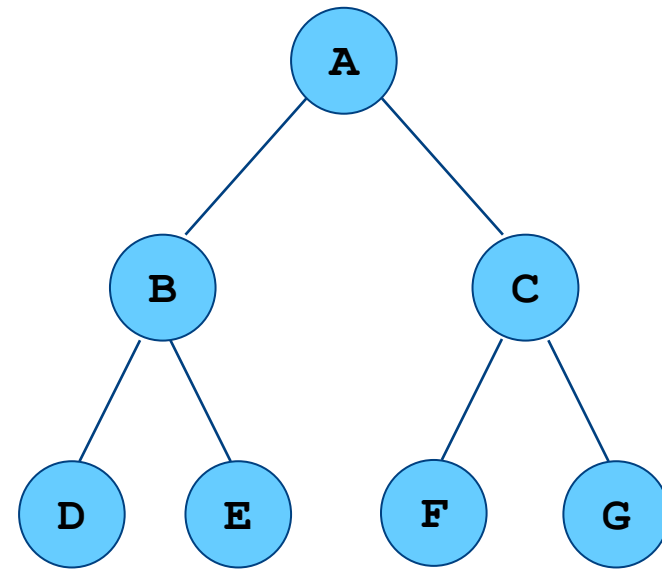


Trees part 2.

Full Binary Trees

- A binary tree is **full** if all the *internal* nodes (nodes other than leaves) has two children and if all the leaves have the same depth
 - A full binary tree of height h has $(2^h - 1)$ nodes, of which 2^{h-1} are leaves (can be proved by induction on the height of the tree).

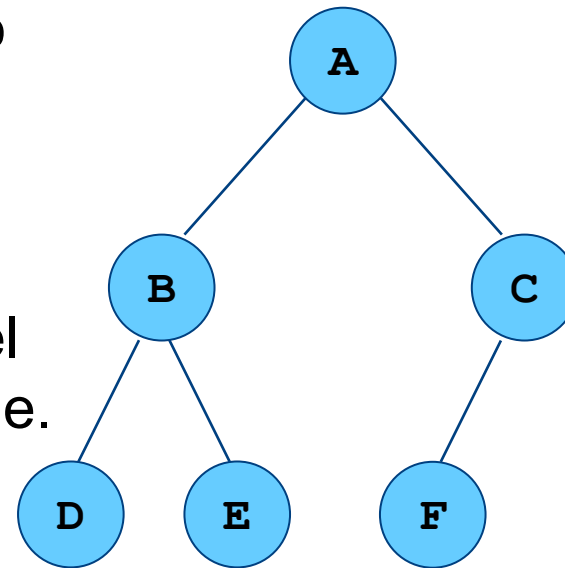


Full binary tree

Height of this tree is 3 and it has $2^3 - 1 = 7$ nodes of which $2^{3-1} = 4$ of them are leaves.

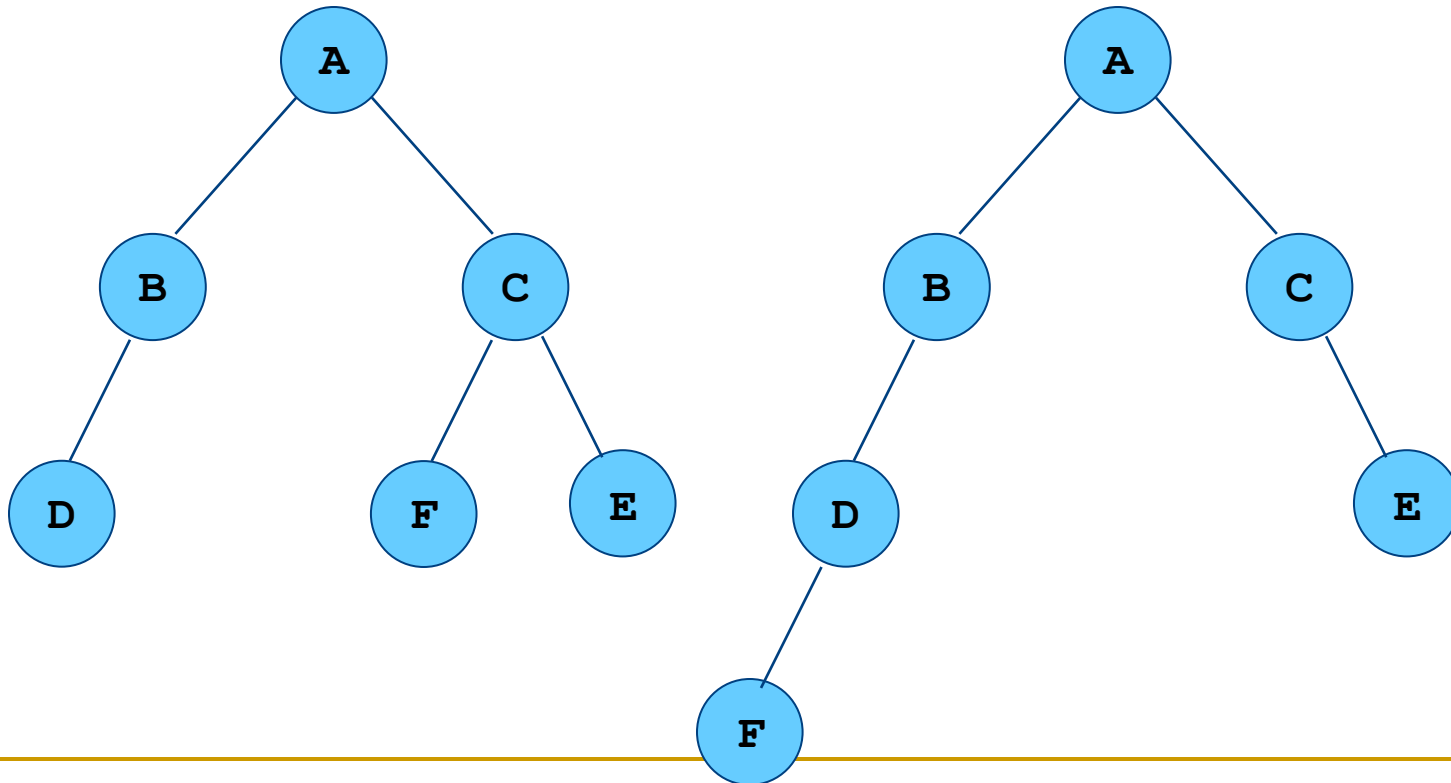
Complete Binary Trees

- A **complete** binary tree is one where
 - The leaves are on at most two different levels,
 - The second to bottom level is filled in (has 2^{h-2} nodes) and
 - The leaves on the bottom level are as far to the left as possible.



