Sp18 CS 61B Discussion 10

Welcome!

Wayne Li

wli2@berkeley.edu

https://wayne-li2.github.io/

Announcements

- Done with MT2!
- We'll be grading through the week (hopefully not into Spring Break)
- No quiz
- Not doing worksheet but take one for scratch paper

Quiz Instructions

- If you haven’t yet, please also **neatly** put your email address **outside the name box** if you want to be emailed!
- Bubble number **41**.

Aside

k-Nearest Neighbors (CS 189)

- Simple yet classic problem
- Given a point (x, y) and a dataset of (x_i, y_i):
  - Find the k-nearest points (Euclidean distance) to the input point.
- Come up with an naive algorithm!
  - Hint: Use a data structure we recently learned!

k-Nearest Neighbors (CS 189)

- Use a **max-heap** (not a min-heap!) of size **k** to maintain nearest neighbors.
- **peek()** returns the kth nearest element found so far
- For each value **v**, if **v** is closer than **peek()**, then **pop()**.
    - Then, add **v** to the heap.

k-Nearest Neighbors (CS 189)

- What’s the runtime?

k-Nearest Neighbors (CS 189)

- What's the runtime?
  - $O(nd + n \log k)$
- What's **d**?

k-Nearest Neighbors (CS 189)

- What's the runtime?
  - O(nd + n log k)
- What's **d**?
  - **d** is the dimensions of your point.
  - If our points are in 3 dimensions (x, y, z), then calculating the distance will take longer.

k-Nearest Neighbors (CS 189)

- At **d ~ 30**, this algorithm becomes kind of slow (practical terms).
  - Your first experience with big data, and how linear runtimes are no longer good enough.
  - Can we get a sublinear query time with respect to n?
    - Remember, in big data, **n >>>> k**

k-Nearest Neighbors (CS 189)

- What about a quadtree for k dimensions?

k-Nearest Neighbors (CS 189)

- What about a quadtree for k dimensions?
  - Quickly infeasible.
  - 2D -> 4 children.
  - 3D -> 8 children.
  - 4D -> 16 children.
  - **That's a lot of hardcoded children**

k-D Tree (CS 189)

- Use a k-D Tree!
  - Exactly like a binary search tree!
  - Except each layer is a separate decision.
- Note: k-NN is just one application of the k-D tree
- Note 2: The “k” in k-NN means something different than the “k” in “k-D tree”.

k-D Tree (CS 189)

- Example: 2-D Tree
- Root ----------------->   (**8**, 8)   Compare x values.
- Less than x   /   \   Greater than x
- Children -----------> (4, **10**) (12, **2**) Compare y values.
- Less than y   /   \ /   \   Greater than y
- ...and repeat

k-D Tree (CS 189)

- Result: Partition a k-D space.
- What’s the runtime to build a k-D tree?

