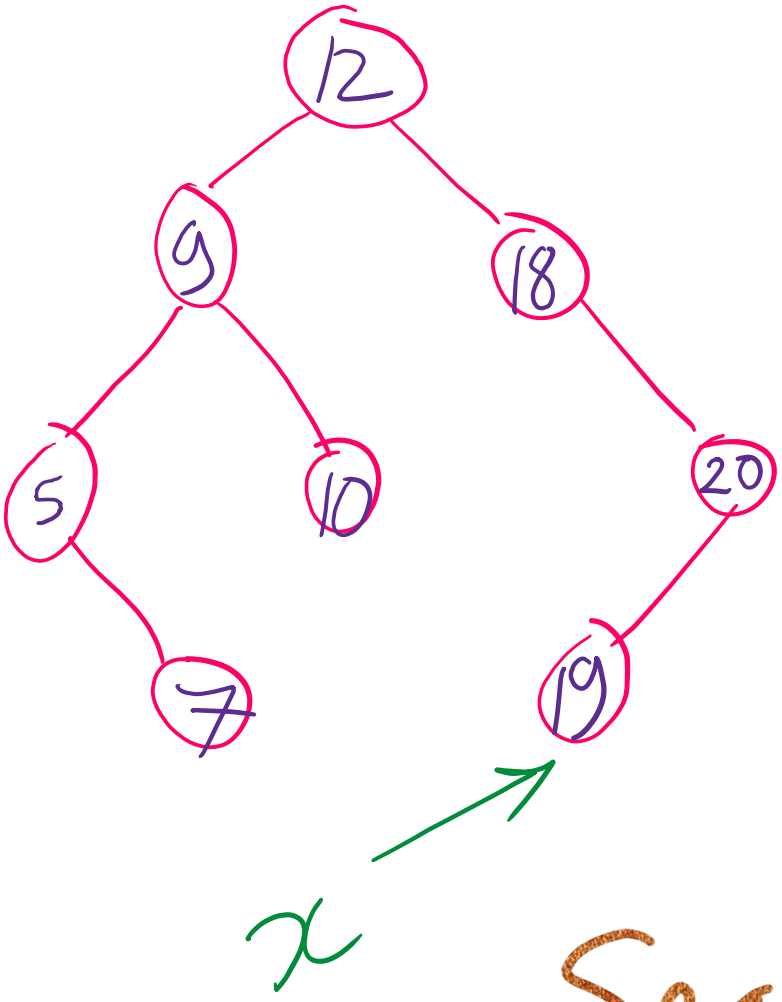
BST search using iteration




```
x = root
while(x != null){
    if equal break;
    if < x = x.left
    if > x = x.right
}
```

Search for 19

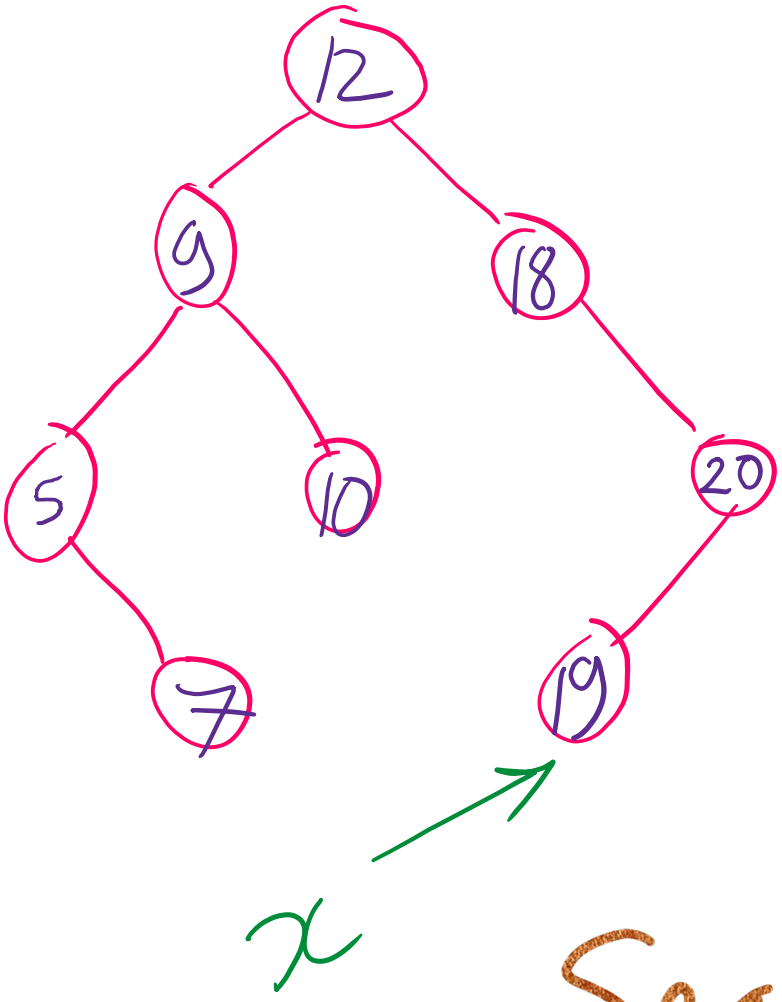
BST search using iteration




```
x = root
while(x != null){
    if equal break;
    if < x = x.left
    if > x = x.right
}
```