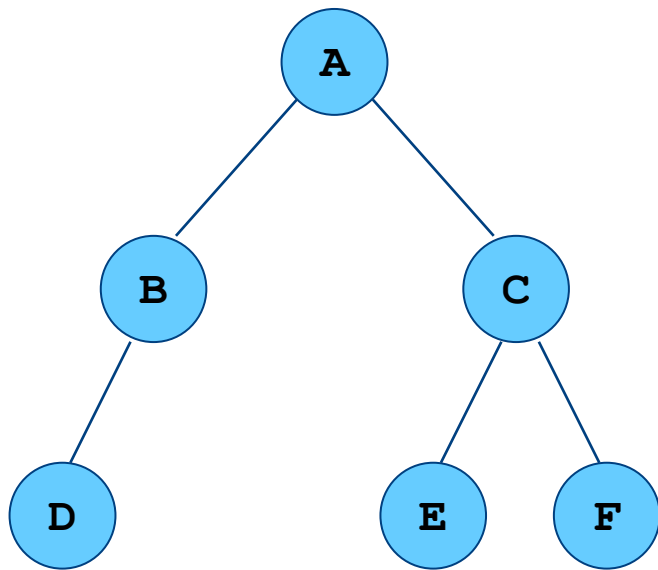
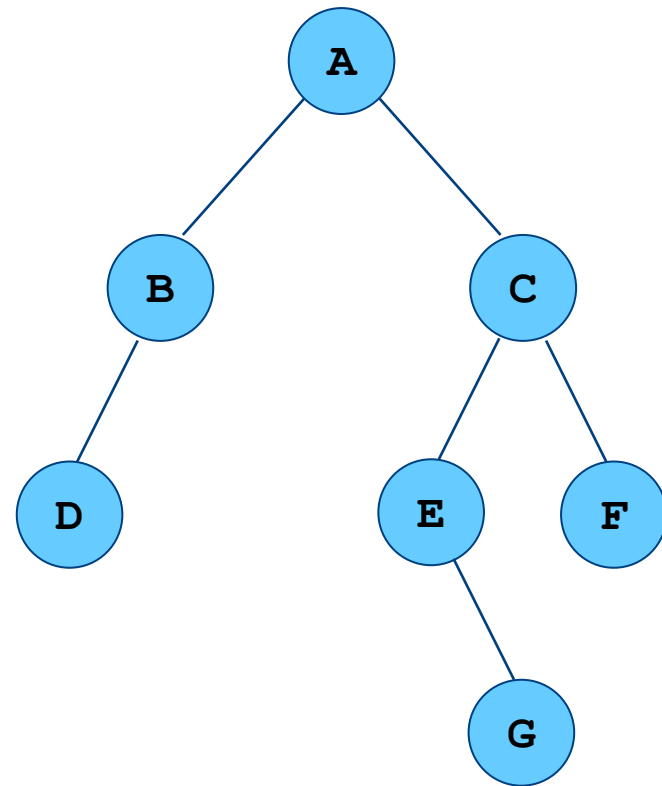
Complete binary tree

■ Not complete binary trees

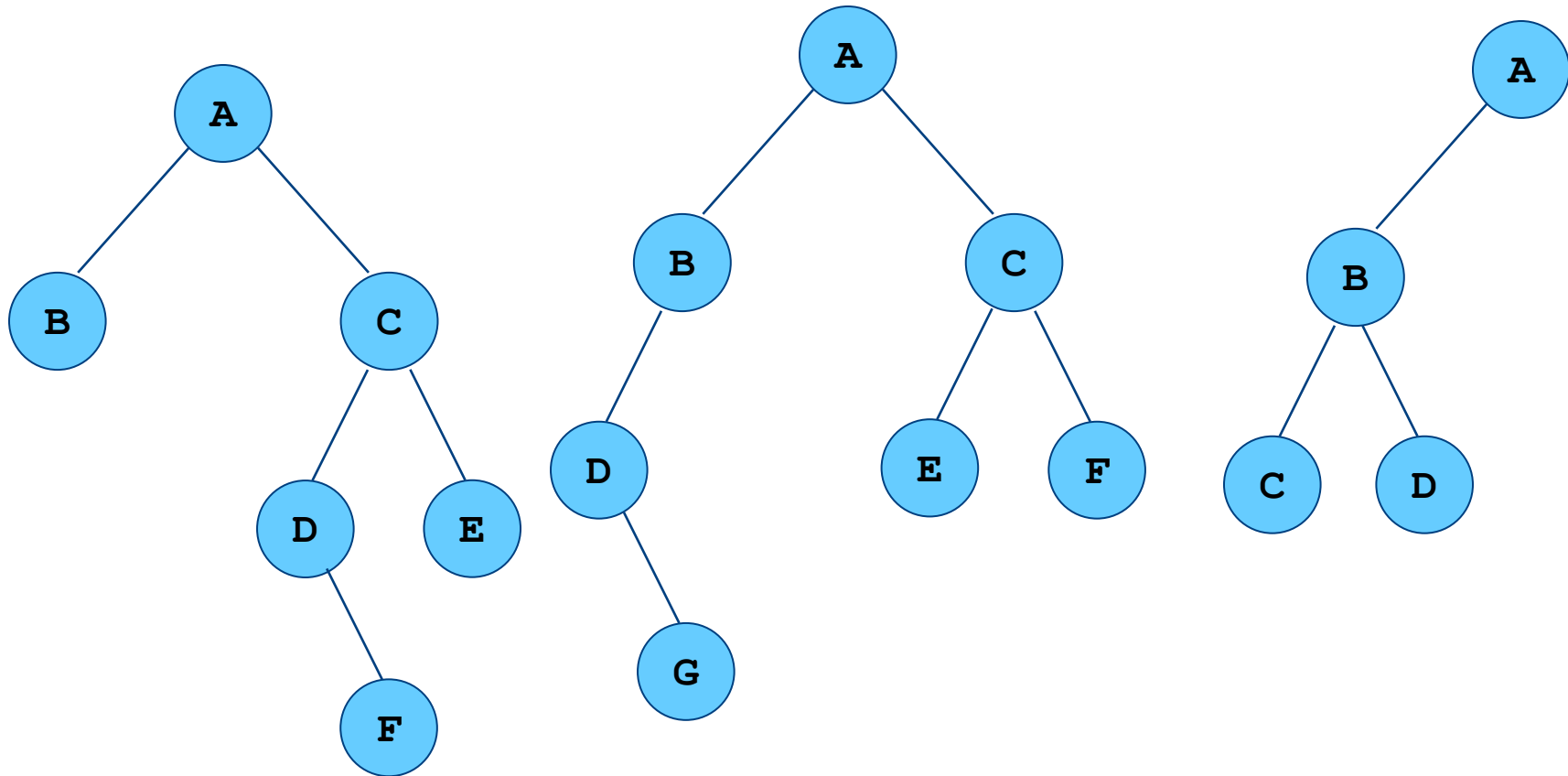


-
- A **balanced** binary tree is one where
 - No leaf is more than a certain amount farther from the root than any other leaf, this is sometimes stated more specifically as:
 - The height of any node's right subtree is at most one different from the height of its left subtree
 - Note that complete and full binary trees are balanced binary trees
-

