Graph Algorithm

(Chapter 4.2, 5.3)

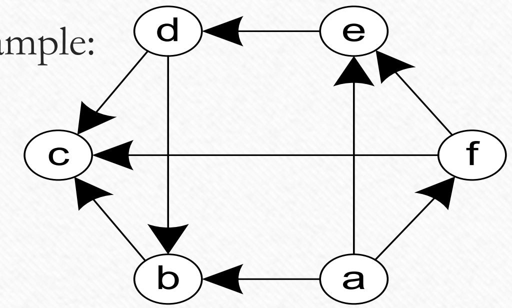
Graph Algorithm

- Topological Sorting
  - Using DFS
  - Decrease by one

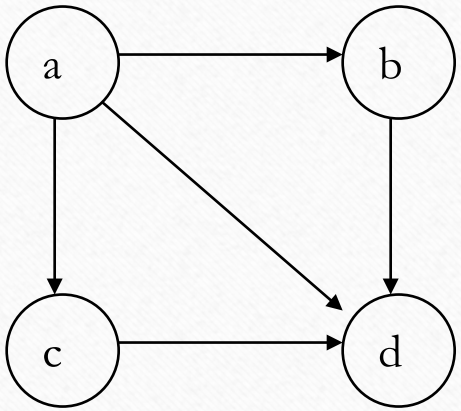
- Binary Tree Traversal
  - Preorder
  - Inorder
  - Postorder

Directed Acyclic Graphs (DAG's)

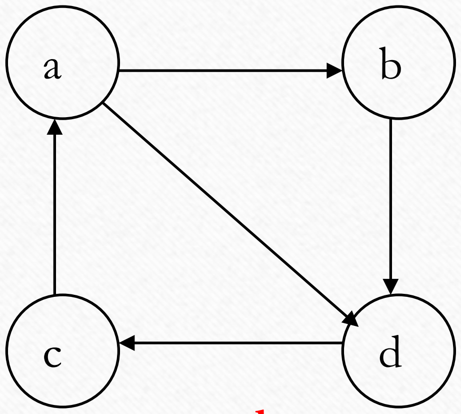
- recall that a **directed graph** is a graph that uses arrows to show direction. For example:



- a **directed acyclic graph**, aka **DAG**, is a directed graph that contains no cycles



a dag



not a dag

Topological Sort

- **Problem:** We have a set of tasks and a set of dependencies (precedence constraints) of form "task A must be done before task B"

- **Goal:** Find a linear ordering that satisfies all dependencies

Topological Sort

Input might look like this ....

- a must be done before b, e, f
- b must be done before c
- d must be done before b and c
- e must be done before d
- f must be done before c and e

one possible solution (topologically sorted order):

- a f e d b c

Topological Sort Algorithm 1: DFS

To obtain a topological sort order for a set of items:

1. represent the items as a directed graph G such that:
   a) vertices are the items that are tasks
   b) edges are the dependencies (constraints) between tasks
      - an edge from v to w (eg: v→w) means that v is dependent on w ... ie ... v must be done before w
2. apply the DFS algorithm to G
3. the order in which vertices become dead ends gives the reverse topological sort order

Note: Topological Sort produces no solution if the graph contains a cycle

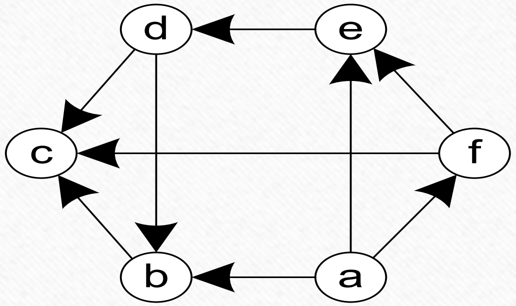
Topological Sort Algorithm (DFS)

Recall:

- the DFS implementation is recursive
- each time a recursive call is made is equivalent to "pushing a vertex on a stack"
- the “order in which vertices become dead ends” is given by the “order in which vertices are popped off the stack”

Example 1: Work Tasks

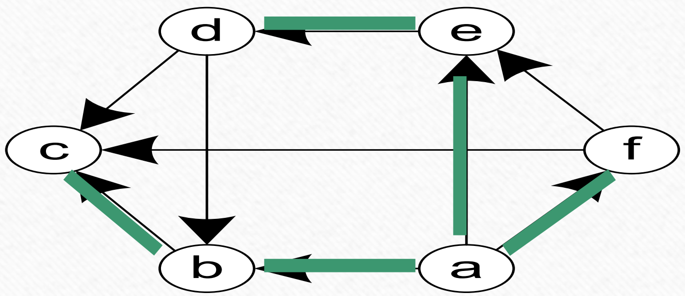
- Assume you have a set of 6 tasks (a, b, c, d, e, f) with the following dependencies:
  - a must be done before b, e, f
  - b must be done before c
  - d must be done before b and c
  - e must be done before d
  - f must be done before c and e



- <u>Step 1:</u> Draw a directed graph to represent these dependencies.