Search for 19

BST search using iteration




```
x = root
while(x != null){
    if equal break;
    if < x = x.left
    if > x = x.right
}
```

Search for 19

Muddiest Points

- **Q: The iterative method for pre-order was confusing for me to grasp but it started to make more sense once we did in-order and post-order.**
- Thanks!

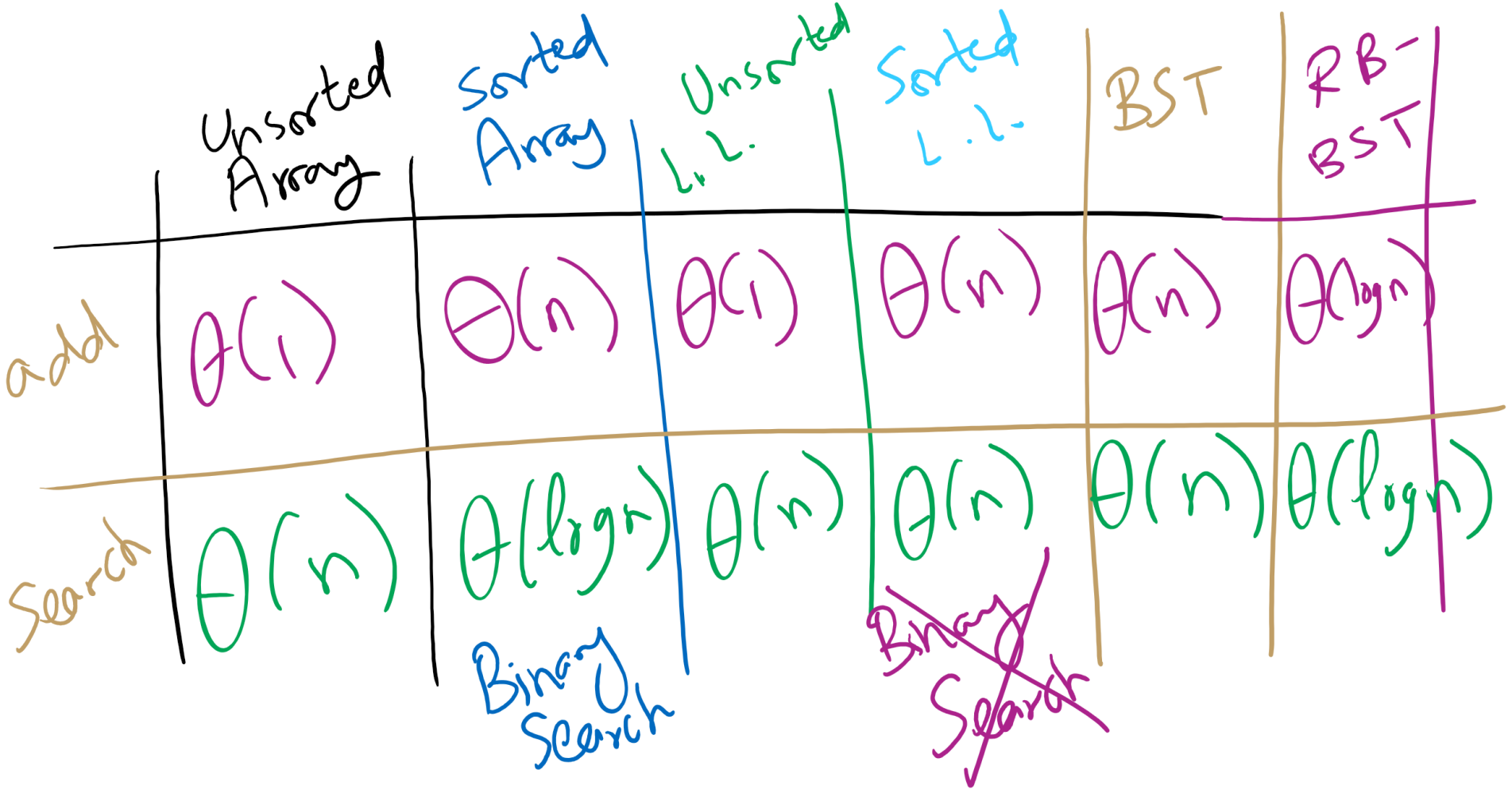
Muddiest Points

- **Q: How does a Red Black BST keep track of the data in the nodes if it's just comparing colors?**
- It is comparing data as we go down the tree (except for delete) and checking colors as we climb back up the tree

Muddiest Points

- **Q: can you explain the stack and what pushing/popping are?**
- Stack is an abstract data type in which data items are ordered in a Last In First Out order.
- Pushing into a Stack means to add an item at the “top” of the stack
- Popping means to remove the top item