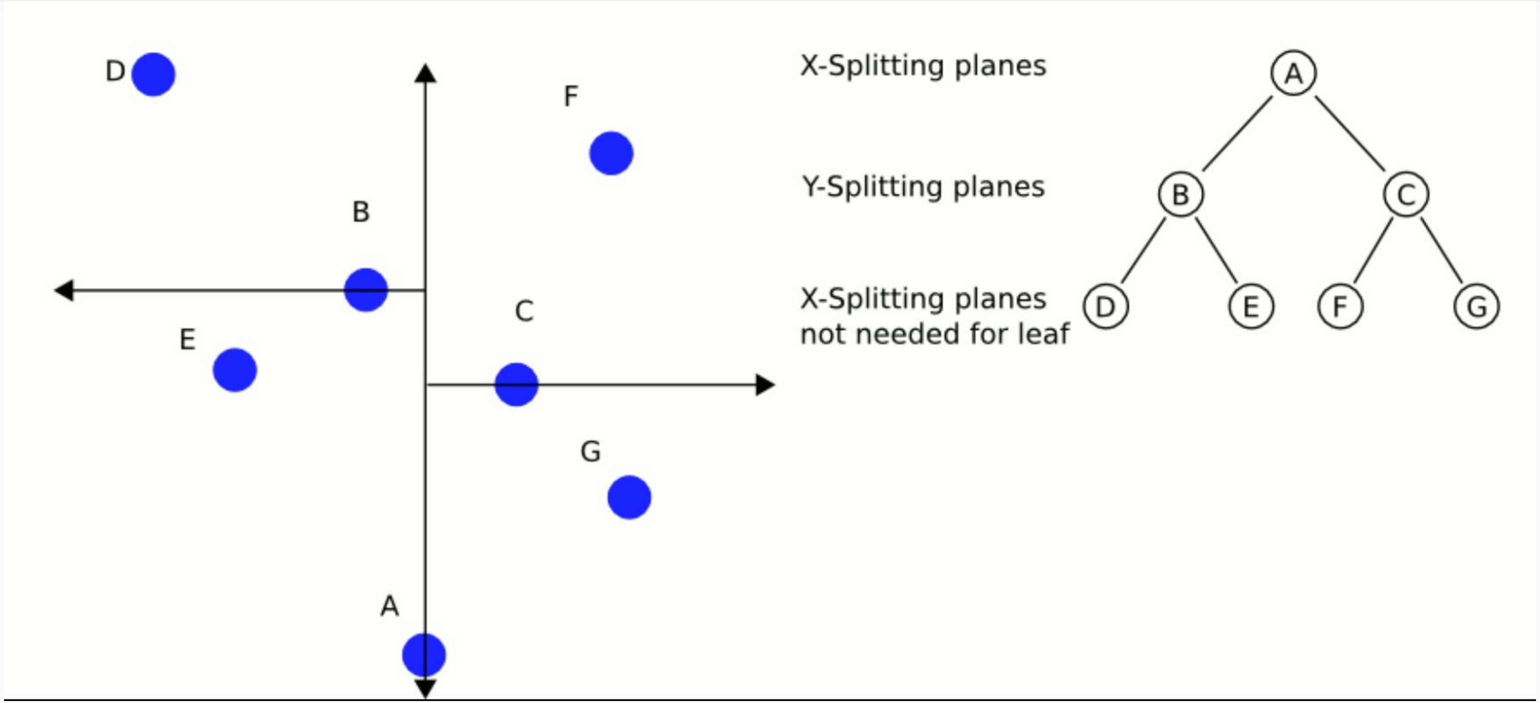
k-D Tree (CS 189)

- Result: Partition a k-D space.
- What's the runtime to build a k-D tree with **n** points?
    - O(n log n)