Balanced Binary Trees



Unbalanced Binary Trees



-
- If T is a balanced binary tree with n nodes, its height is less than $\log n + 1$.
-

Binary Tree Traversals

- A traversal algorithm for a binary tree visits each node in the tree
 - and, typically, does something while visiting each node!
 - Traversal algorithms are naturally recursive
 - There are three traversal methods
 - Inorder
 - Preorder
 - Postorder
-

preOrder Traversal Algorithm

// preOrder traversal algorithm

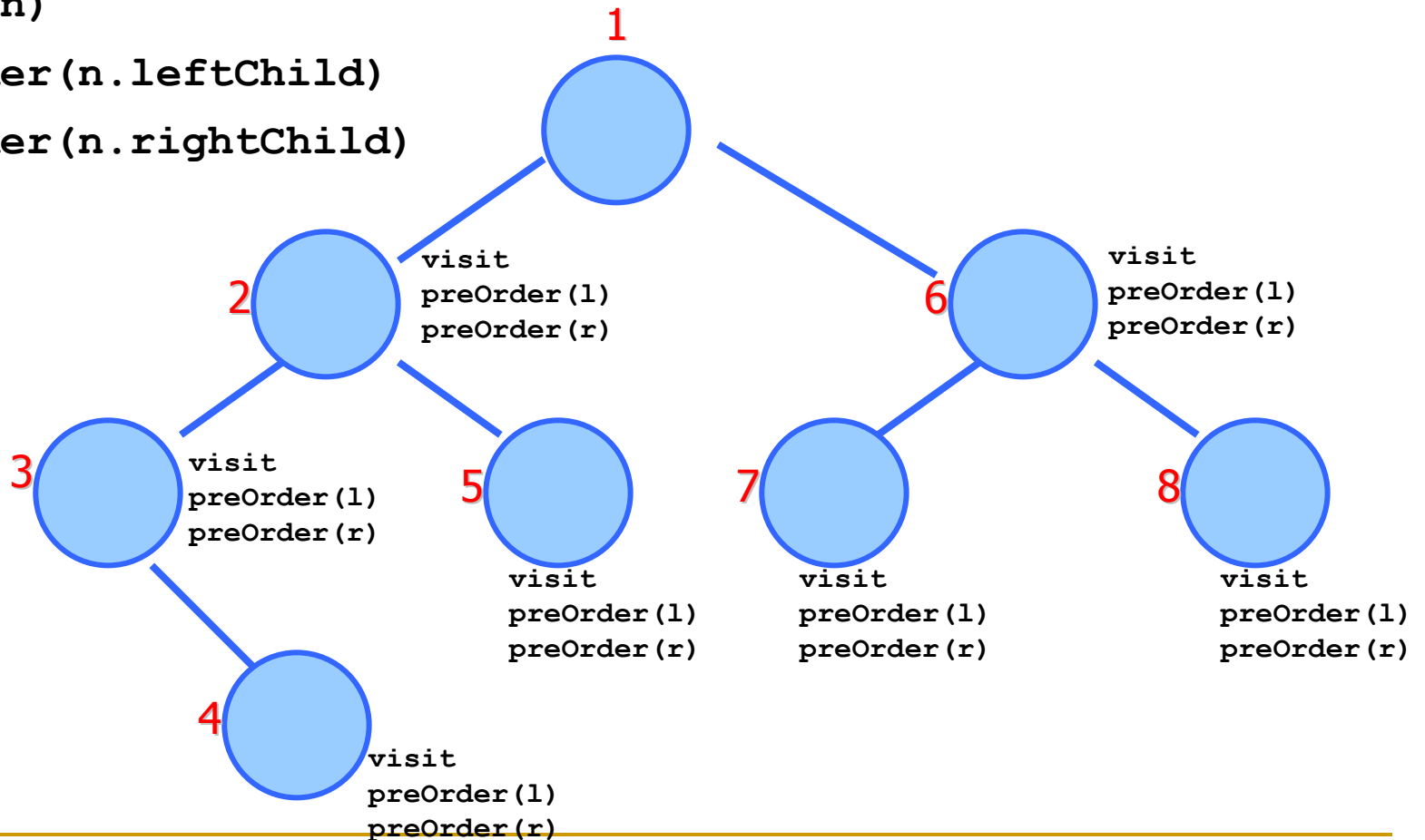
```
preOrder(TreeNode<T> n) {  
    if (n != null) {  
        visit(n);  
        preOrder(n.getLeft());  
        preOrder(n.getRight());  
    }  
}
```