Example 1 (cont)

- Step 2: Apply DFS



Order vertices become dead ends:

c b d e f a

- Step 3:

Reverse this order for the solution:

a f e d b c

Example 2: Work Tasks

2 1 ← (2 before 1) 4 3 ← (4 before 3)

1 4 ← (1 before 4) 5 2 ← (5 before 2)

2 3 ← (2 before 3) 5 1 ← (5 before 1)

5 6 ← (5 before 6) 6 3 ← (6 before 3)

2 4 ← (2 before 4) 6 2 ← (6 before 2)

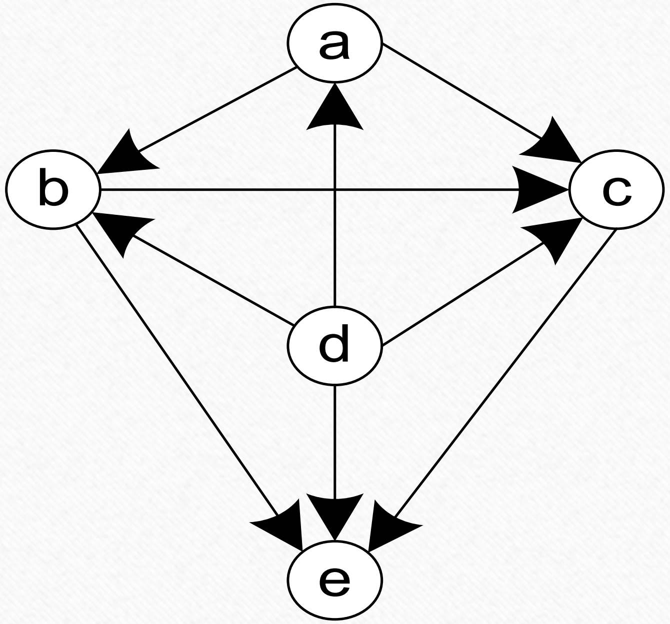
Try at Home

- Step 1: draw the graph (and verify it is a DAG)
- Step 2: apply DFS
- Step 3: find the order vertices were removed from stack, and reverse this order to get topological sort order

Topo Sort Algo 2: Decrease by One

- Observe:
  - if a vertex v in the dependency graph G has no incoming arrows (ie: in-degree(v) == 0), then v does not have any dependencies
  - it follows that any v that does not have dependencies is a candidate to be visited next in topographical order
- A Decrease-by-One approach:
  - identify a $v \in V$ that has in-degree = 0
  - delete v and all of its edges
  - when all vertices have been deleted, the topo sort order is given by the order of deletion
  - if there are $v \in V$, but no v has in-degree = 0, the graph G is not a DAG (no feasible solution exists)

Example



Whiteboard

Topo Sort Algo 2: Decrease by One

- More detailed algorithm:
  - need a set to store the candidate v's (in-degree = 0)
    - I will use a TreeSet. Any ordered set will do.
  - need an ordered list to store the delete order
    - I will use an ArrayList. Any ordered list will do.

- Then the algorithm is:

Topo Sort Algo 2: Decrease by One

Whiteboard

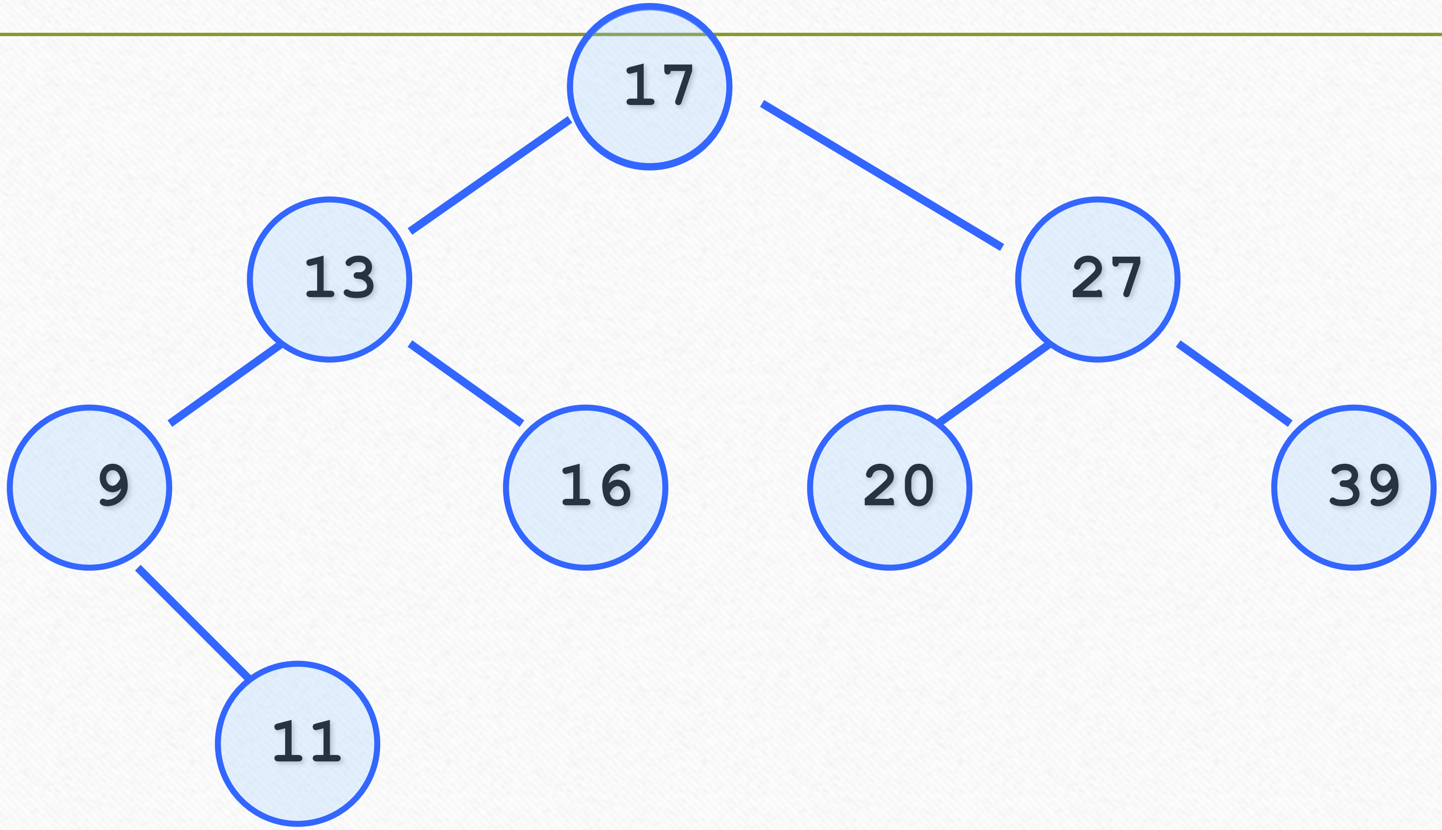
```
topo(G)
   create an empty ArrayList Soln
   create an empty TreeSet Candidates
   add all v with inDegree=0 to Candidates
   while Candidates is not empty
      v ← Candidates.first()
      add v to Soln
      for each vertex w adjacent to v
         remove edge (v,w) from G
         if w has inDegree=0
            add w to Candidates
      remove vertex v from G
   if there are vertices remaining in G
      no feasible solution exists
   else
      solution is in Soln
```