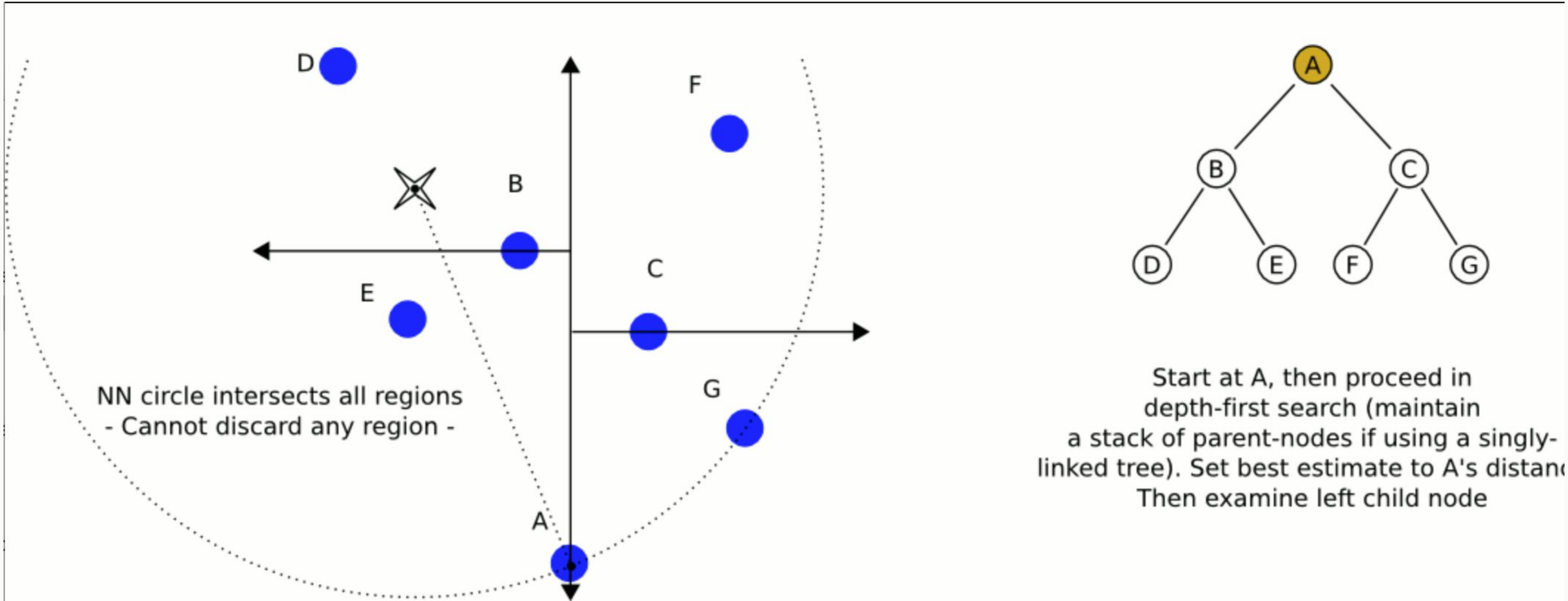
k-D Tree (CS 189)

- Is O(n log n) to build a good runtime?
  - Yes! Hopefully, you only build the tree once, and query it a billion times.
  - If your query is sublinear, then overall it is a good tradeoff.

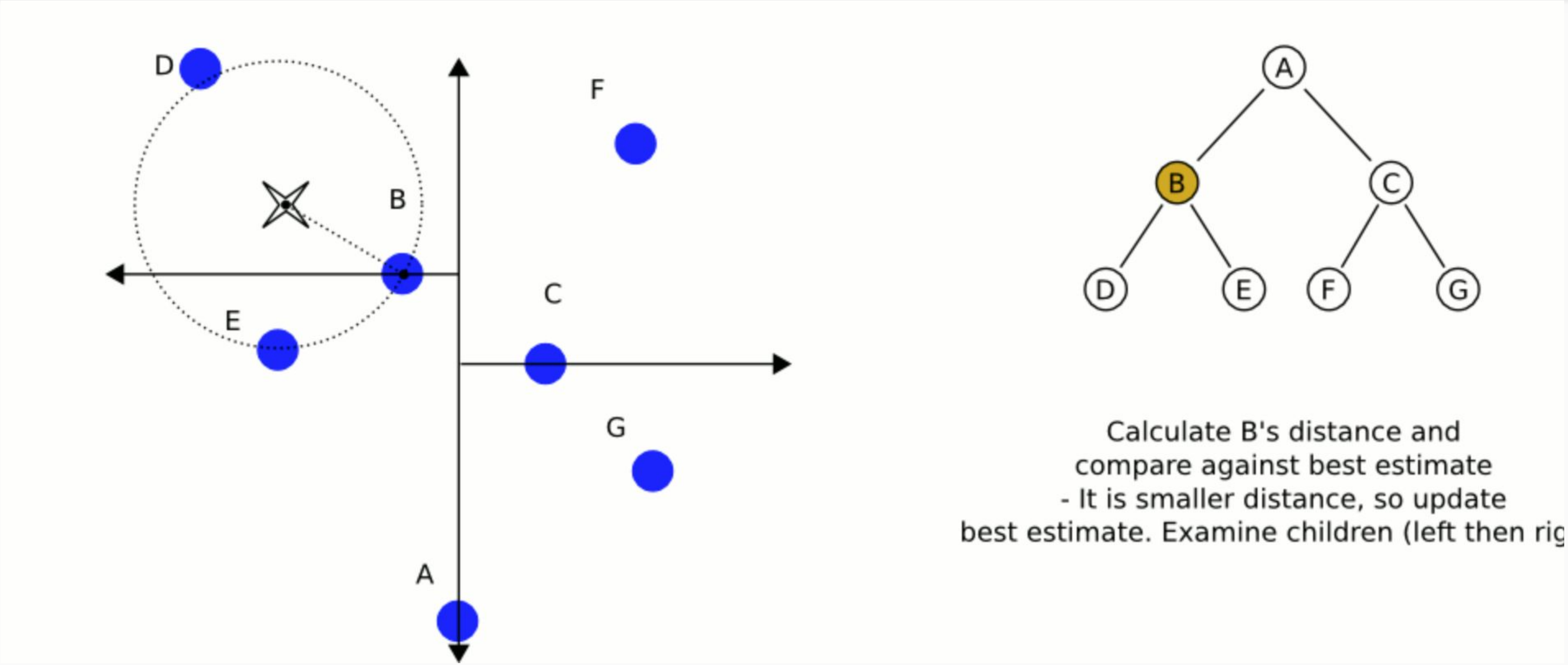
k-D Tree (CS 189)



