TREES

PART I : TREES, BINARY TREES, TREE TRAVERSAL

GRAPHS

A graph is an abstract data type that consists of a set of nodes (vertices) and a set of edges.

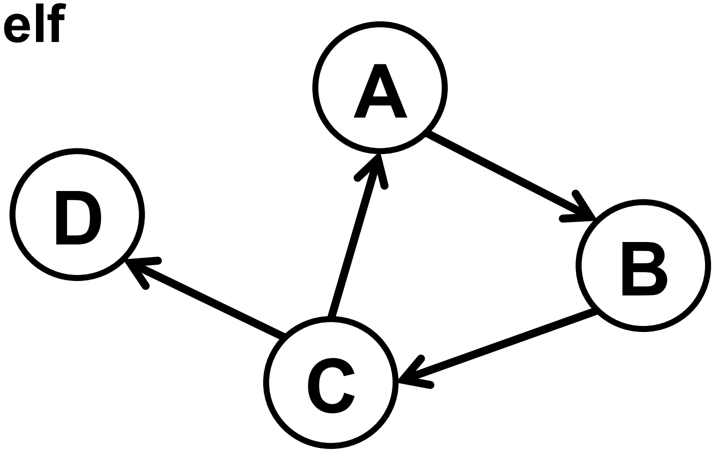
- **Nodes are data elements**
- **Edges are links between nodes**
  - Directed Edges (e.g. from A to B)
  - Undirected Edges (e.g between C and D)

ADJACENCY

- A is adjacent to B (there is an edge from A to B)
- B is not adjacent to A
- C is adjacent to D
- D is adjacent to C

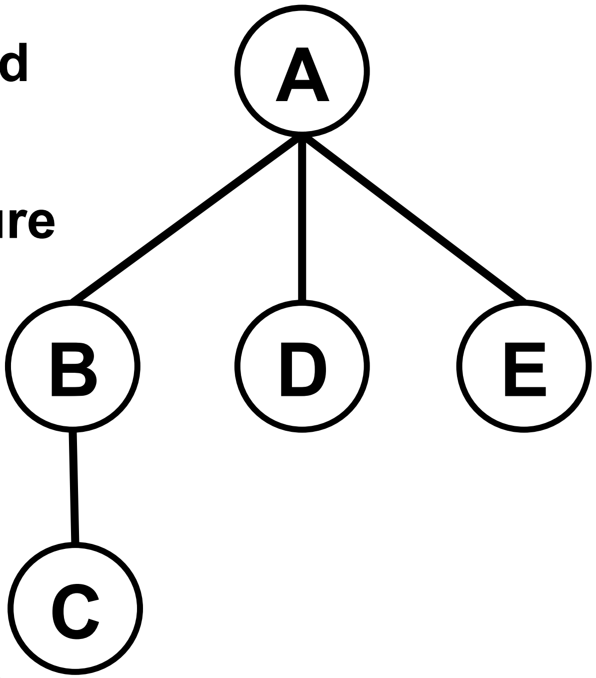
CYCLES

- A cycle is a path from a node to itself
- A→ B→ C→ A is a cycle
- The graph is connected



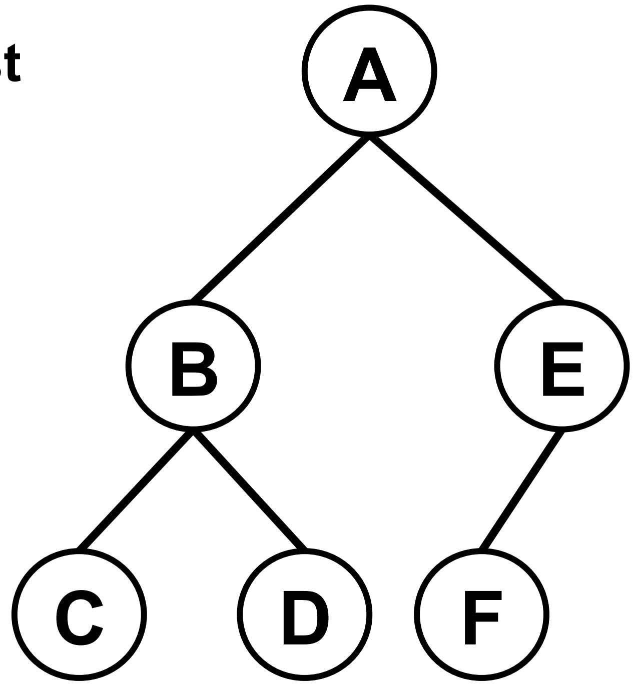
TREES

- A tree is a connected acyclic undirected graph.
- A rooted tree has a hierarchical structure
- A is the root of the tree
- B, D, and E are children of A
- B is the parent of C
- C, D, and E are leaf node
- Rooted trees have a natural orientation away from the root.



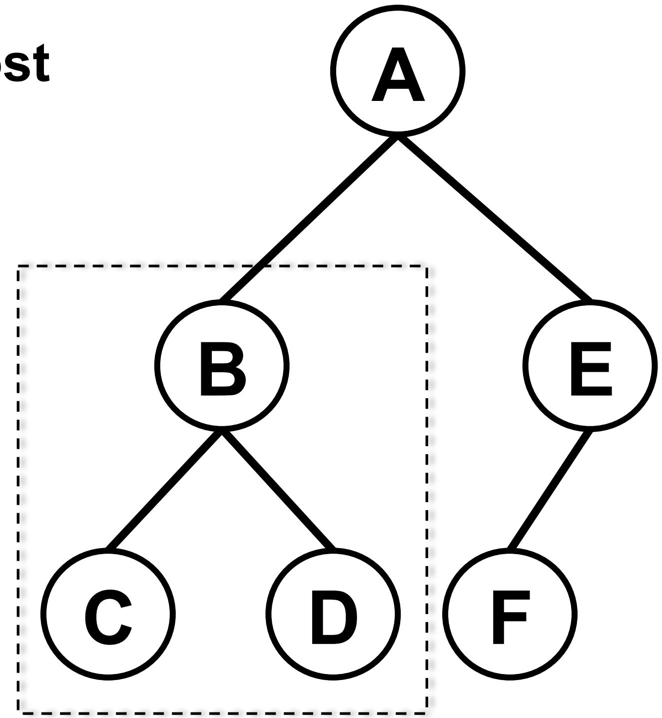
BINARY TREES

- In a binary tree, each node has at most 2 children.
- B is the left child of A
- E is right child of A
- E only has a left child, F
- Each node and its descendants is a subtree