Graph Algorithm

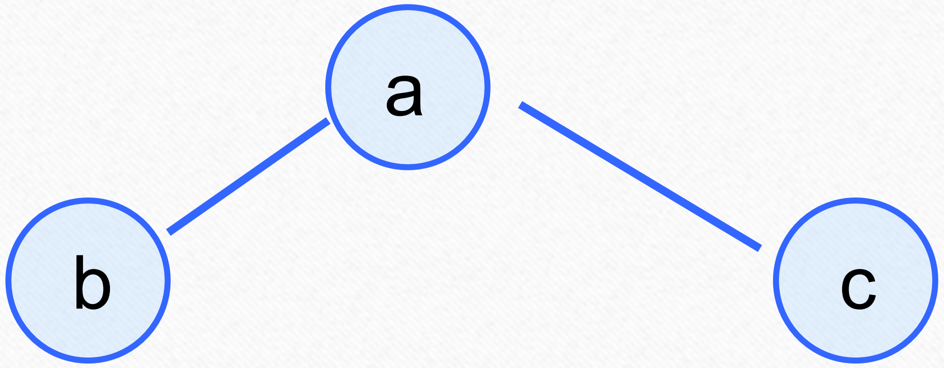
- Topological Sorting
  - Using DFS
  - Decrease by one