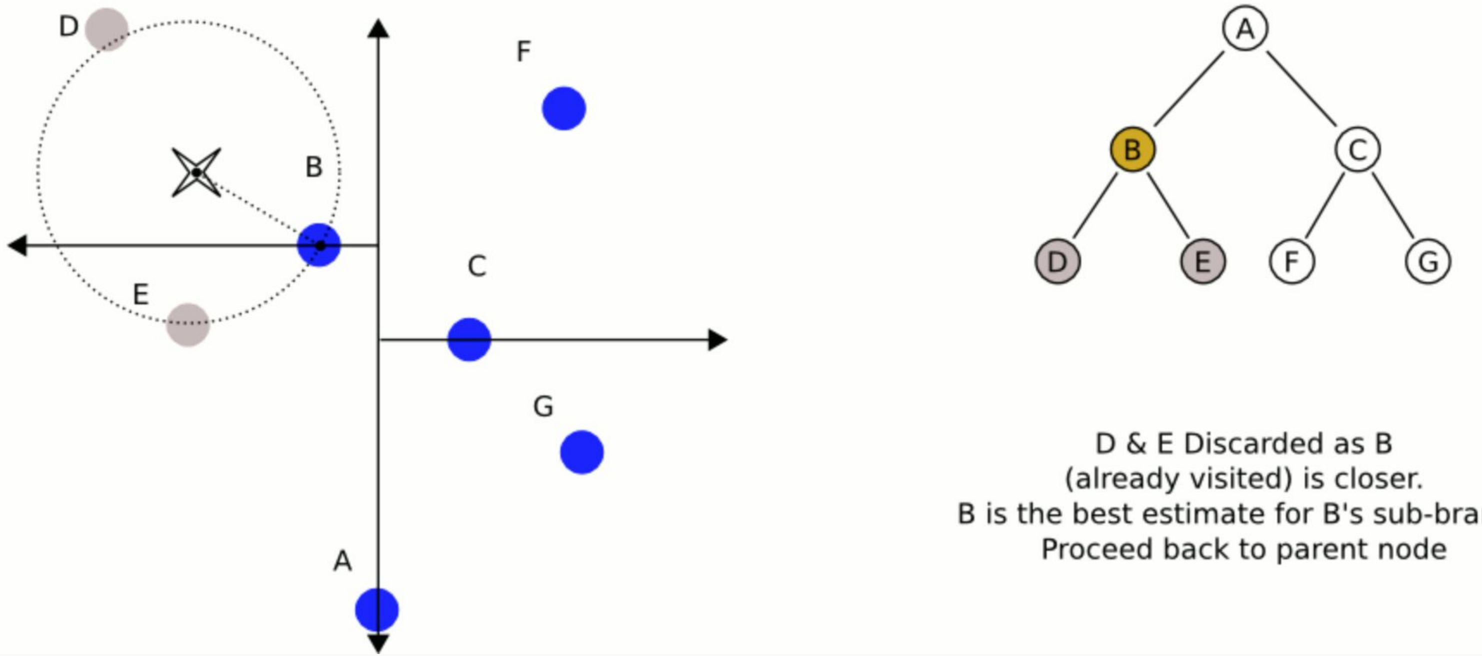
k-D Tree (CS 189)



k-D Tree (CS 189)