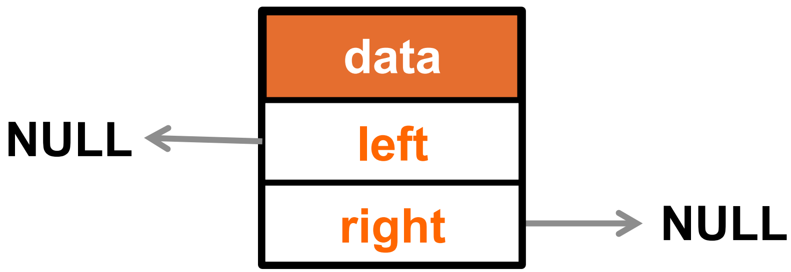
BINARY TREES

- In a binary tree, each node has at most 2 children.
- B is the left child of A
- E is right child of A
- E only has a left child, F
- Each node and its descendants is a subtree
- The tree rooted at B is the left subtree of A



BINARY TREE
IMPLEMENTATION

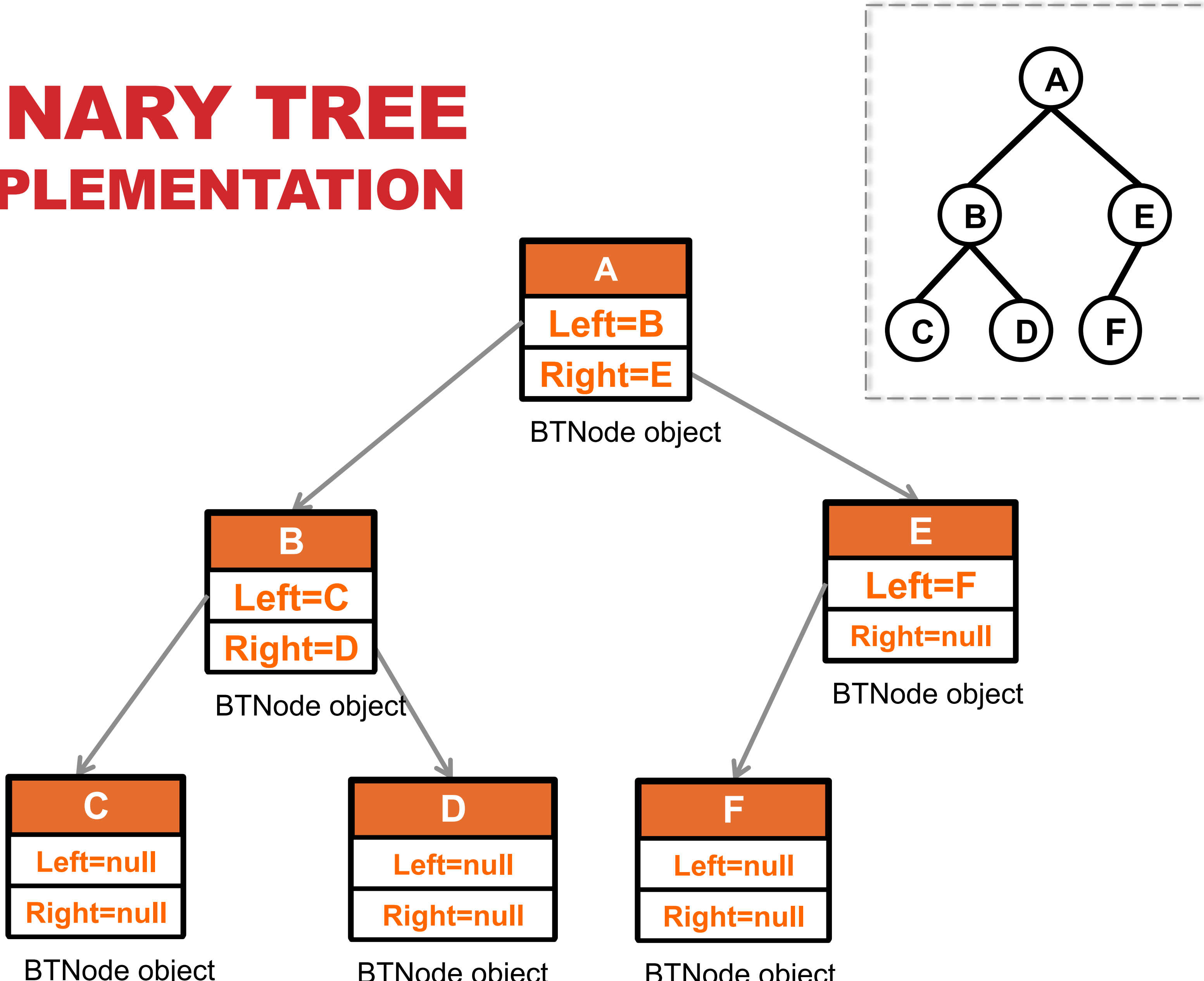
- **The basic component of a binary tree is a binary tree node**
- **A binary tree node (BTNode) consist of 3 parts:**
  - A data element
  - A reference to the left child
  - A reference to right child



BTNode object

  - In a leaf node, left and right references are set to **null**

BINARY TREE
IMPLEMENTATION



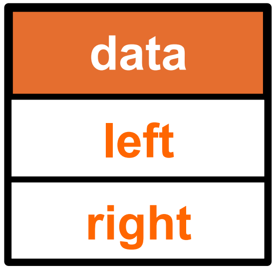
BINARY TREE
IMPLEMENTATION

- Class BTNode, with integer data

```
public class BTNode {

    private int data;
    private BTNode left;
    private BTNode right;

}
```



BTNode object

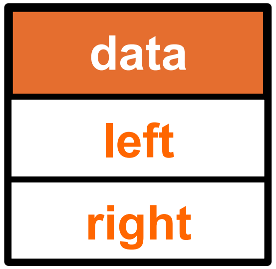
BINARY TREE
IMPLEMENTATION

- Class BTNode, with generic data

```
public class BTNode<T> {

    private T data;
    private BTNode<T> left;
    private BTNode<T> right;

}
```



BTNode object

BINARY TREE
IMPLEMENTATION

- **Constructor for the BTNode class**

```
public BTNode(T initialData, BTNode<T> initialLeft, BTNode<T> initialRight)
 {
          data = initialData;
           left   = initialLeft;
           right = initialRight;
 }
```

- **Example:**

```
BTNode<String> B=new BTNode<String>("B", null, null); //leaf node
BTNode<String> E=new BTNode<String>("E", null, null; //leaf node
BTNode<String> A=new BTNode<String>("A",B,E); // A is parent of B and E
```

BINARY TREE
IMPLEMENTATION

- **Methods that involve an operation in tree nodes are usually implemented with recursion**
- **For example, to get the value of the left-most node:**

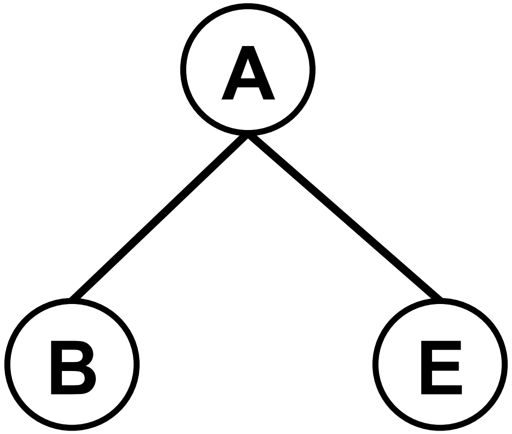
```
Public T getLeftmostData( )
 {
     if (left == null)
           return data;
     else
           return left.getLeftmostData( );
}
```