- Binary Tree Traversal
  - Preorder
  - Inorder
  - Postorder

Binary Tree

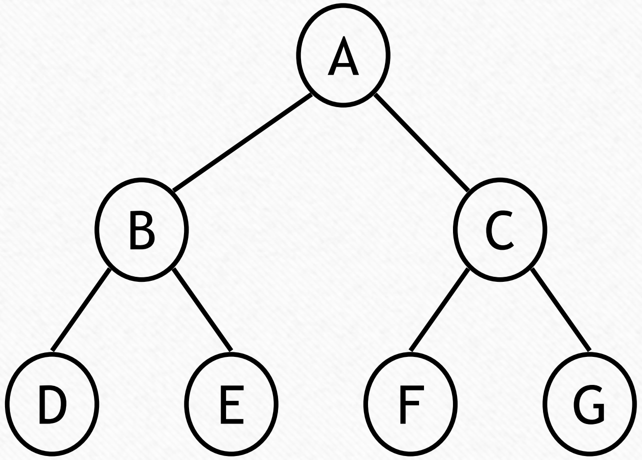


Preorder



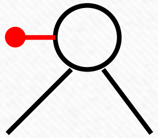
a b c

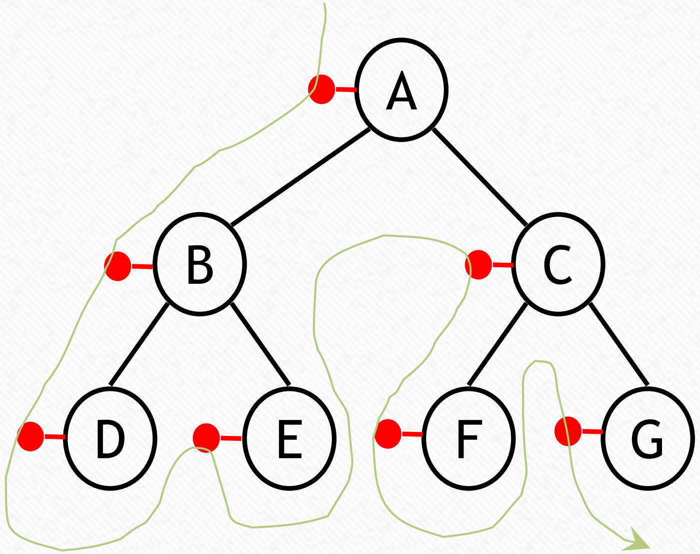
Example



Preorder: A B D E C F G

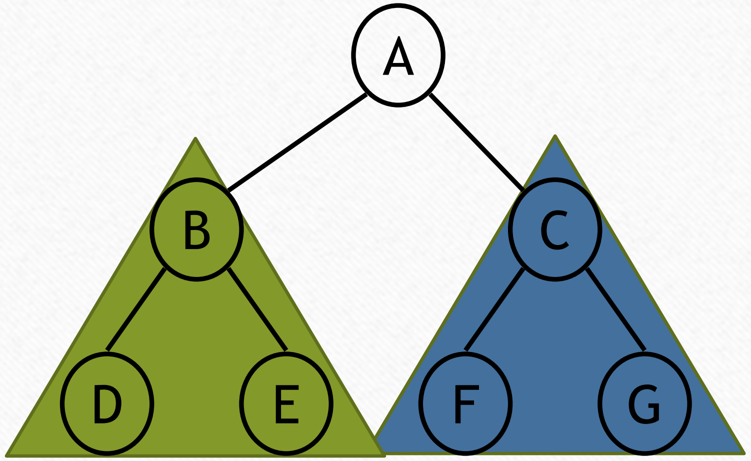
Example





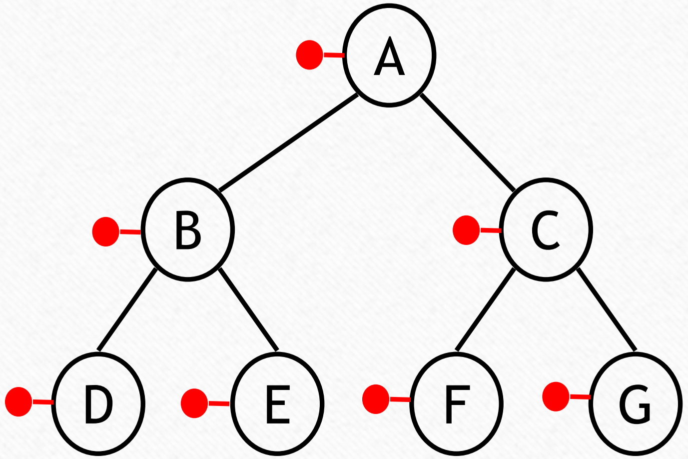
Preorder: A B D E C F G

Example



Preorder: A B D E C F G

Preorder

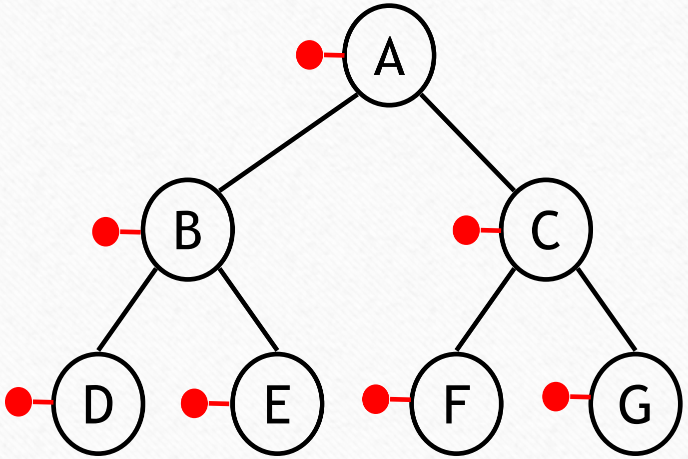


Preorder: A B D E C F G

```
public void preorderPrint(Node N) {

}
```

Create
Pseudo Code

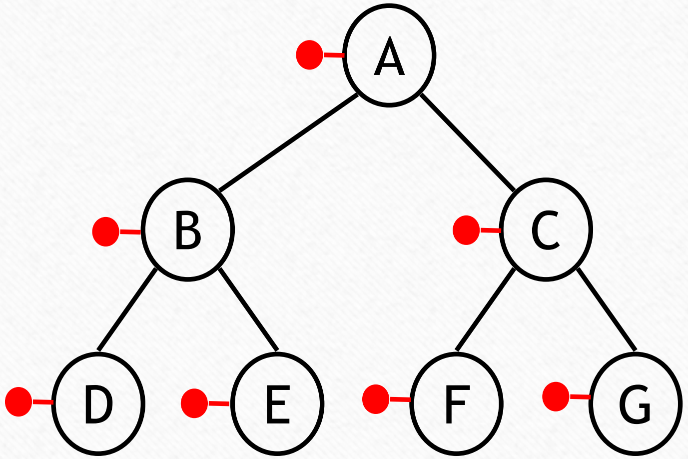
Preorder



Preorder: A B D E C F G

```
public void preorderPrint(Node N) {
        Base
        Visit Node
        Visit Left
        Visit Right
}
```

Preorder



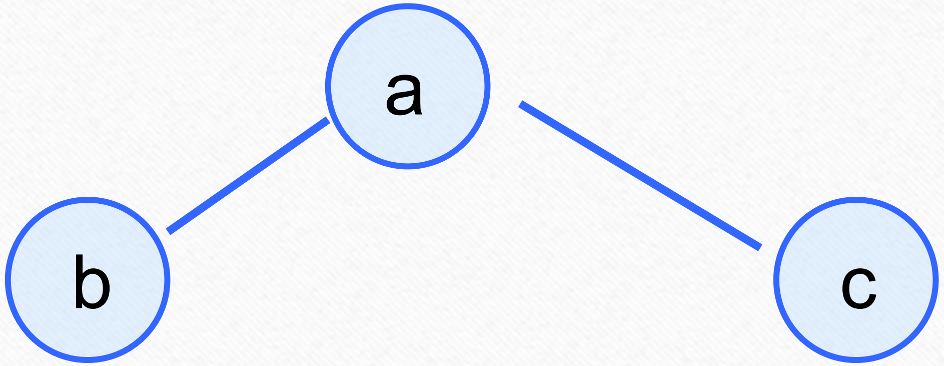
Preorder: A B D E C F G

```
public void preorderPrint(Node N) {

    if (N == null) return;
    System.out.println(N.value);

    preorderPrint(N.leftChild);
    preorderPrint(N.rightChild);
}
```