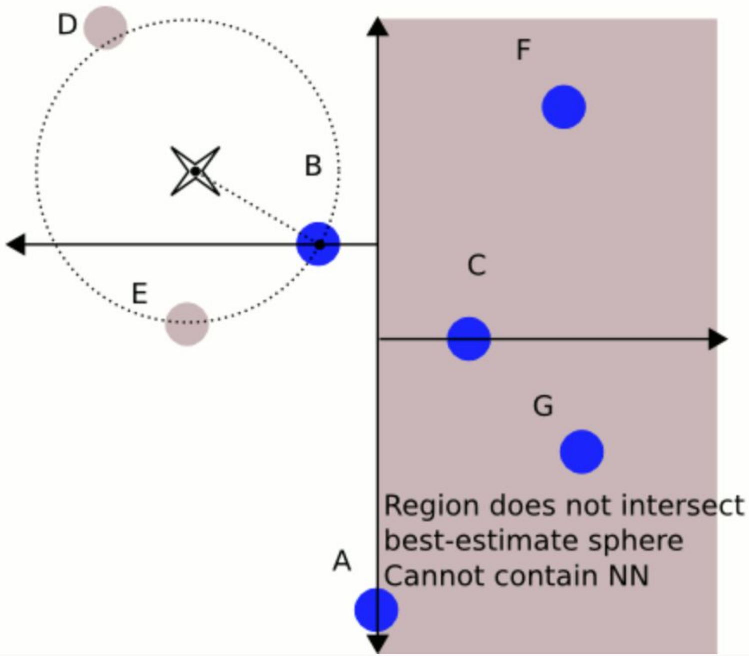


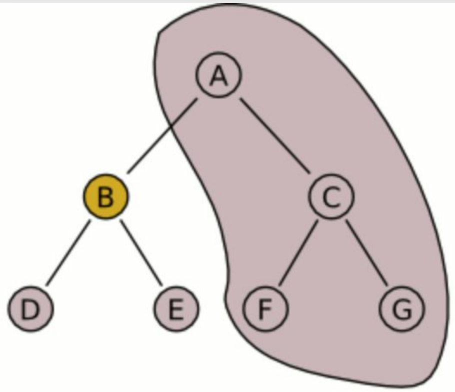
k-D Tree (CS 189)



D & E Discarded as B
(already visited) is closer.
B is the best estimate for B's sub-branch
Proceed back to parent node

k-D Tree (CS 189)





A's children have all been searched,
B is the best estimate for entire tree

k-D Tree and k-NN (CS 189)

- Can we modify this algorithm for k-nearest neighbors?
  - Hint: Same idea as before...

k-D Tree and k-NN (CS 189)

- Can we modify this algorithm for k-nearest neighbors?
  - Hint: Same idea as before…
- Use a max-heap!

k-D Tree and k-NN (CS 189)

- Runtime of k-NN with k-D tree?
  - We know it is O(n log n) to build tree.
  - What about query time?

k-D Tree and k-NN (CS 189)

- At worst, check every single node.
- ~O(nd + n log k)
  - Actually O(dn^(1 - 1/d) + n^(1 - 1/d) log k)
    - **As d increases, runtime -> O(nd + n log k)**
- How to fix?

Approximate k-NN search (CS 170)

- Use an **approximation algorithm**!
  - In this case, shrink the “circle” by **epsilon** length.
  - Thus, search less branches.
- In practice, 10x to 100x speed up.

Advanced Topics

- Random projections! (CS 174)
  - Say d = 1000. Infeasible even for k-D tree.
  - Instead, randomly select 100 dimensions!
    - You'll still be pretty accurate (proof is **hard**)
- … so much more

Moral of the Story

- CS 189 is probably the coolest class at Berkeley!
  - Prof. Shewchuk is probably the best professor I've ever had as well.
- Mathematics is an important side-skill to CS!