Symbol Table Implementations



This Lecture

- Binary Search Tree uses comparisons between keys to guide the searching
- What if we use the digital representation of keys for searching instead?
  - Keys are represented as a sequence of digits (e.g., bits) or alphabetic characters
- Digital Searching Problem

Digital Searching Problem

- Input:
  - a (large) dynamic set of data items in the form of
    - *n* (key, value) pairs; key is a string from an alphabet of size *R*
    - Each key has *b* bits or *w* characters (the chars are from the alphabet)
    - What is the relationship between *b* and *w*?
  - a *target key (k)*
- Output:
  - The corresponding value to *k* if target key found
  - Key not found otherwise

Digital Search Trees (DSTs)

Instead of looking at less than/greater than, lets go left or right based on the bits of the key

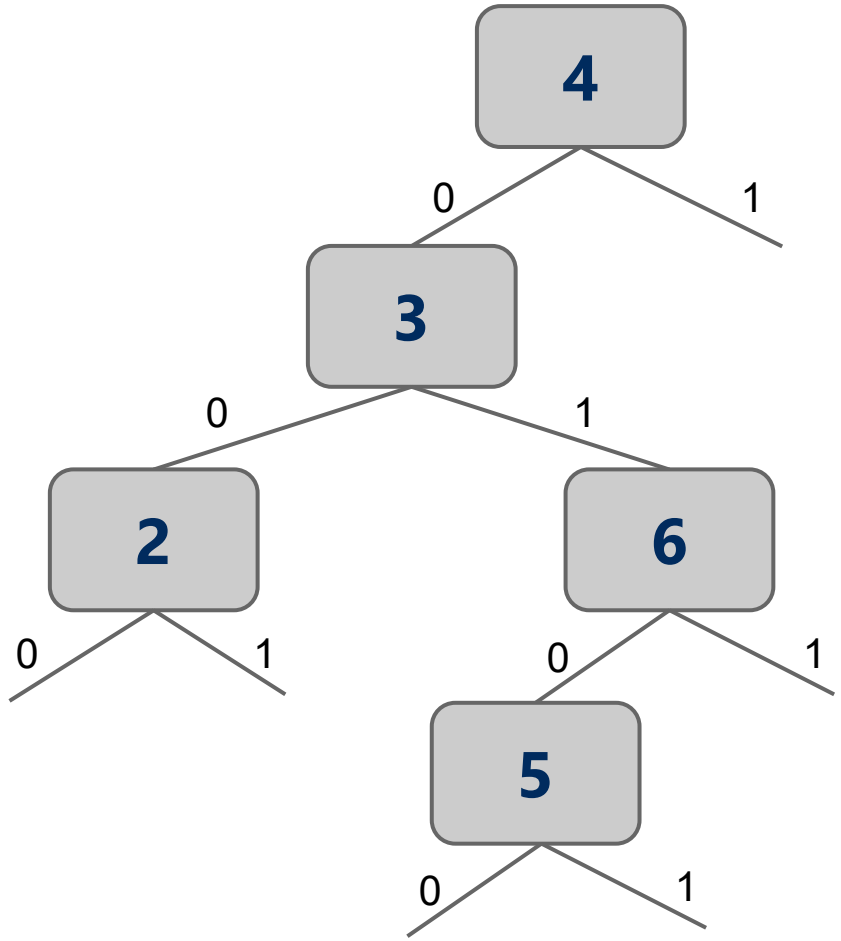
So, we again have 4 options:

- current node is null, k not found
- k is equal to the current node's key, k is found, return corresponding value
- current bit of k is 0, continue to left child
- current bit of k is 1, continue to right child

DST example: Insert and Search

Insert:

4	0100
3	0011
2	0010
6	0110
5	0101

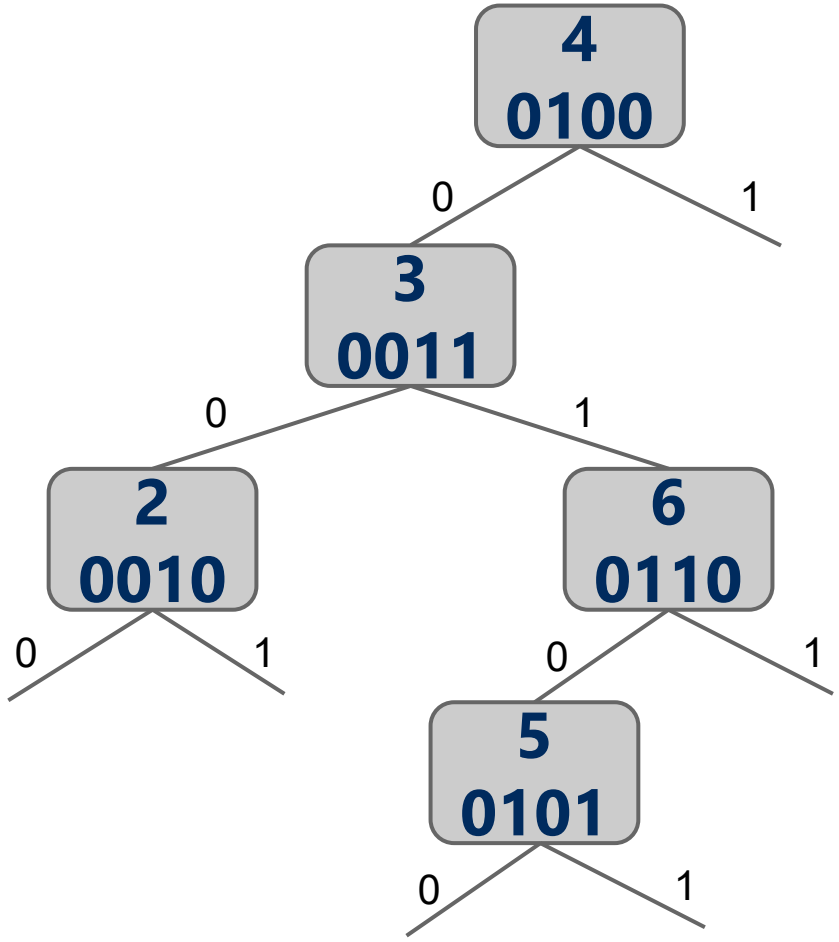


Search:

3	0011
7	0111

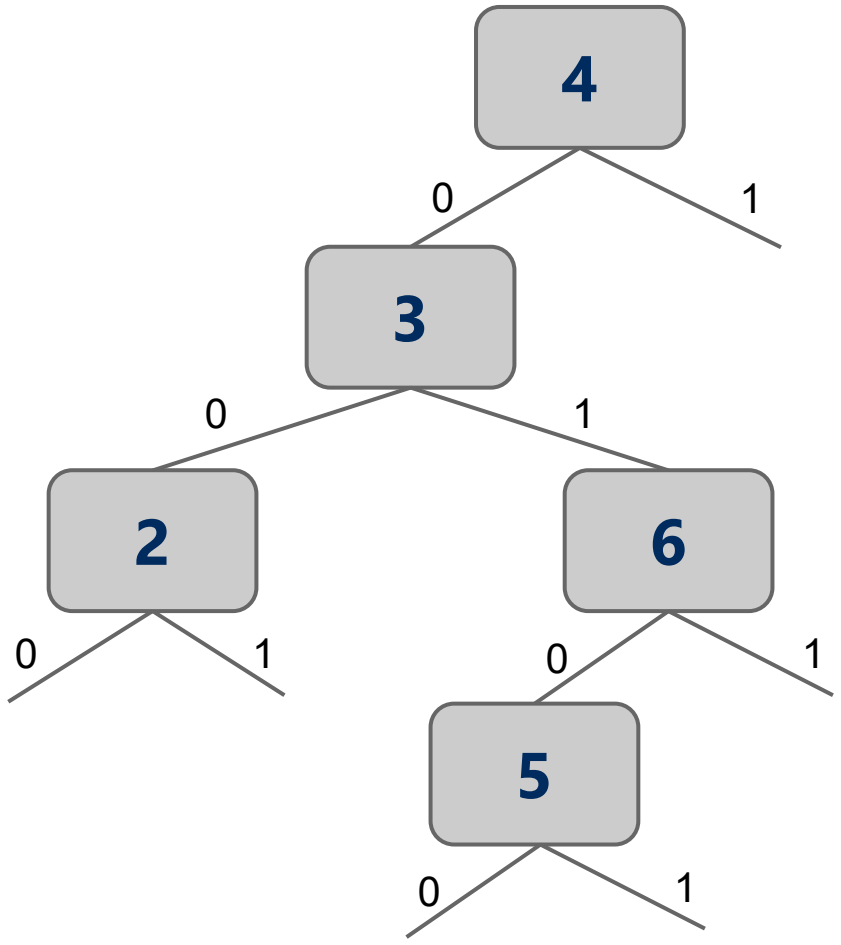
DST and Prefixes

- In a DST, each node shares a **common prefix** with all nodes in its subtree
  - E.g., 6 shares the prefix "01" with 5
- In-order traversal doesn't produce a sorted order of the items
  - Insertion algorithm can be modified to make a DST a BST at the same time