Application of Preorder

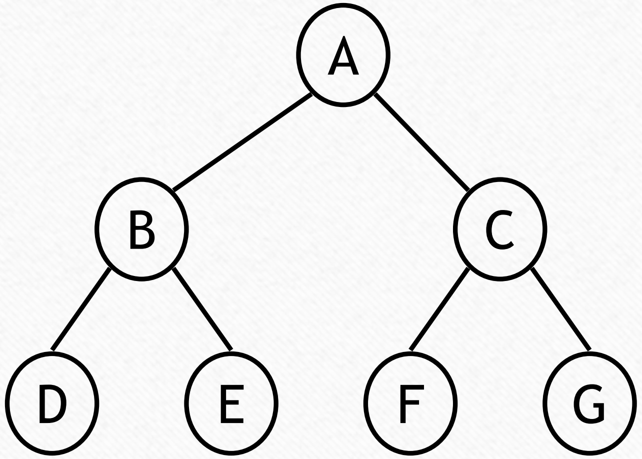
- Directory Trees

Inorder



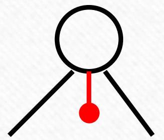
b a c

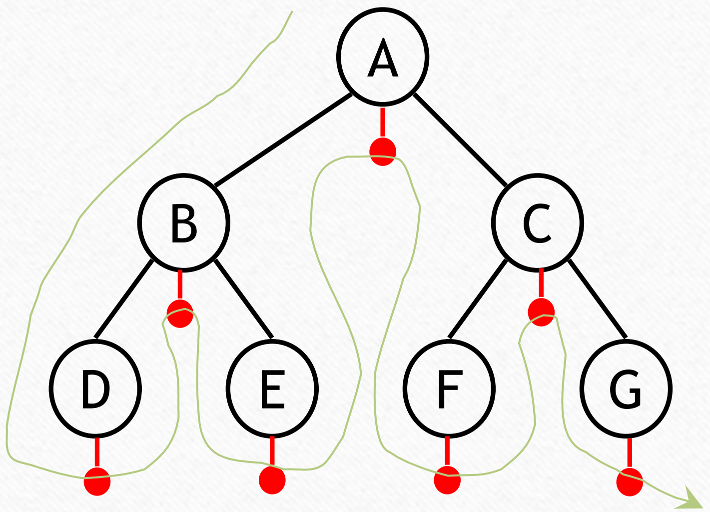
Example



Inorder: D B E A F C G

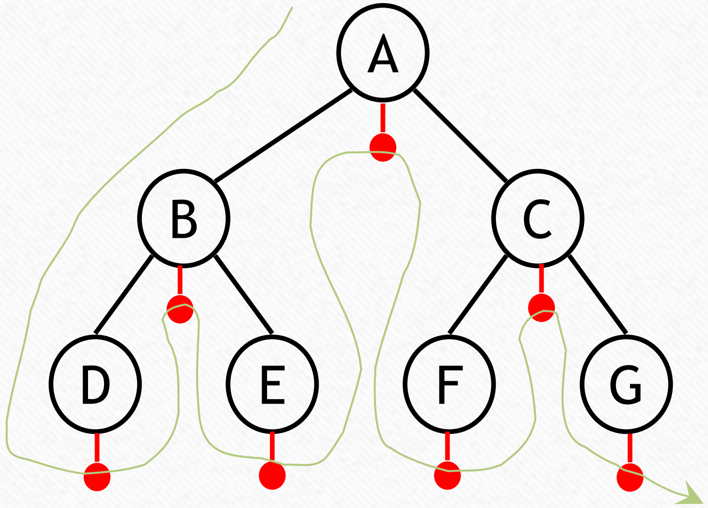
Example





Inorder: D B E A F C G

Inorder



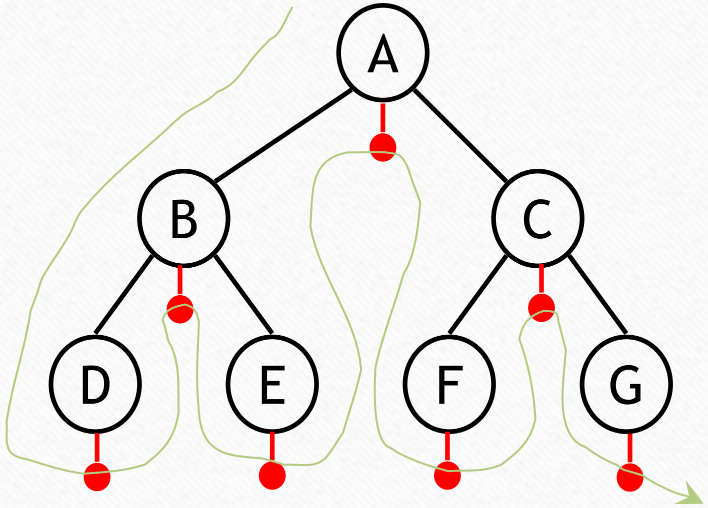
Inorder: D B E A F C G

```
public void inorderPrint(Node N) {
```