PreOrder Traversal

`visit(n)`

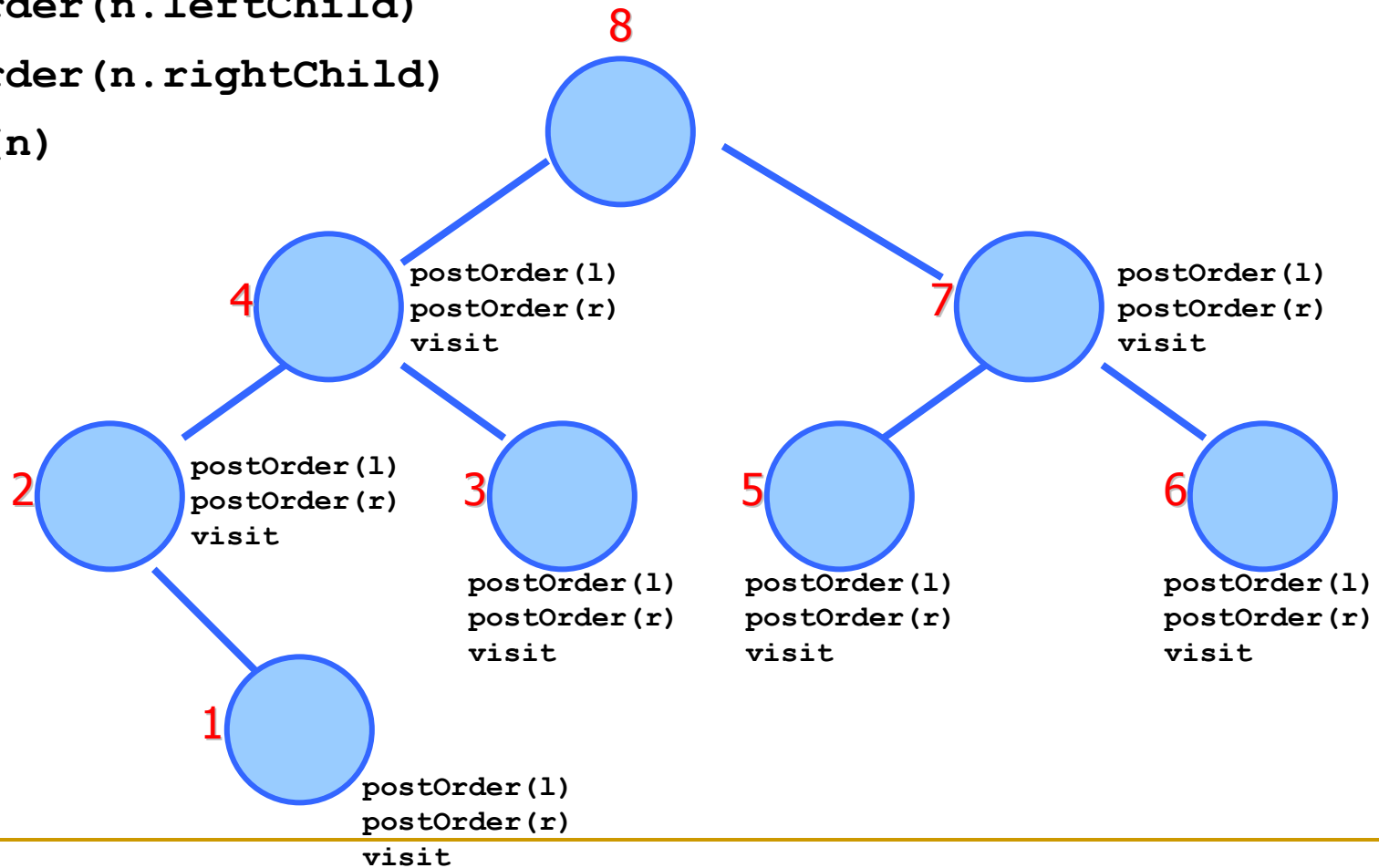
`preOrder(n.leftChild)`

`preOrder(n.rightChild)`



PostOrder Traversal

```
postOrder(n.leftChild)
postOrder(n.rightChild)
visit(n)
```



InOrder Traversal Algorithm

```
// InOrder traversal algorithm
inOrder(TreeNode<T> n) {
    if (n != null) {
        inOrder(n.getLeft());
        visit(n);
        inOrder(n.getRight());
    }
}
```

Examples

- Iterative version of in-order traversal
 - Option 1: using Stack
 - Option 2: with references to parents in TreeNodes
 - Iterative version of height() method
-

Iterative implementation of inOrder

```
public void inOrderNonRecursive( TreeNode root){
    Stack visitStack = new Stack();
    TreeNode curr=root;
    while ( true ){
        if ( curr != null){
            visitStack.push(curr);
            curr = curr.getLeft();
        }
        else {
            if (!visitStack.isEmpty()){
                curr = visitStack.pop();
                System.out.println (curr.getItem());
                curr = curr.getRight();
            }
            else
                break;
        }
    }
}
```

Binary Tree Implementation

- The binary tree ADT can be implemented using a number of data structures
 - Reference structures (similar to linked lists), as we have seen
 - Arrays – either simulating references or complete binary trees allow for a special very memory efficient array representation (called heaps)
-

Possible Representations of a Binary Tree

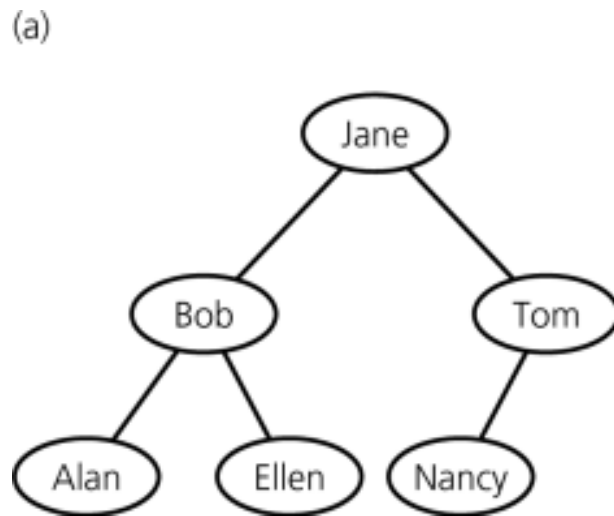


Figure 11-11a

a) A binary tree of names

(b)

tree				
	item	leftChild	rightChild	root
0	Jane	1	2	<div>0</div>
1	Bob	3	4	free
2	Tom	5	-1	<div>6</div>
3	Alan	-1	-1	
4	Ellen	-1	-1	
5	Nancy	-1	-1	
6	?	-1	7	} Free list
7	?	-1	8	
8	?	-1	9	
•	•	•	•	
•	•	•	•	
•	•	•	•	
•	•	•	•	

Figure 11-11b

b) its array-based implementations

Array based implementation of BT.

```
public class TreeNode<T> {  
    private T item; // data item in the tree  
    private int leftChild; // index to left child  
    private int rightChild; // index to right child  
  
    // constructors and methods appear here  
} // end TreeNode  
  
public class BinaryTreeArrayBased<T> {  
    protected final int MAX_NODES = 100;  
    protected ArrayList<TreeNode<T>> tree;  
    protected int root; // index of tree's root  
    protected int free; // index of next unused array  
                        // location  
    // constructors and methods  
} // end BinaryTreeArrayBased
```

Possible Representations of a Binary Tree

- An array-based representation of a complete tree
 - If the binary tree is complete and remains complete
 - A memory-efficient array-based implementation can be used
 - In this implementation the reference to the children of a node does not need to be saved in the node, rather it is computed from the index of the node.
-