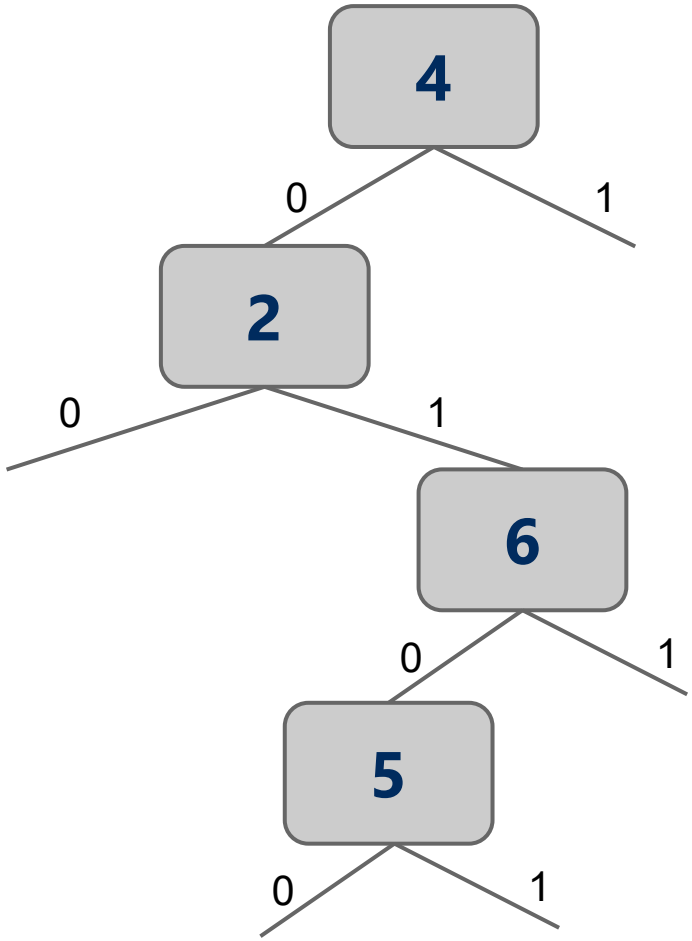
DST example: Delete

- Delete 3
- Can replace it with any leaf in its subtree
- Let's replace it with 2
- OK because 2 shares "0" as a prefix with 3, so it also shares "0" as a prefix with 6 and 5



DST example: Delete

- Delete 3
- Can replace it with any leaf in its subtree
- Let's replace it with 2
- OK because 2 shares "0" as a prefix with 3, so it also shares "0" as a prefix with 6 and 5