TREE TRAVERSAL

- **Tree traversal: each node is visited once**
  - Visiting a node means performing some operation on that node (e.g. printing its value)
- **Types of tree traversal:**
  - Pre-order
  - In-order
  - Post-order
- **Tree traversal methods are recursive**

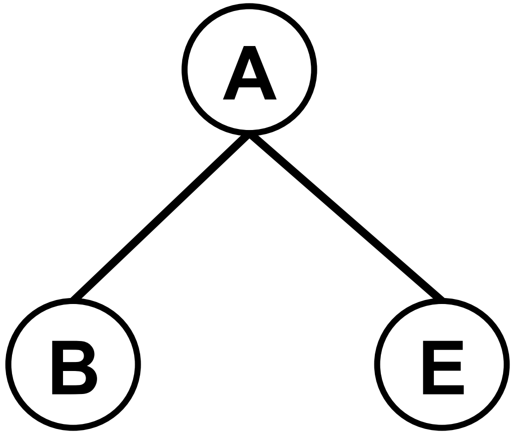
PREORDER TRAVERSAL

- **The parent node is visited before its children**
    - Visit parent node
    - Traverse left subtree
    - Traverse right subtree



PREORDER TRAVERSAL

- **The parent node is visited before its children**
  - Visit node
  - Traverse left subtree
  - Traverse right subtree

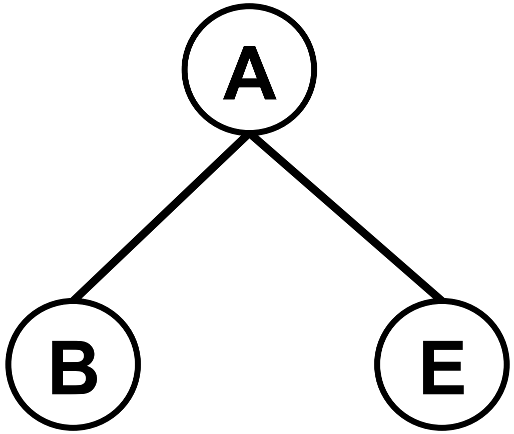


1. **Visit A**
2. **Traverse left child of A**
   1. Visit B
3. **Traverse right child of A**
   1. Visit E

Output:
A B E

INORDER TRAVERSAL

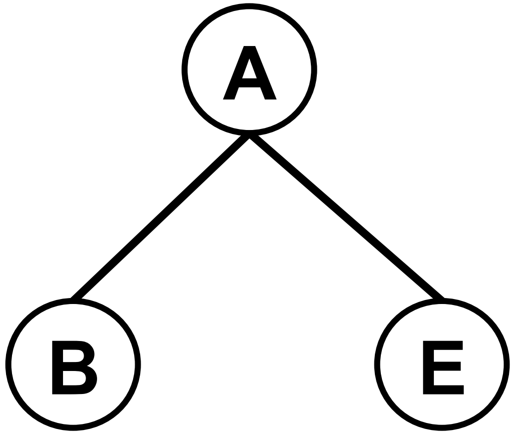
- **The parent node is visited after its left child, and before its right child**
  - Traverse left subtree
  - Visit node
  - Traverse right subtree



INORDER TRAVERSAL

- **The parent node is visited after its left child, and before its right child**
  - Traverse left subtree
  - Visit node
  - Traverse right subtree

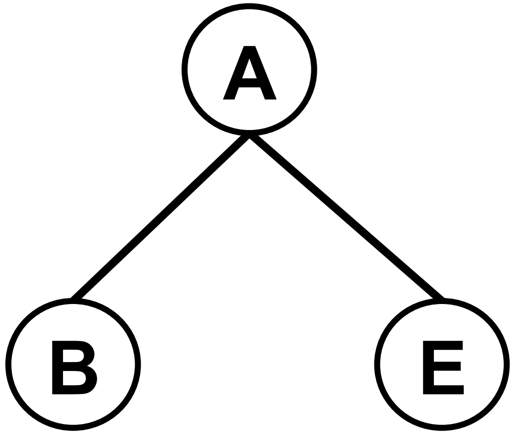
1. **Traverse left child of A**
   1. Visit B
2. **Visit A**
3. **Traverse right child of A**
   1. Visit E



Output:
B A E

POSTORDER TRAVERSAL

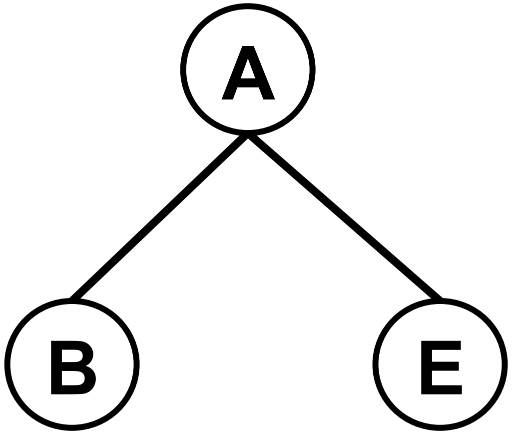
- **The parent node is visited after its children**
  - Traverse left subtree
  - Traverse right subtree
  - Visit node



POSTORDER TRAVERSAL

- **The parent node is visited after its children**
  - Traverse left subtree
  - Traverse right subtree
  - Visit node

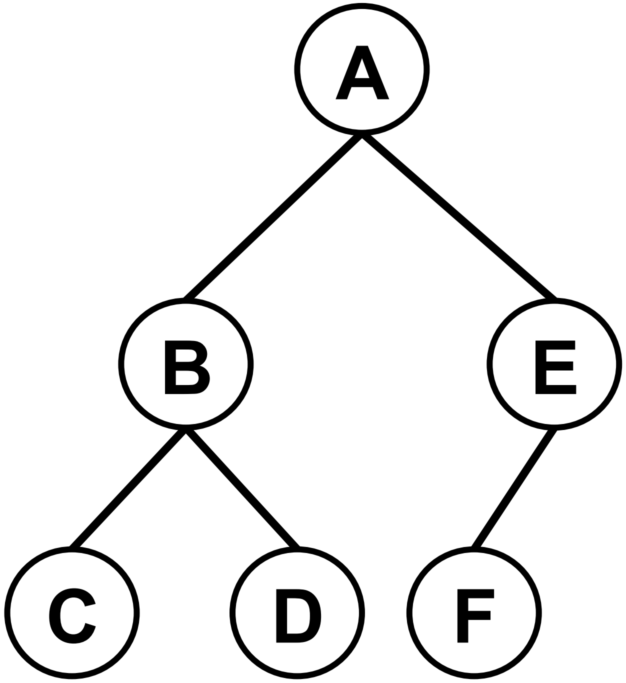
1. **Traverse left child of A**
   1. Visit B
2. **Traverse right child of A**
   1. Visit E
3. **Visit A**



Output:
B E A

TREE TRAVERSAL

- Example: what is the output of printing the following tree using preorder, inorder, and postorder traversal?