Create Pseudo Code

```
}
```

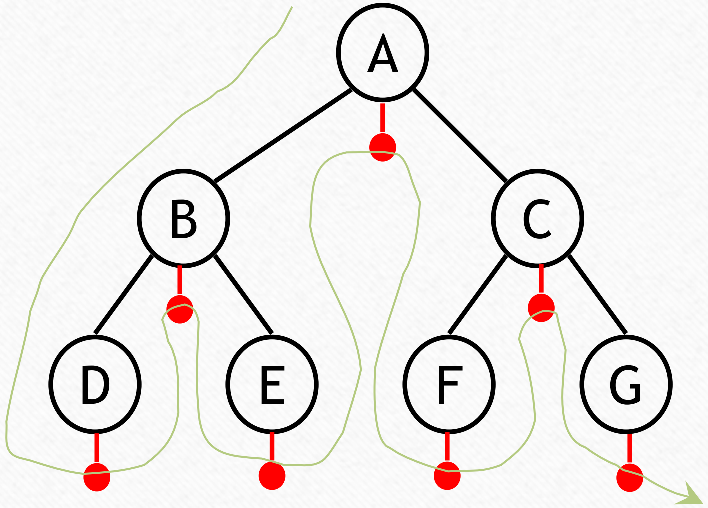
Inorder



Inorder: D B E A F C G

```
public void inorderPrint(Node N) {

    base
    visit left
    visit node
    visit right
}
```

Inorder

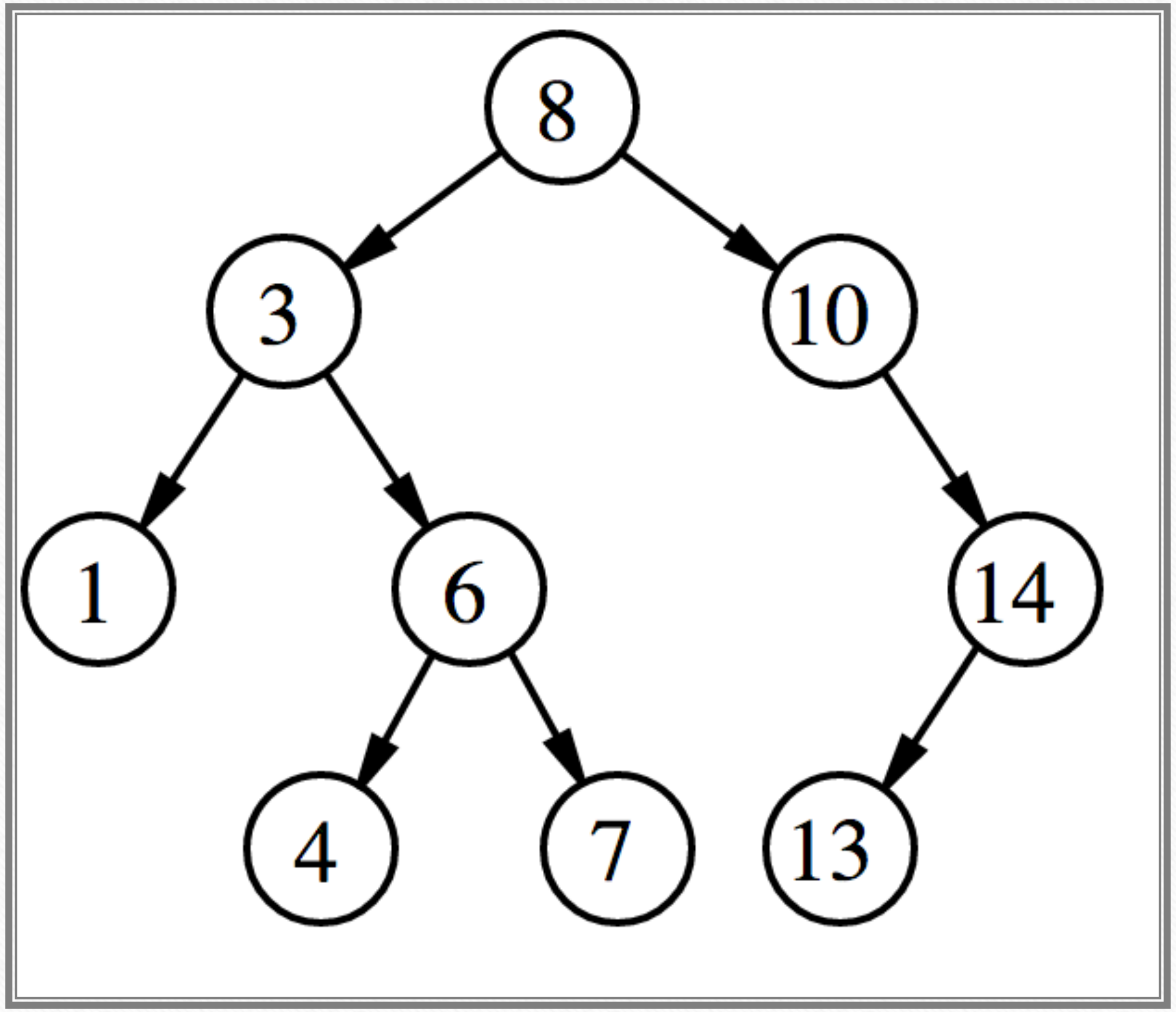


Inorder: D B E A F C G

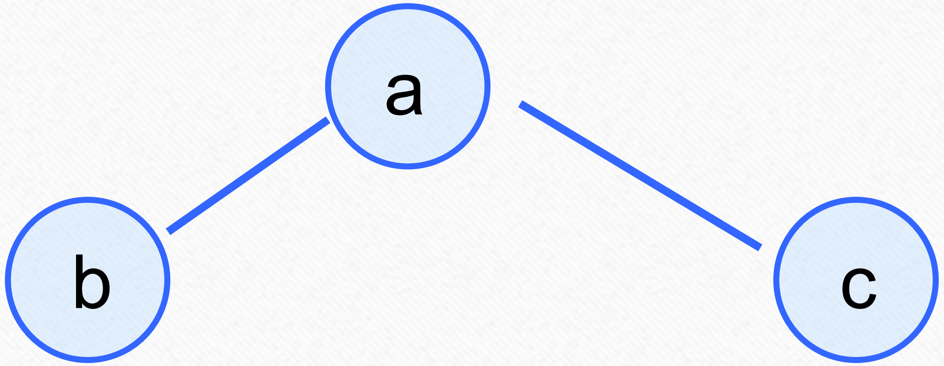
```
public void inorderPrint(Node N) {

        if (N == null) return;
        inorderPrint(N.leftChild);
        System.out.println(N.value);
        inorderPrint(N.rightChild);
}
```