Possible Representations of a Binary Tree

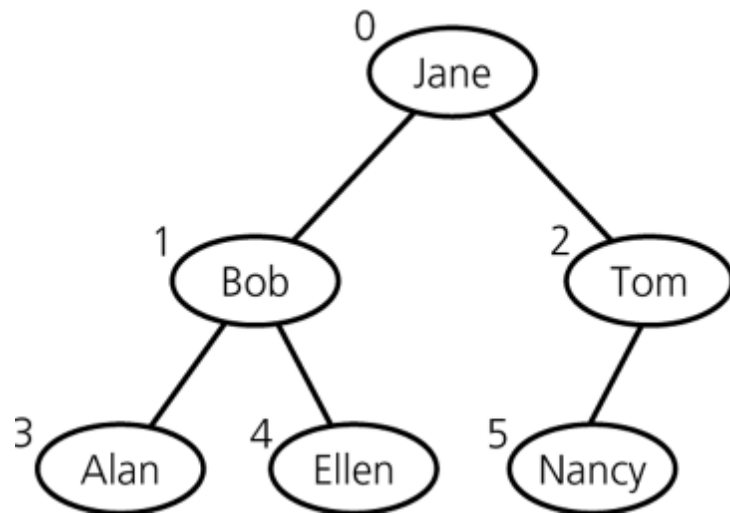


Figure 11-12

Level-by-level numbering of a complete binary tree

0	Jane
1	Bob
2	Tom
3	Alan
4	Ellen
5	Nancy
6	
7	

Figure 11-13

An array-based implementation of the complete binary tree in Figure 10-12

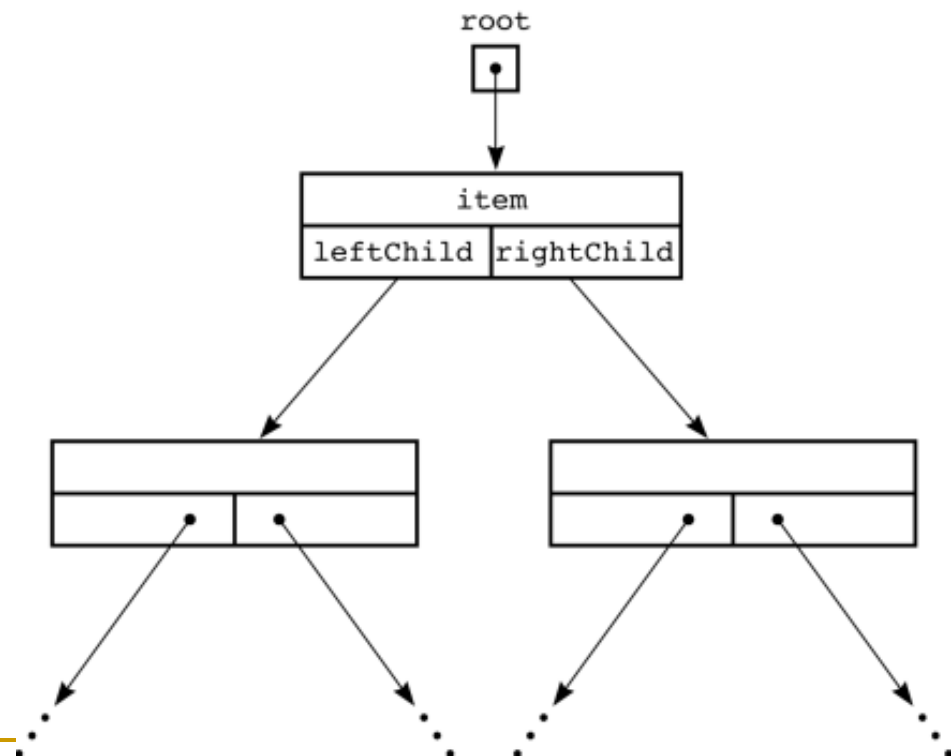
-
- In this memory efficient representation $\text{tree}[i]$ contains the node numbered i ,
 - $\text{tree}[2*i+1]$, $\text{tree}[2*i+2]$ and $\text{tree}[(i-1)/2]$ contain the left child, right child and the parent of node i , respectively.
-

Possible Representations of a Binary Tree

- A reference-based representation
 - Java references can be used to link the nodes in the tree

Figure 11-14

A reference-based
implementation of a binary
tree



```
public class TreeNode<T> {  
    private T item; // data item in the tree  
    private TreeNode<T> leftChild; // index to left child  
    private TreeNode<T> rightChild; // index to right child  
  
    // constructors and methods appear here  
} // end TreeNode  
  
public class BinaryTreeReferenceBased<T> {  
    protected TreeNode<T> root; // index of tree's root  
  
    // constructors and methods  
} // end BinaryTreeReferenceBased
```

-
- We will look at 3 applications of binary trees
 - Binary search trees (references)
 - Red-black trees (references)
 - Heaps (arrays)
-

Problem: Design a data structure for storing data with keys

- Consider maintaining data in some manner
 - The data is to be frequently searched on the **search key** e.g. a dictionary, records in database
 - Possible solutions might be:
 - A sorted array (by the keys)
 - Access in $O(\log n)$ using binary search
 - Insertion and deletion in linear time –i.e $O(n)$
 - An sorted linked list
 - Access, insertion and deletion in linear time.
-

Dictionary Operations

- The data structure should be able to perform all these operations efficiently
 - Create an empty dictionary
 - Insert
 - Delete
 - Look up (by the key)
 - The insert, delete and look up operations should be performed in $O(\log n)$ time
 - Is it possible?
-

Data with keys

- For simplicity we will assume that keys are of type `long`, i.e., they can be compared with operators `<`, `>`, `<=`, `==`, etc.
- All items stored in a container will be derived from `KeyedItem`.

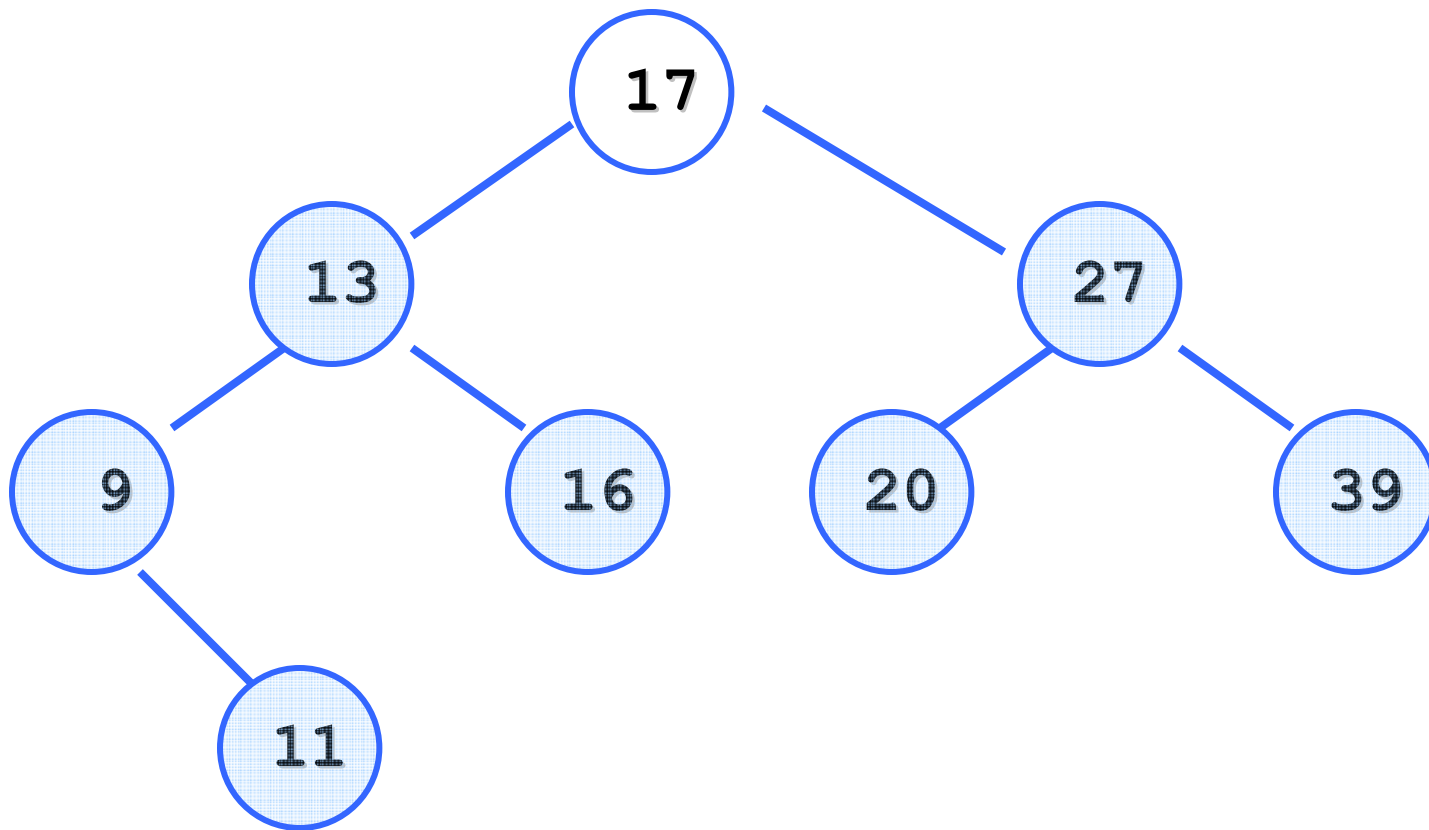
```
public class KeyedItem
{
    private long key;

    public KeyedItem(long k)
    {
        key=k;
    }
    public getKey() {
        return key;
    }
}
```