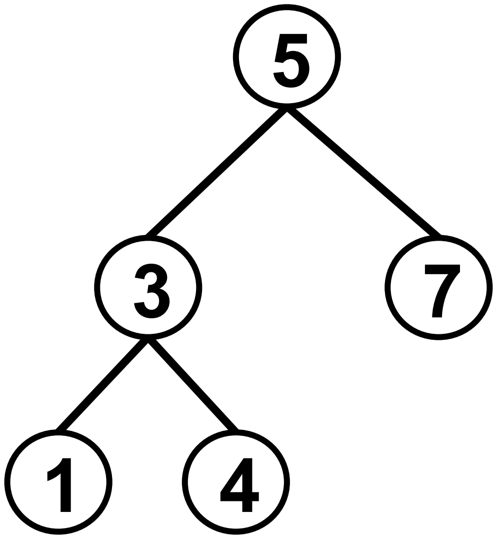
TREES

PART II : BINARY SEARCH TREES

BINARY SEARCH TREES

Definition: A binary search tree (BST) is a binary tree where each node has a key, and:

- All keys in the left subtree of a node are smaller
- All keys in the right subtree are larger



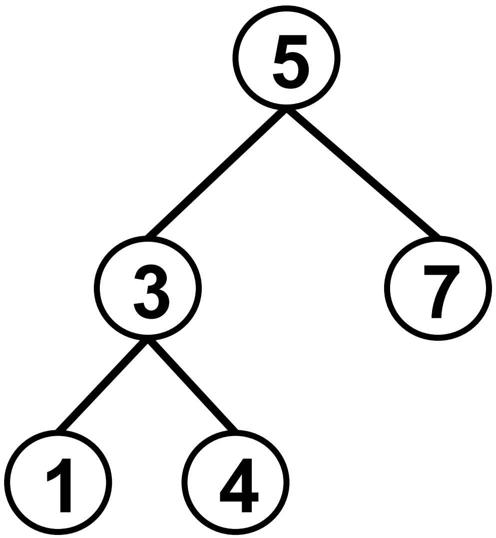
BST SEARCH ALGORITHM

Search (value)

- **Compare the search value with the current key**
- **If equal, return true**
- **If smaller**
  - If left child is null, return false
  - If left child is not null, search value in left subtree
- **If larger**
  - If right child is null, return false
  - If right child is not null, search value in right subtree

BST SEARCH DEMO

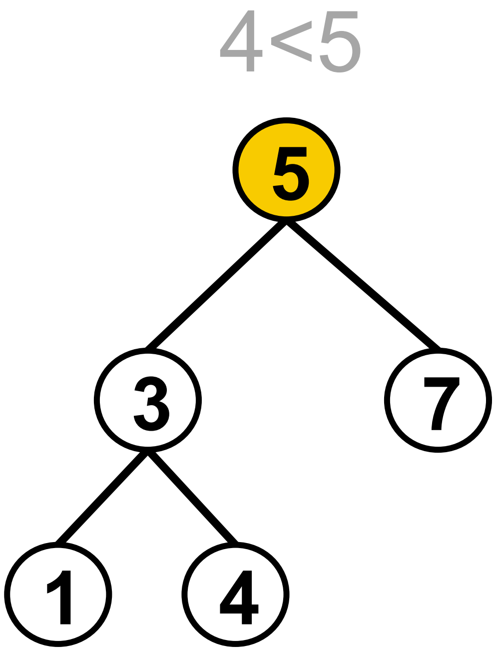
Search for 4



BST SEARCH DEMO

Search for 4

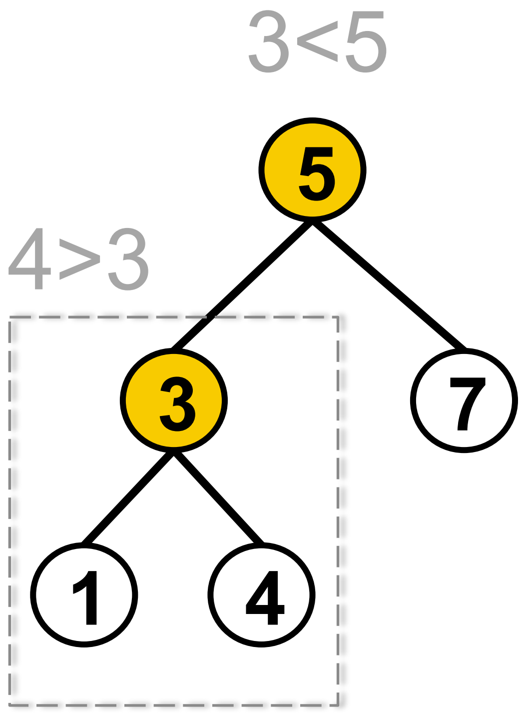
- **Starting at the root**
- **Compare current node with 4**
    - Search left subtree



BST SEARCH DEMO

Search for 4

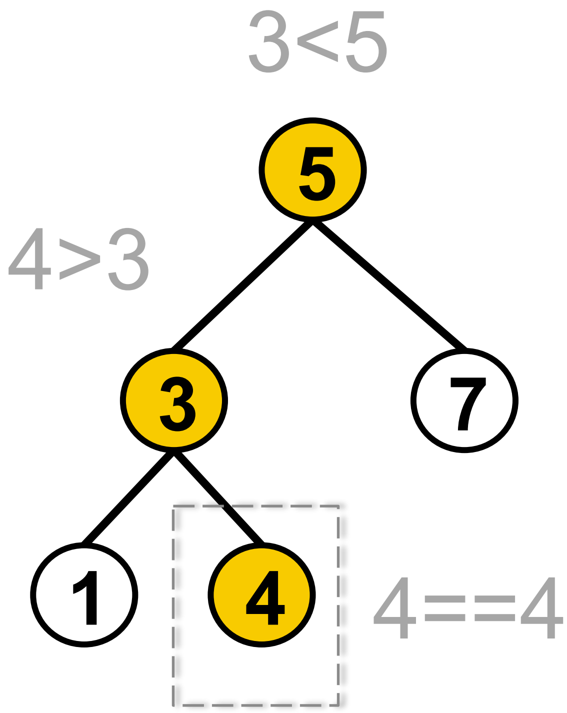
- **Compare with the current node**
  - Search right subtree