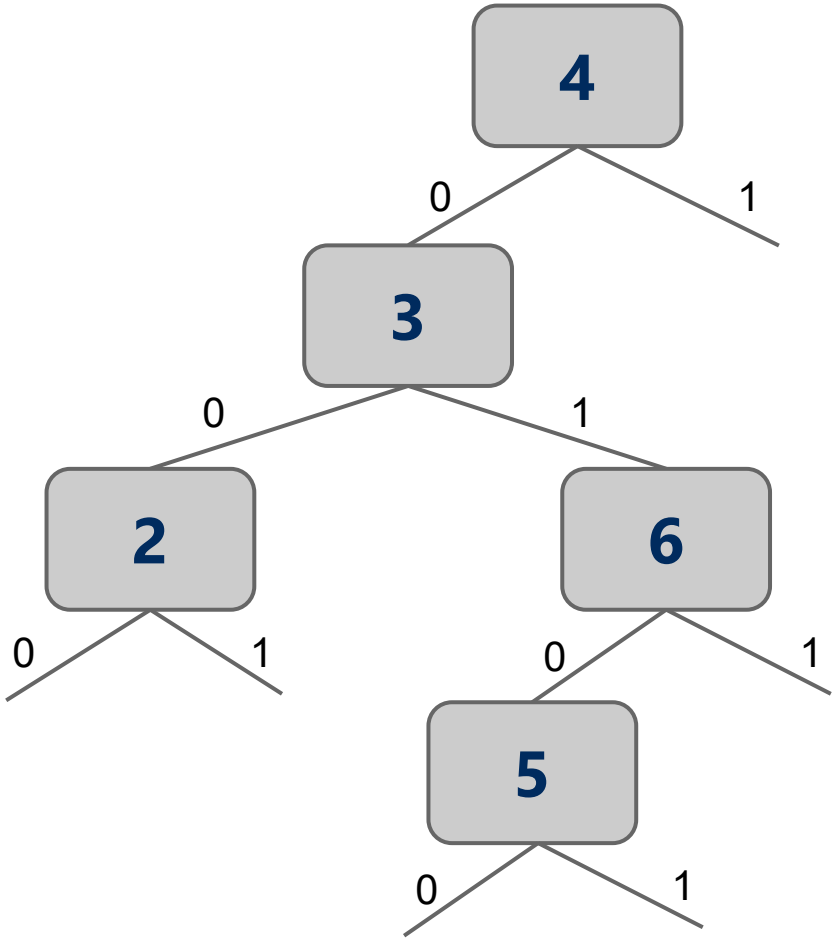


DST example: Variable length keys

- Insert

1 01

- Must be in place of 6
- Replace 6 by 1 and re-insert 6



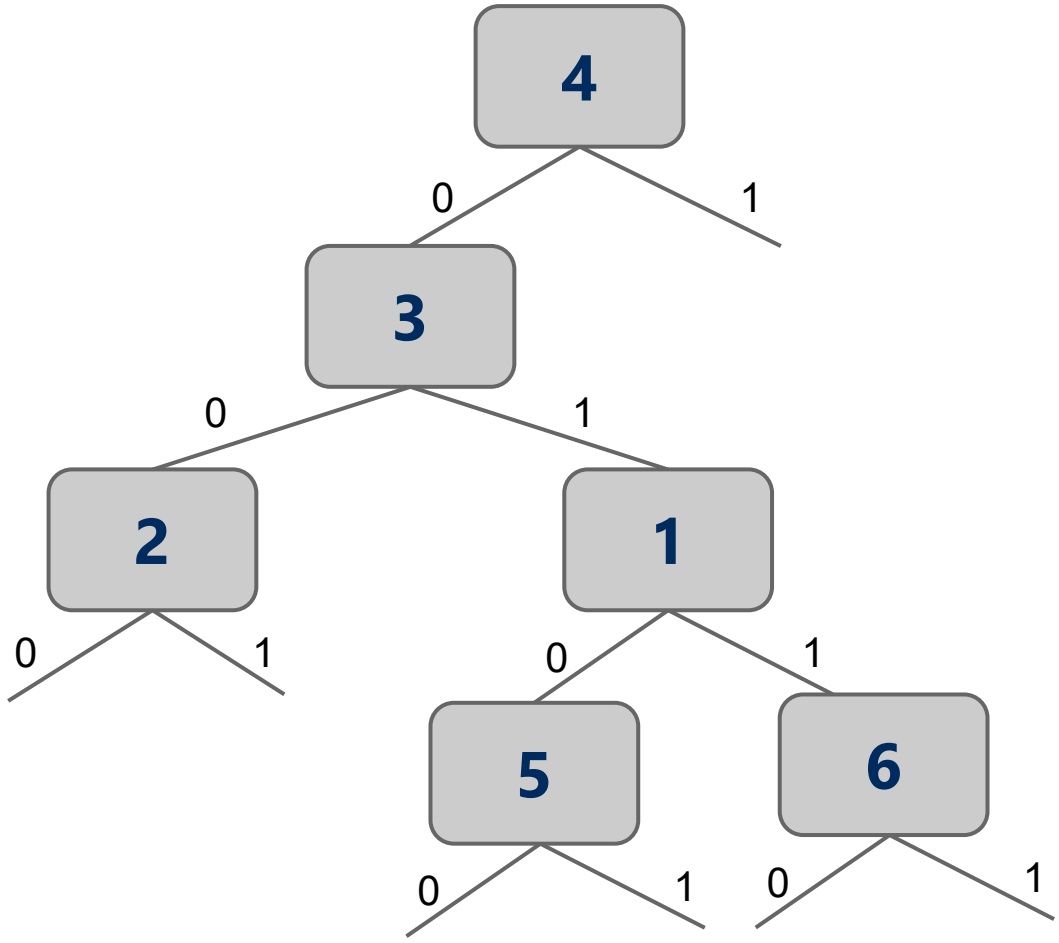
DST example: Variable length keys

- Insert

1 01

- Must be in place of 6
- Replace 6 by 1 and re-insert

6 0110



Analysis of digital search trees

$$\text{average case runtime} = \sum_{\text{all cases}} Pr(\text{case}_i) * \text{runtime for case}_i$$

- Runtime?
  - O(b), b is the bit length of the target or inserted key
  - On average, b = log(n)
  - When branching according to a 0 or 1 is equally likely
  - In general b >= $\lceil \log n \rceil$
- We end up doing many **<u>equality</u>** comparisons against the full key
- This is better than less than/greater than comparison in BST
- Can we improve on this?

Radix search tries (RSTs)

- Trie as in retrieve, pronounced the same as "try"
- Instead of storing keys inside nodes in the tree, we store them implicitly as paths down the tree
  - Interior nodes of the tree only serve to direct us according to the bitstring of the key
  - Values can then be stored at the end of key's bitstring path (i.e., at leaves)
  - RST uses less space than BST and DST

RST example

Insert:

4	0100
3	0011
2	0010
6	0110
5	0101

Search:

3	0011
7	0111