Example of Inorder

- Returns the ordered list of a Binary Search Tree

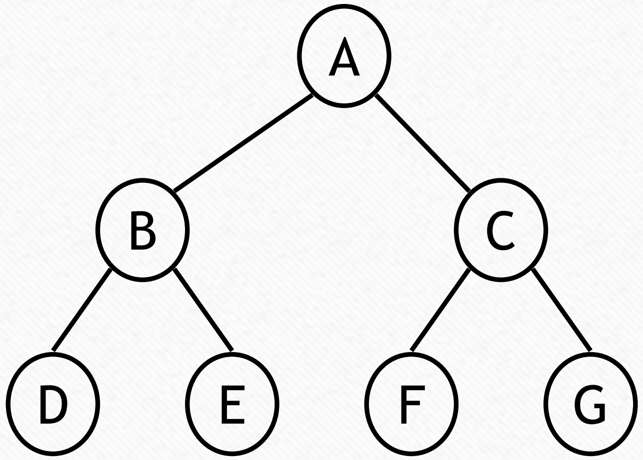


Postorder



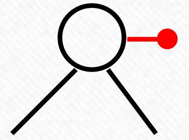
b c a

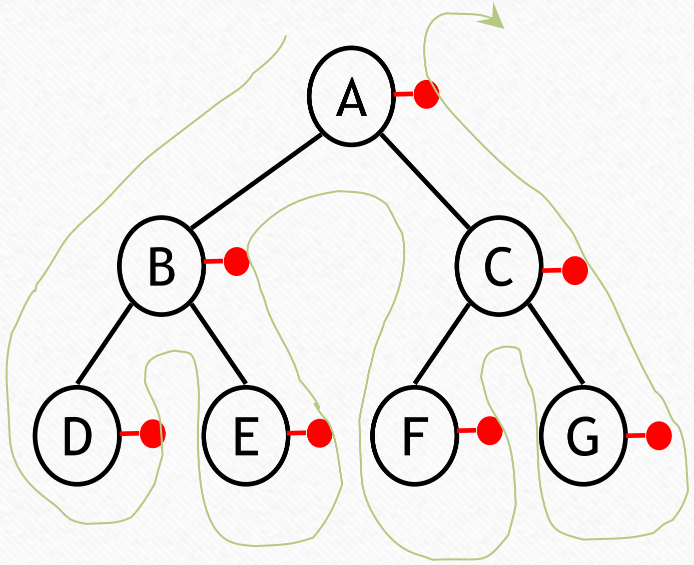
Example



postorder: D E B F G C A

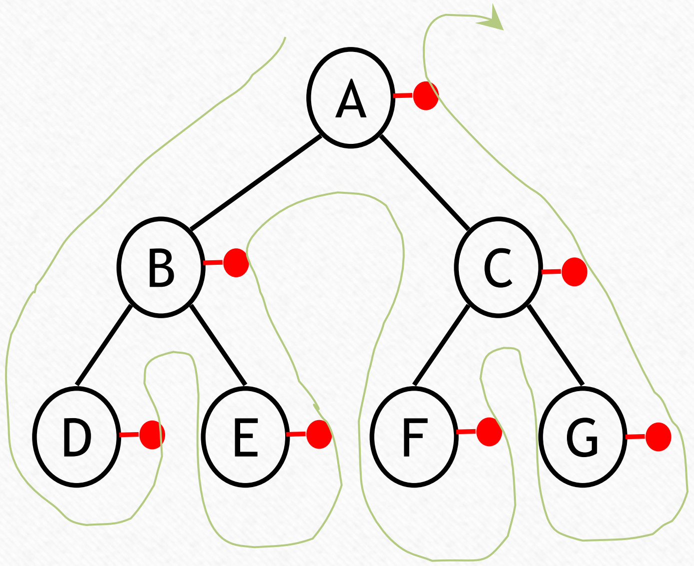
Example





Postorder: D E B F G C A

Post-order



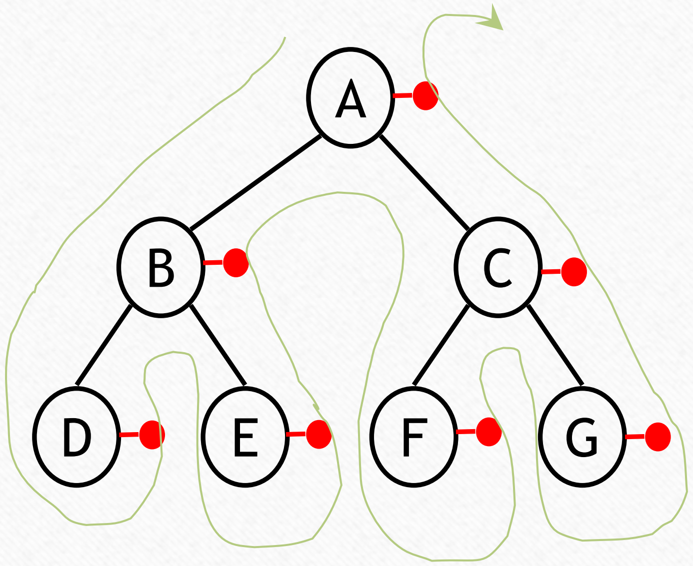
Postorder: D E B F G C A

```
public void postorderPrint(Node N) {
```