BST SEARCH DEMO

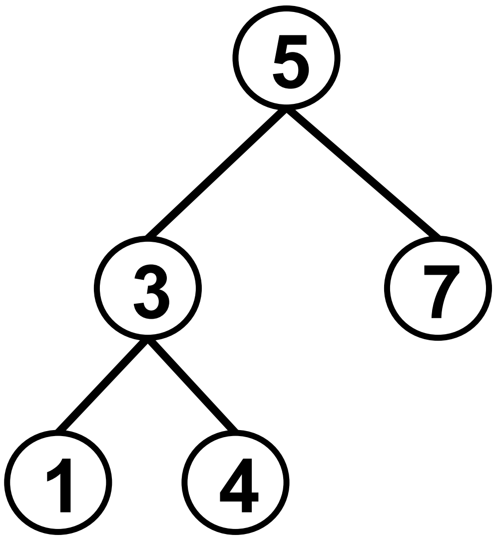
Search for 4

- **Compare with the current node**
  - Equal, return true



BST SEARCH DEMO

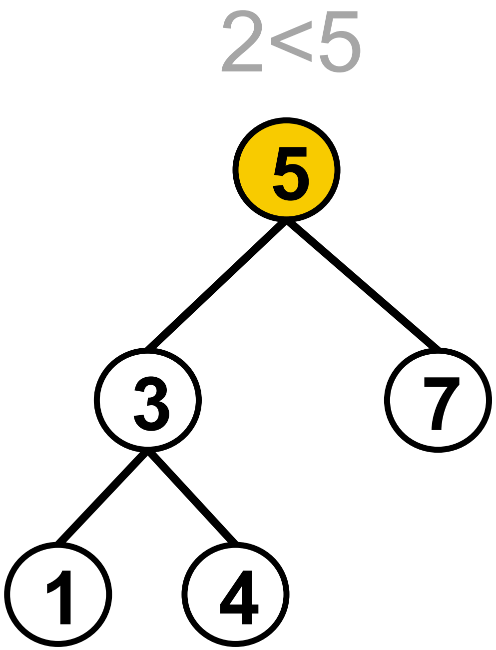
Search for 2



BST SEARCH DEMO

Search for 2

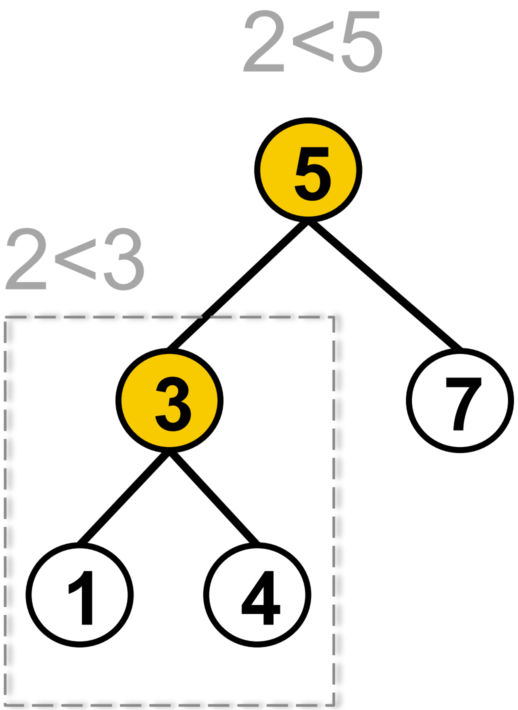
- **Starting at the root**
- **Compare current node with 3**
  - Search left subtree



BST SEARCH DEMO

Search for **2**

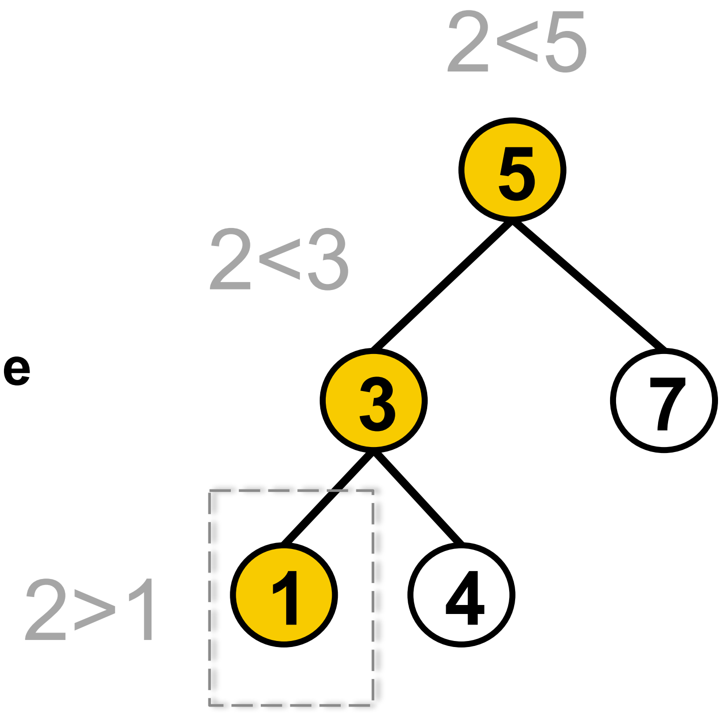
- **Compare with the current node**
  - Search left subtree



BST SEARCH DEMO

Search for 2

- **Compare with the current node**
  - Right child is null
  - Key not found, return false



BST INSERTION ALGORITHM

If root is null, create a new root node with the value

Insert (value)

- **If value < current node**
    - If left child is null, insert new value as a left child
    - If left child is not null, insert in left subtree
- **If value > current node**
    - If right child is null, insert new value as a right child
    - If right child is not null, insert in right subtree

BST INSERTION DEMO

Insert **5**

BST INSERTION DEMO

Insert 5

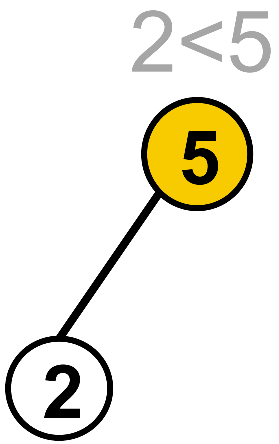
- Root is null, create new root node

BST INSERTION DEMO

Insert **2**

BST INSERTION DEMO

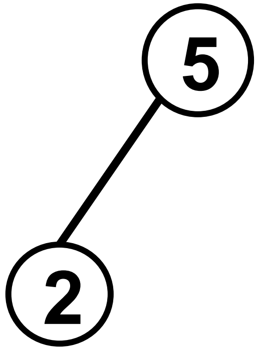
Insert 2



- **Compare with current node**
- **Left child is null**
  - Insert as left child of current node

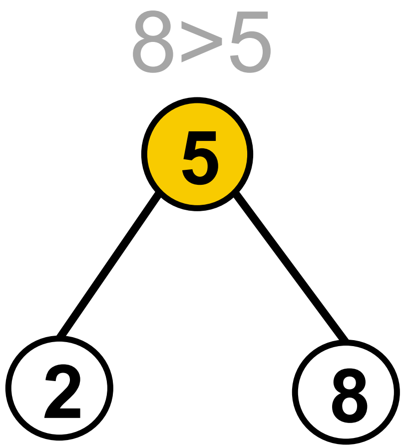
BST INSERTION DEMO

Insert **8**



BST INSERTION DEMO

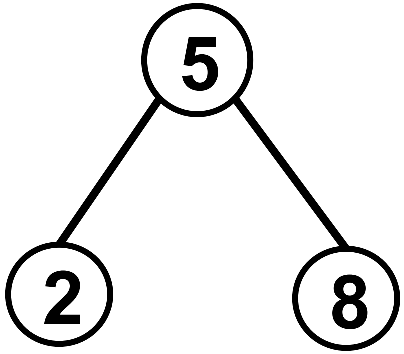
Insert 8



- **Compare with current node**
- **Right child is null**
  - Insert as right child of current node