Binary Search Trees (BSTs)

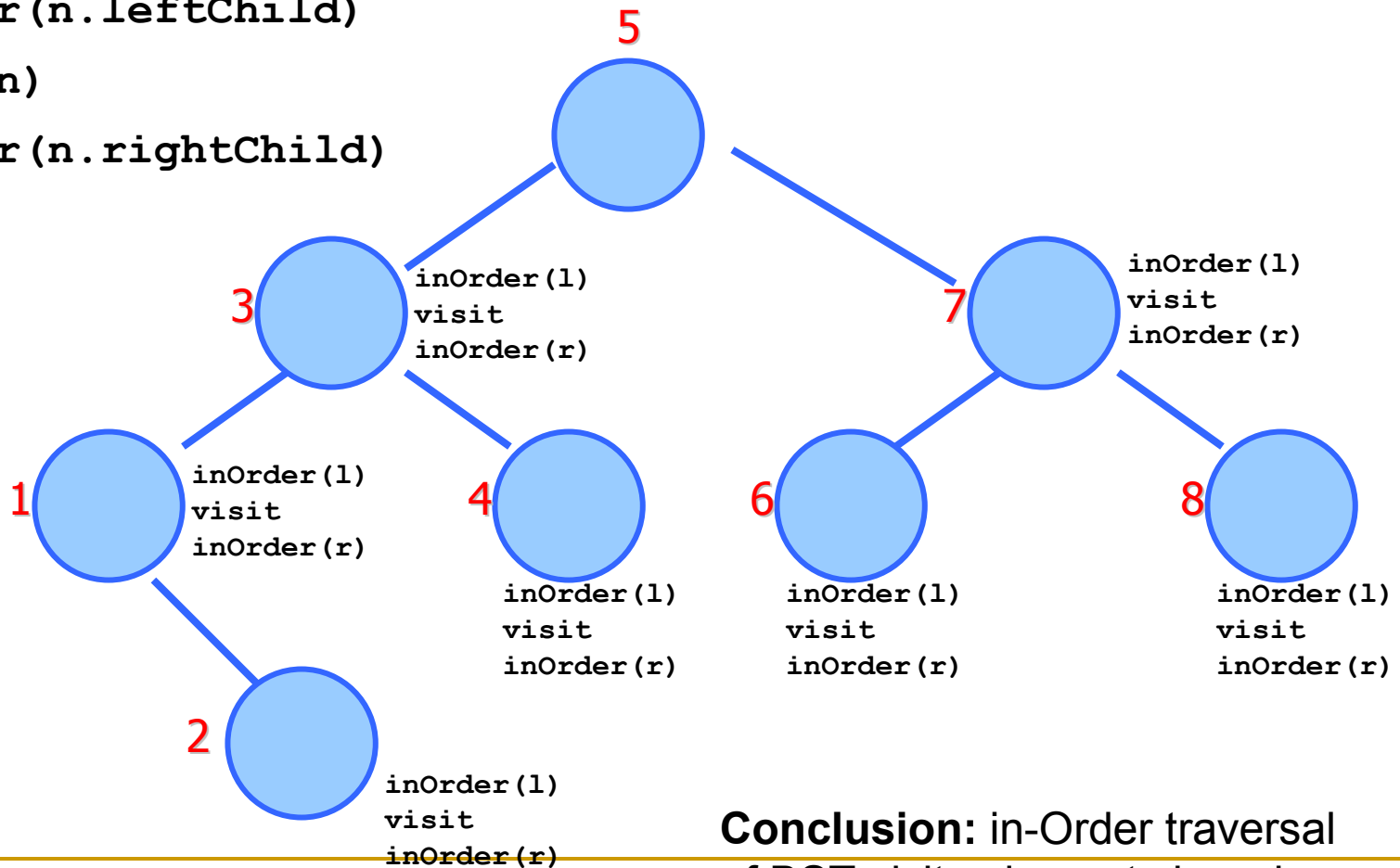
- A binary search tree is a binary tree with a special property
 - For all nodes v in the tree:
 - All the nodes in the **left subtree** of v contain items **less than equal to** the item in v and
 - All the nodes in the **right subtree** of v contain items **greater than or equal to** the item in v
-

BST Example



BST InOrder Traversal

```
inOrder(n.leftChild)
visit(n)
inOrder(n.rightChild)
```



Conclusion: in-Order traversal of BST visits elements in order.