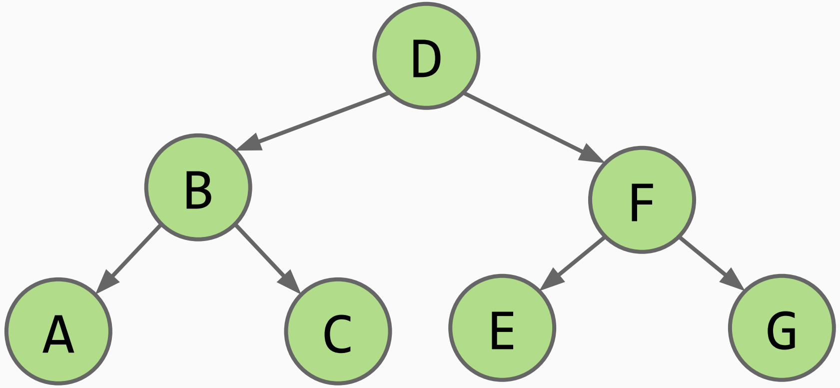
References

- Prof. Shewchuk's notes on k-NN:
    - https://people.eecs.berkeley.edu/~jrs/189/lec/24.pdf
    - https://people.eecs.berkeley.edu/~jrs/189/lec/25.pdf
- https://en.wikipedia.org/wiki/K-d_tree

Tree Traversals

Tree Traversals

- Depth First
  - Preorder (DBACFEG)
  - Inorder (ABCDEFG)
  - Postorder (ACBEGFD)



Recursive Preorder

- preOrder(Node n):
  - Process(n)
  - preOrder(n.left)
  - preOrder(n.right)

Do: Recursive Inorder/Postorder

Recursive Inorder

- inOrder(Node n):
    - inOrder(n.left)
    - Process(n)
    - inOrder(n.right)

Recursive Postorder

- postOrder(Node n):
  - postOrder(n.left)
  - postOrder(n.right)
  - Process(n)

Do: Iterative Preorder

Iterative Preorder

- Hint: Use a data structure!

Iterative Preorder

- Hint: Use a data structure!
  - Stack!

Iterative Preorder

- iterativePreOrder(Tree t):
  - Create stack **S**, add **t.root**
  - while **S** not empty:
    - **n = S.pop()**, then **process(n)**
    - **S.push(n.right)**, then **S.push(n.left)**

Iterative Preorder

- Why do we push the right one first?

Iterative Preorder

- Why do we push the right one first?
    - So on the stack, we pop left, then right.

Do: Iterative Inorder (Hard!)

Iterative Inorder

- Hint: Use a data structure!

Iterative Inorder

- Hint: Use a data structure!
- Hint 2: It’s the same data structure as before.

Iterative Inorder

- Hint: Use a data structure!
- Hint 2: It’s the same data structure as before.
- Hint 3: You need an additional pointer.

Iterative Inorder

- iterativeInOrder(Tree t):
    - Create stack **S**, leave it **empty**
    - Set pointer **current = t.root**

Iterative Inorder

- iterativeInOrder(Tree t):
  - Create stack **S**, leave it **empty**
  - Set pointer **current = t.root**
  - while (current != null): // **findLeftmostNode(current)**
    - **S.push(current)**
    - **current = current.left**