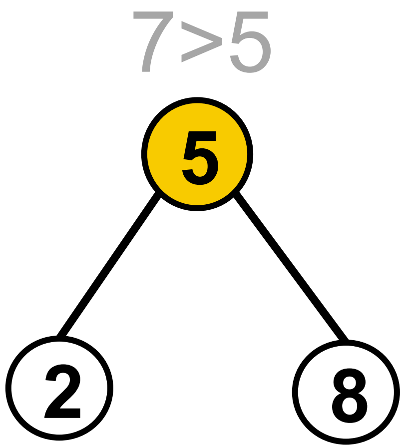
BST INSERTION DEMO

Insert 7



BST INSERTION DEMO

Insert 7

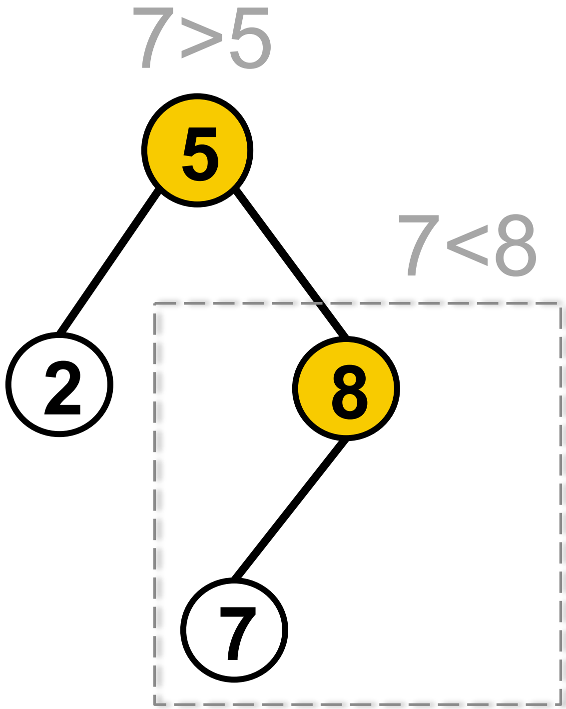


- **Compare with current node**
- **Right child is not null**
  - Insert in right subtree

BST INSERTION DEMO

Insert 7

- **Compare with current node**
- **Left child is null**
  - Insert as left child



BST DELETION ALGORITHM (DELETE MIN)

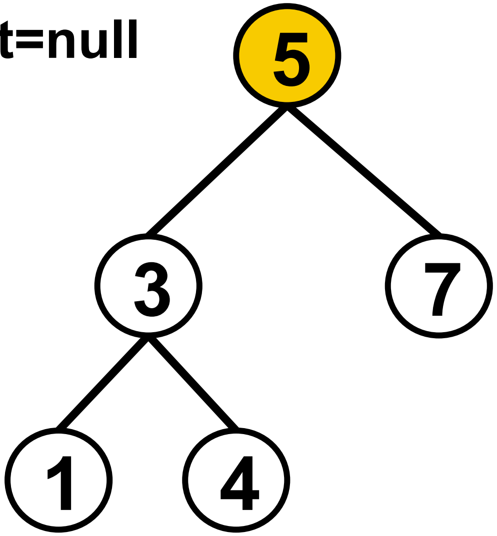
To delete the minimum value in a binary search tree

- **Find a node x that has a null left child (left-most node)**
  - In a BST, the minimum value is the left-most node
- **Set the left child of the parent of x to the right child of x.**
  - xParent.left = x.right;

DELETE MIN DEMO

To delete the minimum value in a binary search tree

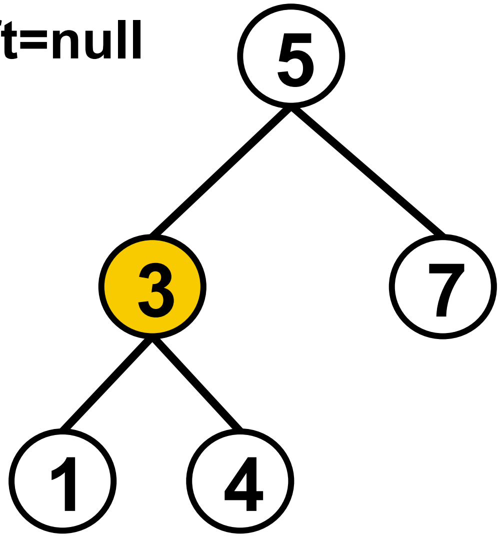
- Starting at the root, find a node with left=null



DELETE MIN DEMO

To delete the minimum value in a binary search tree

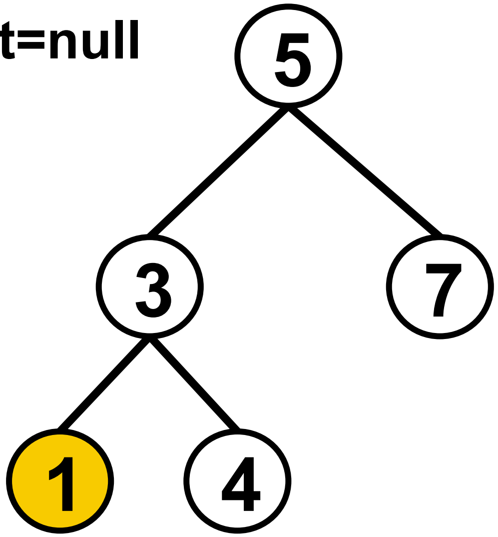
- **Starting at the root, find a node with left=null**



DELETE MIN DEMO

To delete the minimum value in a binary search tree

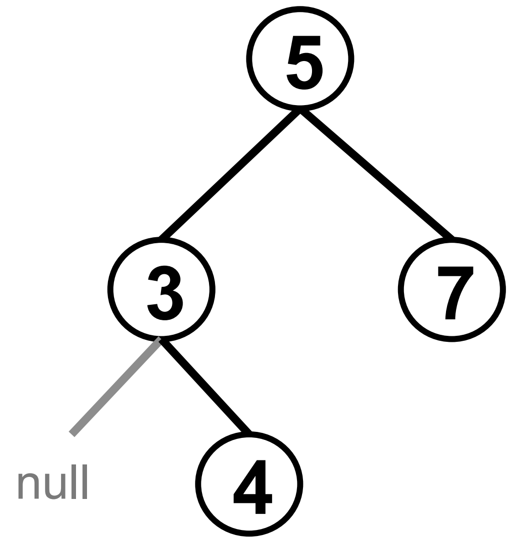
- **Starting at the root, find a node with left=null**
  - Node1.left =null
- **Right child of Node1 is also null**