BST Implementation

- Binary search trees can be implemented using a reference structure
- Tree nodes contain data and two references to nodes

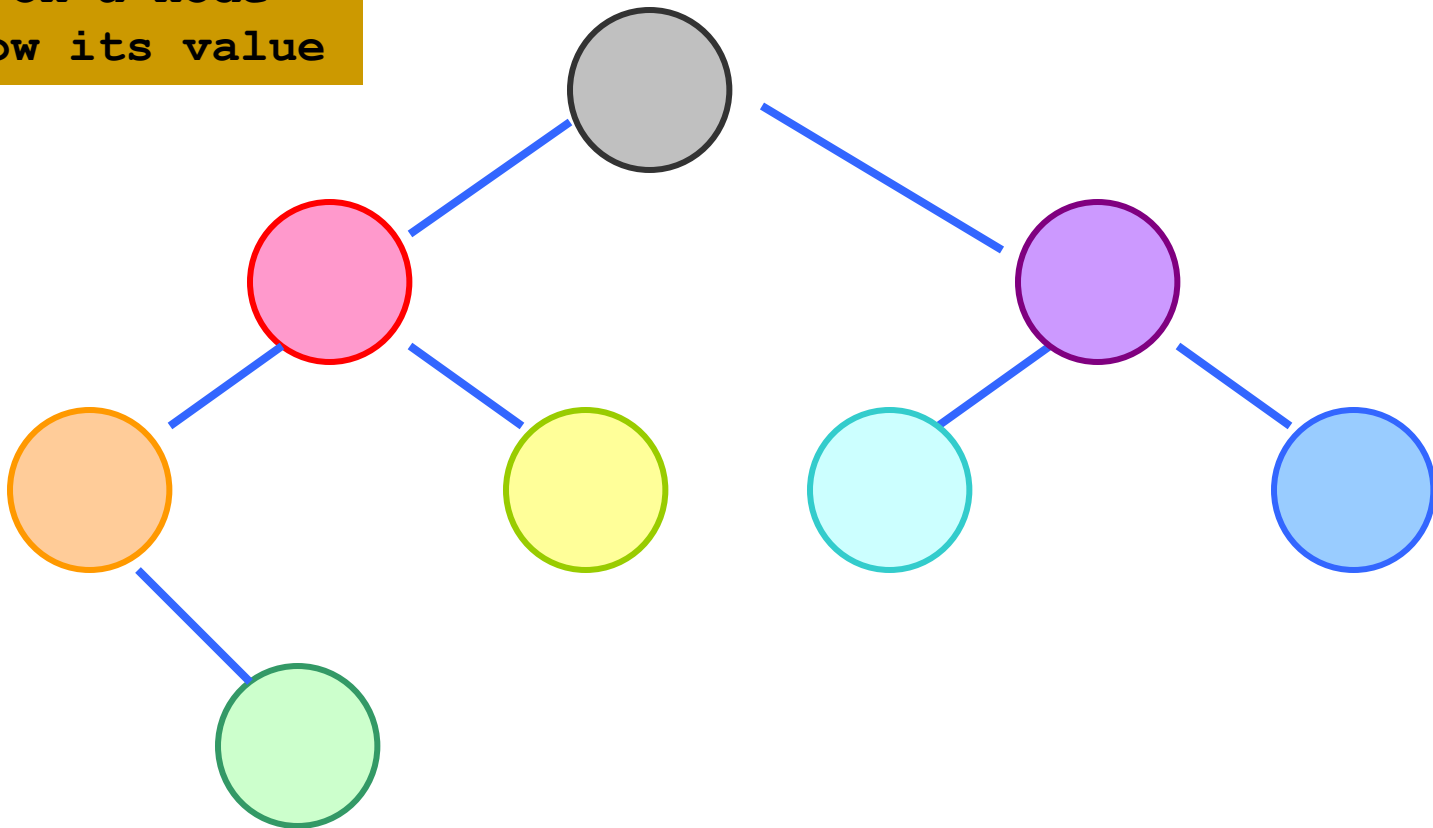
Node leftChild	Object data	Node rightChild
----------------	-------------	-----------------

BST Search

- To find a value in a BST search from the root node:
 - If the target is equal to the value in the node return data.
 - If the target is less than the value in the node search its left subtree
 - If the target is greater than the value in the node search its right subtree
 - If null value is reached, return null (“not found”).
 - How many comparisons?
 - One for each node on the path
 - Worst case: height of the tree
-

BST Search Example

click on a node
to show its value



Search algorithm (recursive)

```
T retrieveItem(TreeNode<T extends KeyedItem> n, long searchKey)
// returns a node containing the item with the key searchKey
// or null if not found
{
    if (n == null) {
        return null;
    }
    else {
        if (searchKey == n.getItem().getKey()) {
            // item is in the root of some subtree
            return n.getItem();
        }
        else if (searchKey < n.getItem().getKey()) {
            // search the left subtree
            return retrieveItem(n.getLeft(), searchKey);
        }
        else { // search the right subtree
            return retrieveItem(n.getRight(), searchKey);
        } // end if
    } // end if
} // end retrieveItem
```

BST Insertion

- The BST property must hold **after** insertion
 - Therefore the new node must be inserted in the correct position
 - This position is found by performing a search
 - If the search ends at the (null) left child of a node make its left child refer to the new node
 - If the search ends at the (null) right child of a node make its right child refer to the new node
 - The cost is about the same as the cost for the search algorithm, $O(\text{height})$
-