Iterative Inorder

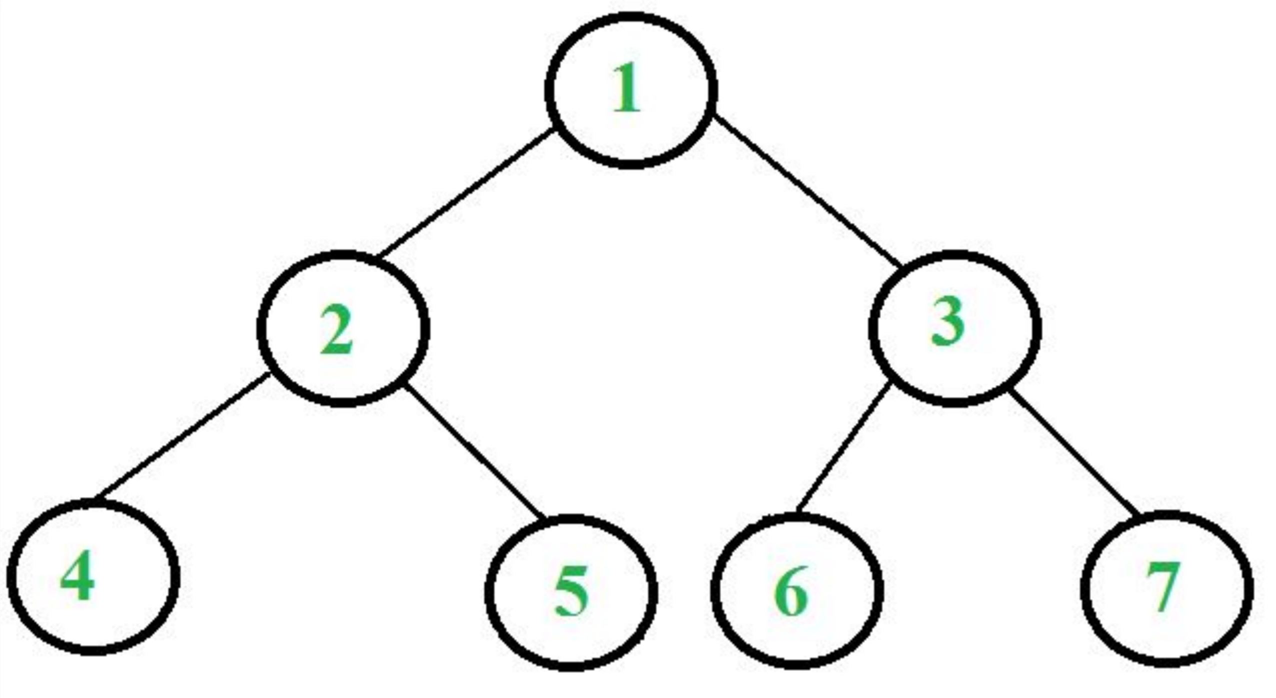
- iterativeInOrder(Tree t):
  - …
  - while **S** not empty:
    - **n = S.pop()**, then **process(n)**
    - if **n.right** exists:
      - **n = n.right**, then **findLeftmostNode(n)**

Do: Iterative Postorder (Really Hard!)

Iterative Postorder

- Hint: Using one stack is possible, but super hard

Iterative Postorder: Find it!



Iterative Postorder

- Answer: 4, 5, 2, 6, 7, 3, 1

Iterative Postorder

- Answer: 4, 5, 2, 6, 7, 3, 1
- **Hint: Reverse this order:**
  - **1, 3, 7, 6, 2, 5, 4**
  - What does this **sorta** look like to you?

Iterative Postorder

- It’s a preorder traversal that goes **root, right, left!**
  - Instead of the usual **root, left, right.**
- With this in mind, what are some ways we can get the **reverse** of a preorder traversal?
  - Hint: Use a data structure.

Iterative Postorder

- It's a preorder traversal that goes **root, right, left!**
  - Instead of the usual **root, left, right.**
- With this in mind, what are some ways we can get the **reverse** of a preorder traversal?
  - Hint: Use a data structure.
    - Use a stack!

Iterative Postorder

- iterativePostOrder(Tree t):
  - Create stack **S**, add **t.root**
  - Create result stack **R**
  - while **S** not empty:
    - **n = S.pop()**, then **R.push(n)**
    - **S.push(n.left)**, then **S.push(n.right)** // Swapped!

Iterative Postorder

- iterativePostOrder(Tree t):
  - ...
  - while **S** not empty:
    - **n = S.pop()**, then **R.push(n)**
    - **S.push(n.left)**, then **S.push(n.right)** // Swapped!
  - while **R** not empty: **process(R.pop())**