DELETE MIN DEMO

To delete the minimum value in a binary search tree

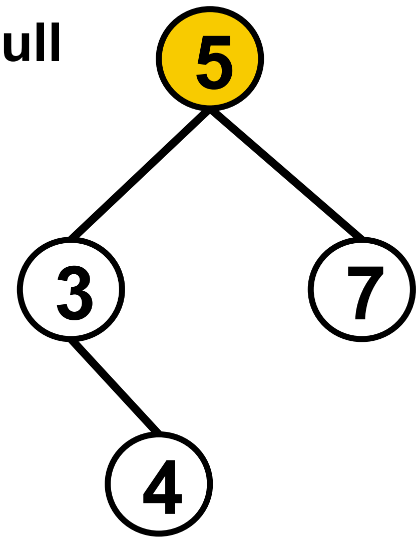
- **Node3.left = Node1.right = null**



DELETE MIN DEMO

To delete the minimum value in a binary search tree

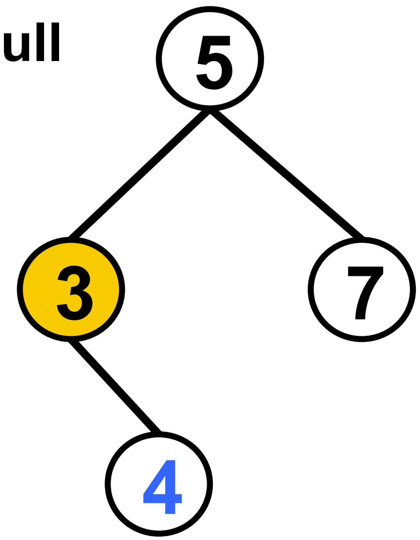
- **Starting at the root, find a node with left=null**



DELETE MIN DEMO

To delete the minimum value in a binary search tree

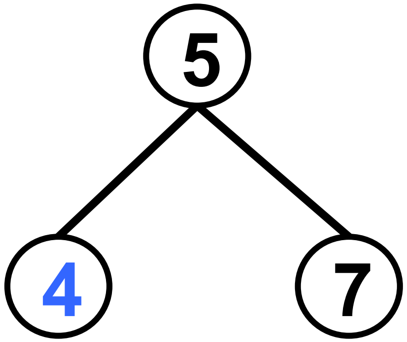
- **Starting at the root, find a node with left=null**
  - Node3.left = null
- **Right child of node 3 is Node4**



DELETE MIN DEMO

To delete the minimum value in a binary search tree

- **Node5.left = Node3.right = Node4**



DELETE MIN CODE

```
public BTNode<T> removeLeftmost( ){
      if (left == null)
         return right;
      else{
         left = left.removeLeftmost( );
         return this;
      }
}
```

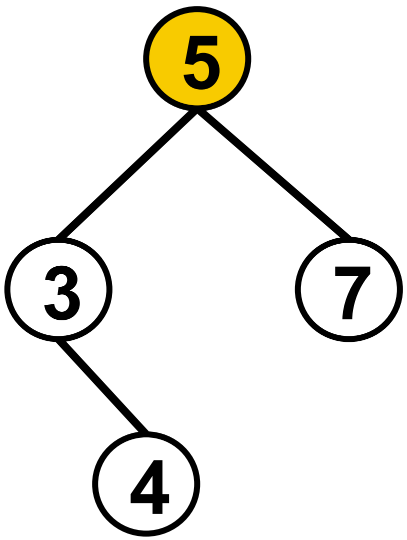
To remove the left most node from the tree:

```
rootNode=rootNode.removeLeftmost();
```

DELETE MIN CODE

```
Node5=Node5.removeLeftMost();
```

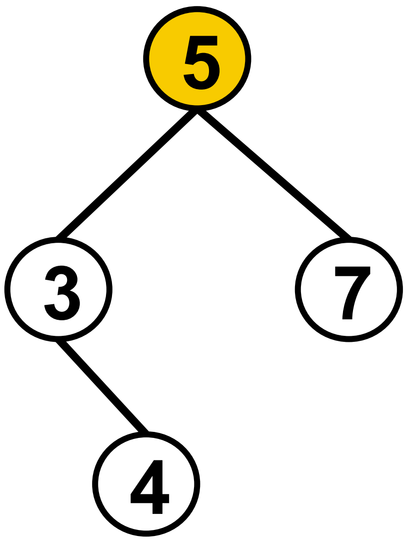
```
public BTNode<T> removeLeftmost( ){
     if (left == null)
       return right;
     else{
       left = left.removeLeftmost( );
       return this;
     }
}
```



DELETE MIN CODE

```
Node5=Node5.removeLeftMost();
```

- **left= left.removeLeftMost();**
- **Return this**; \\this=Node5
  - **Node5=Node5;**

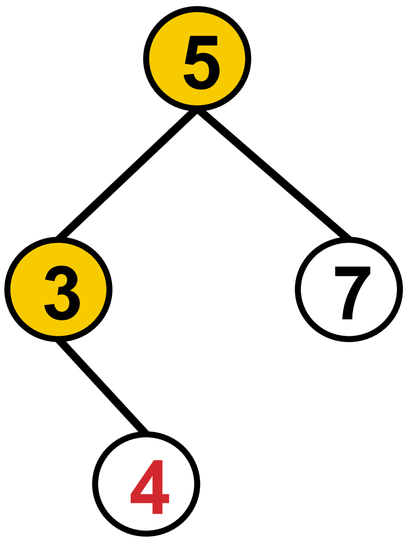


DELETE MIN CODE

```
Node5=Node5.removeLeftMost();
```

- **left= left.removeLeftMost();**
  - → Node5.left=Node3.removeLeftMost() :

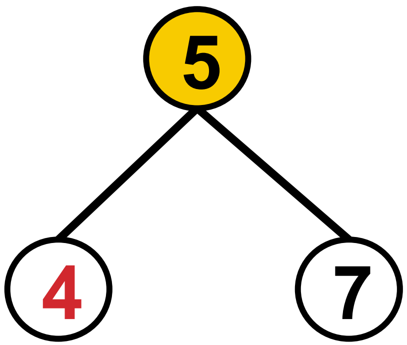
```
public BTNode<T> removeLeftmost( ){
    if (left == null)
      return right;
    else{
      left = left.removeLeftmost( );
      return this;
    }
}
```



DELETE MIN CODE

```
Node5=Node5.removeLeftMost();
```

- **left= left.removeLeftMost();**
  - → **Node5.left=Node4;**
- **Return this**; \\this=Node5
  - **Node5=Node5;**