Create Pseudo Code

```
}
```

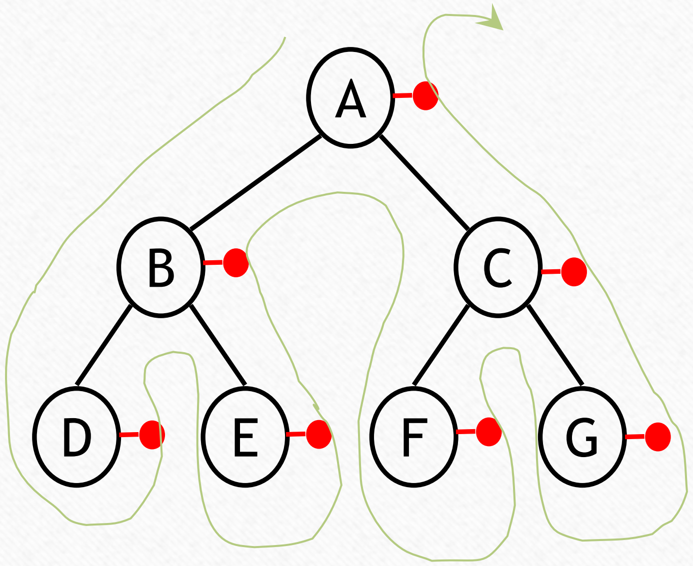
Post-order



Postorder: D E B F G C A

```
public void postorderPrint(Node N) {

    base

    Visit Left
    Visit Right

    Visit Node
}
```

Post-order



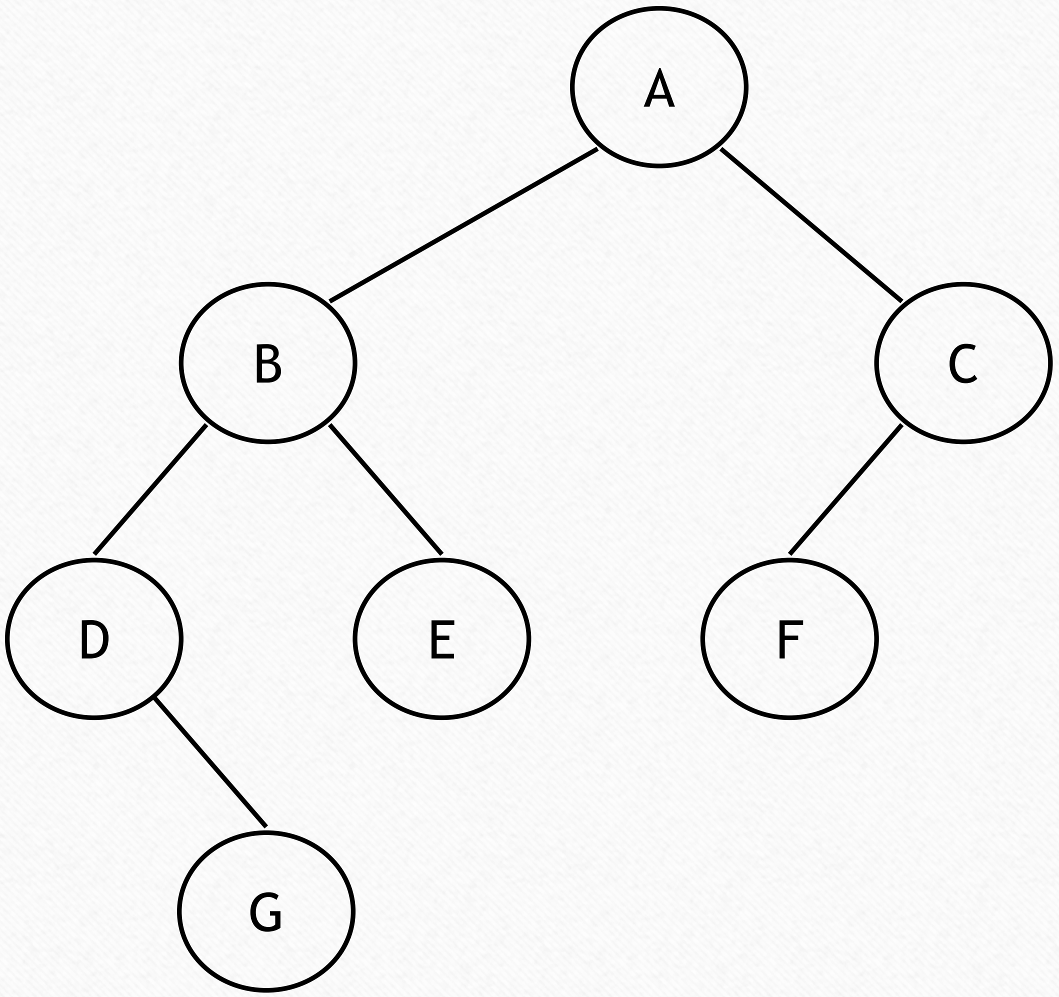
Postorder: D E B F G C A

```
public void postorderPrint(Node N) {

    if (N == null) return;

    postorderPrint(N.leftChild);
    postorderPrint(N.rightChild);

    System.out.println(N.value);
}
```

Example

For this graph, what is the:

- Pre-order
- In-order
- Post-order



Try it/ homework

1. Chapter 4.2, page 142, question 1
2. Chapter 5.3, page 185, questions 5,6