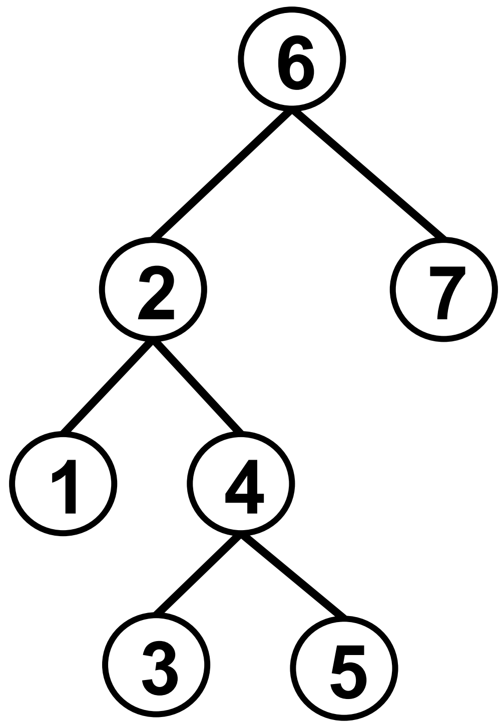


BST DELETION ALGORITHM

- **Deleting a leaf node or a node with a single child is simple**
  - Similar to delete min algorithm
- **Deleting a node with two children**
  - Find the minimum value in the right subtree, call it x
    - This value is larger than all nodes in the left subtree
  - Replace the data value of the node to be deleted with x
  - Delete the minimum from the right subtree

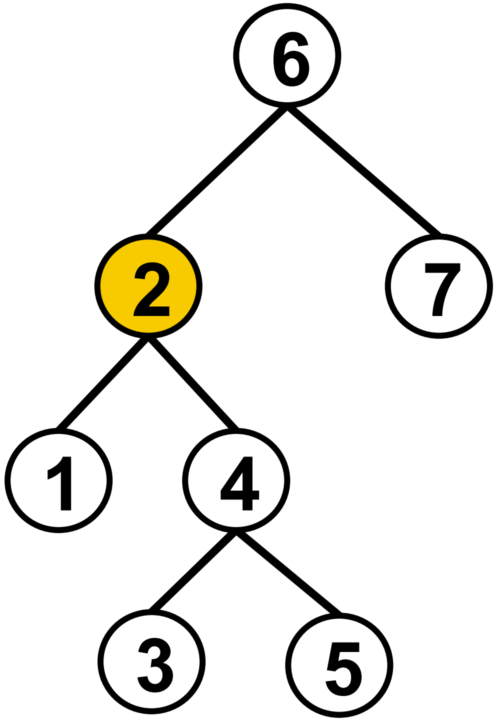
BST DELETION DEMO

- Delete 2



BST DELETION DEMO

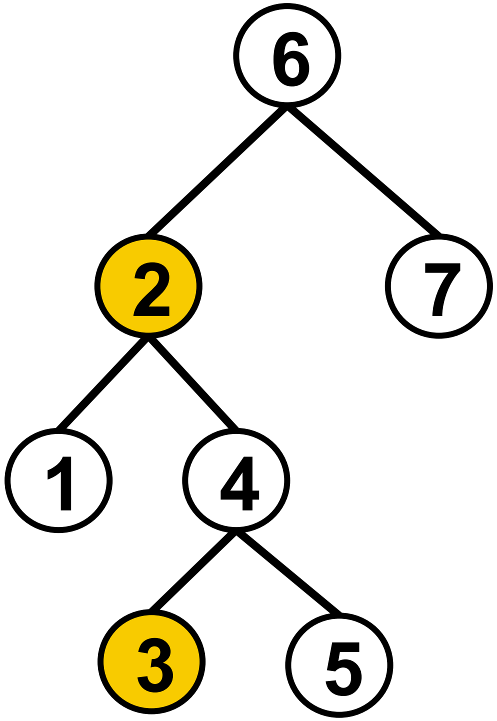
- **Delete 2**
- **First find 2**
  - Node2 has two children



BST DELETION DEMO

- **Delete 2**

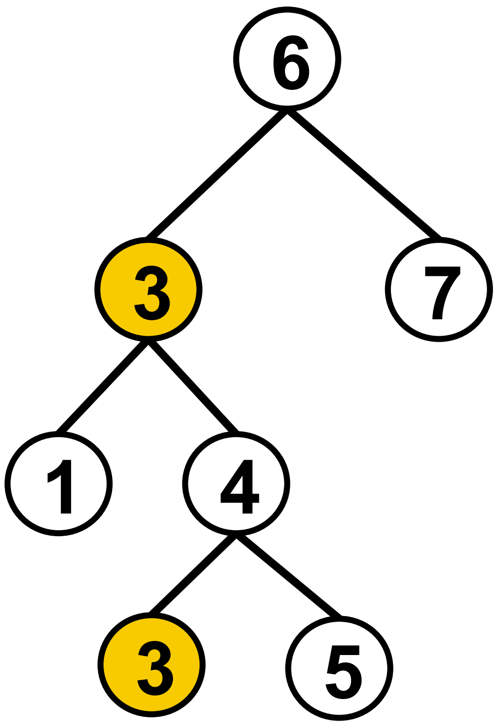
- **Get the min value in the right**
  - right.getLeftMost();



BST DELETION DEMO

- **Delete 2**

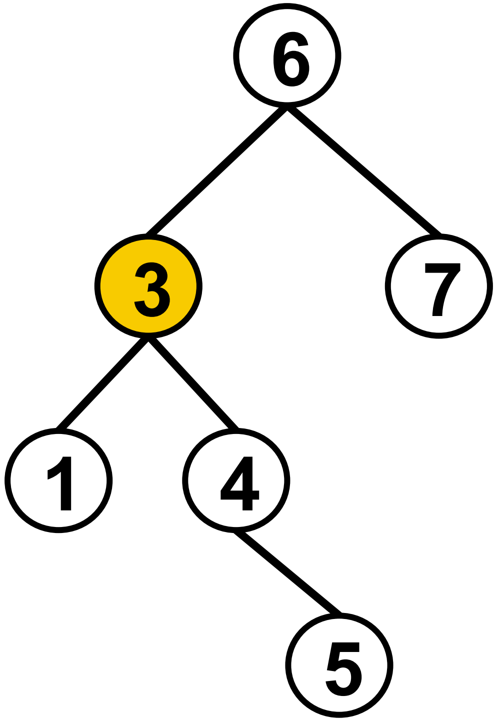
- **Reset the data value to 3**
  - data=right.getLeftMost();



BST DELETION DEMO

- **Delete 2**

- **Delete the left most node in the right subtree**
  - right = right.removeLeftMost();



BST DELETION ALGORITHM

Method: Delete (int D) \\ root=root.Delete(D);

- **If D==data**
  - If right child is null, return left child
  - If left child is null, return right child
  - otherwise
    - data=minimum value in the right subtree
    - Remove minimum value from the right subtree
- **If D < data**
  - left=left.Delete(D), if left is not null
- **If D > data**
  - right=right.Delete(D), if right is not null
- **Return this**