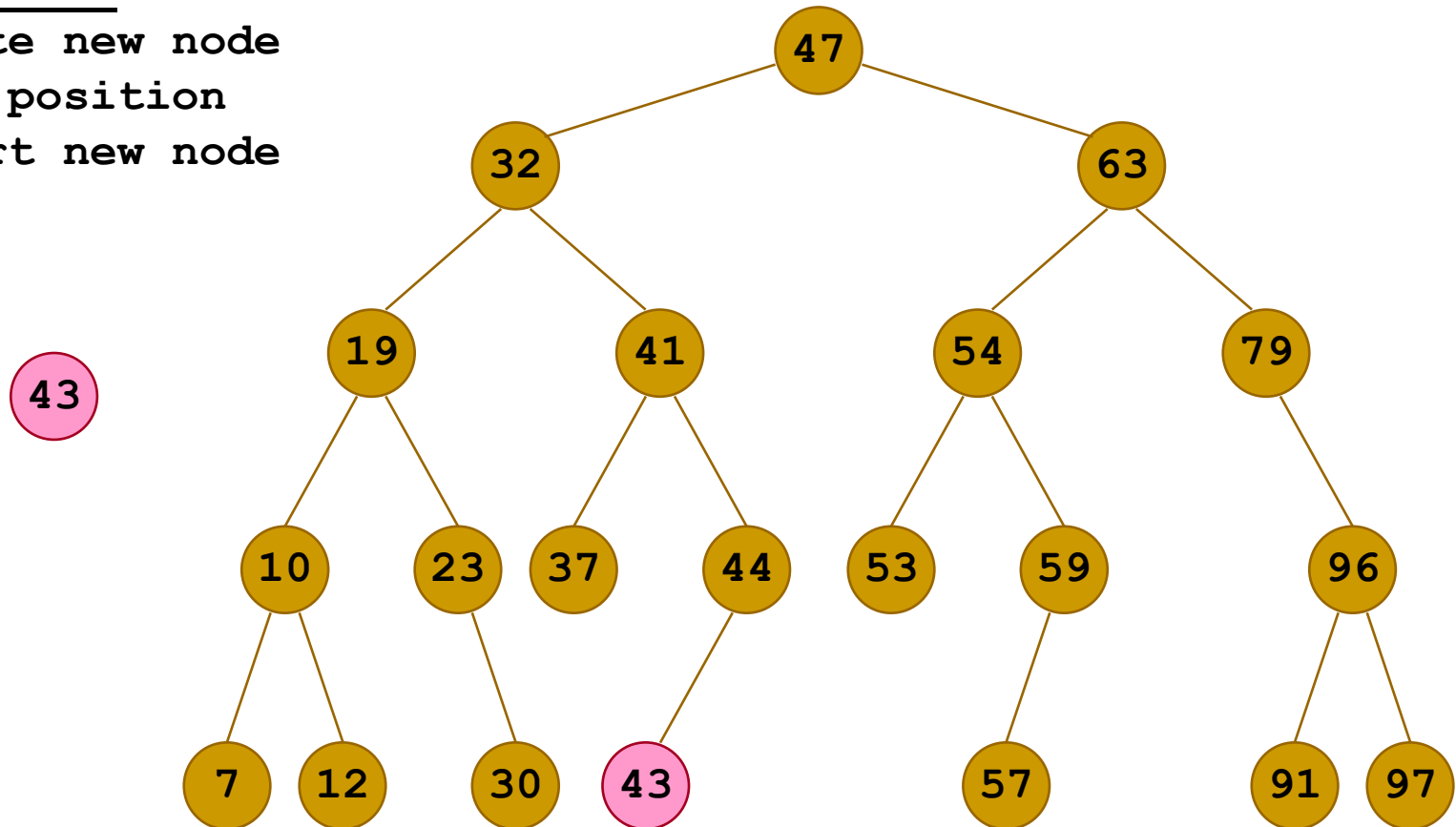
BST Insertion Example

insert 43

create new node

find position

insert new node



Insertion algorithm (recursive)

```
TreeNode<T> insertItem(TreeNode<T> n, T newItem)
// returns a reference to the new root of the subtree rooted in n
{
    TreeNode<T> newSubtree;
    if (n == null) {
        // position of insertion found; insert after leaf
        // create a new node
        n = new TreeNode<T>(newItem, null, null);
        return n;
    } // end if

    // search for the insertion position
    if (newItem.getKey() < n.getItem().getKey()) {
        // search the left subtree
        newSubtree = insertItem(n.getLeft(), newItem);
        n.setLeft(newSubtree);
        return n;
    }
    else { // search the right subtree
        newSubtree = insertItem(n.getRight(), newItem);
        n.setRight(newSubtree);
        return n;
    } // end if
} // end insertItem
```

BST Deletion

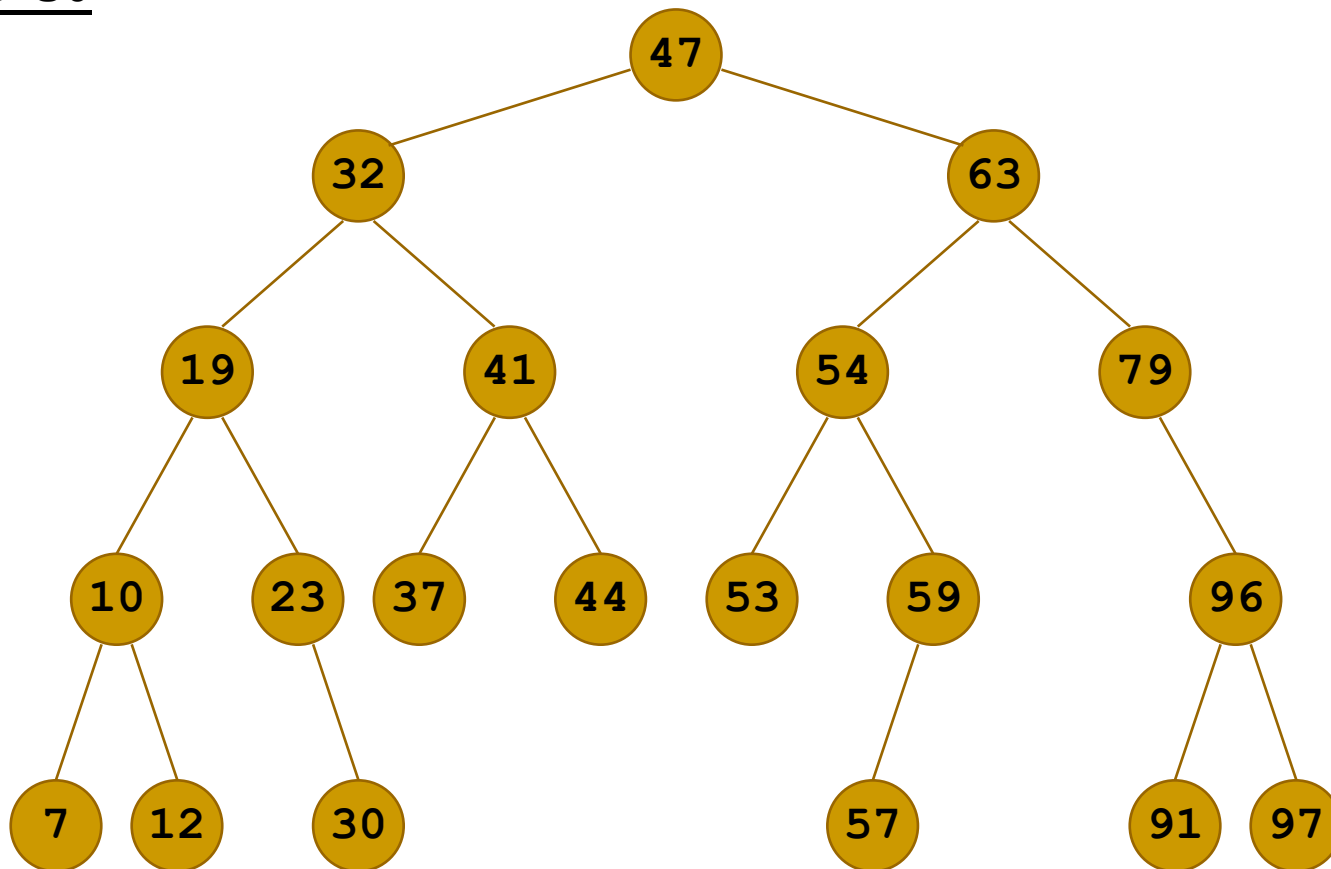
- After deleting a node the BST property must still hold
 - Deletion is not as straightforward as search or insertion
 - There are a number of different cases that have to be considered
 - The first step in deleting a node is to locate its parent and itself in the tree.
-

BST Deletion Cases

- The node to be deleted has no children
 - Remove it (assign null to its parent's reference)
 - The node to be deleted has one child
 - Replace the node with its subtree
 - The node to be deleted has two children
 - Replace the node with its predecessor = the right most node of its left subtree (or with its successor, the left most node of its right subtree)
 - If that node has a child (and it can have at most one child) attach that to the node's parent
-

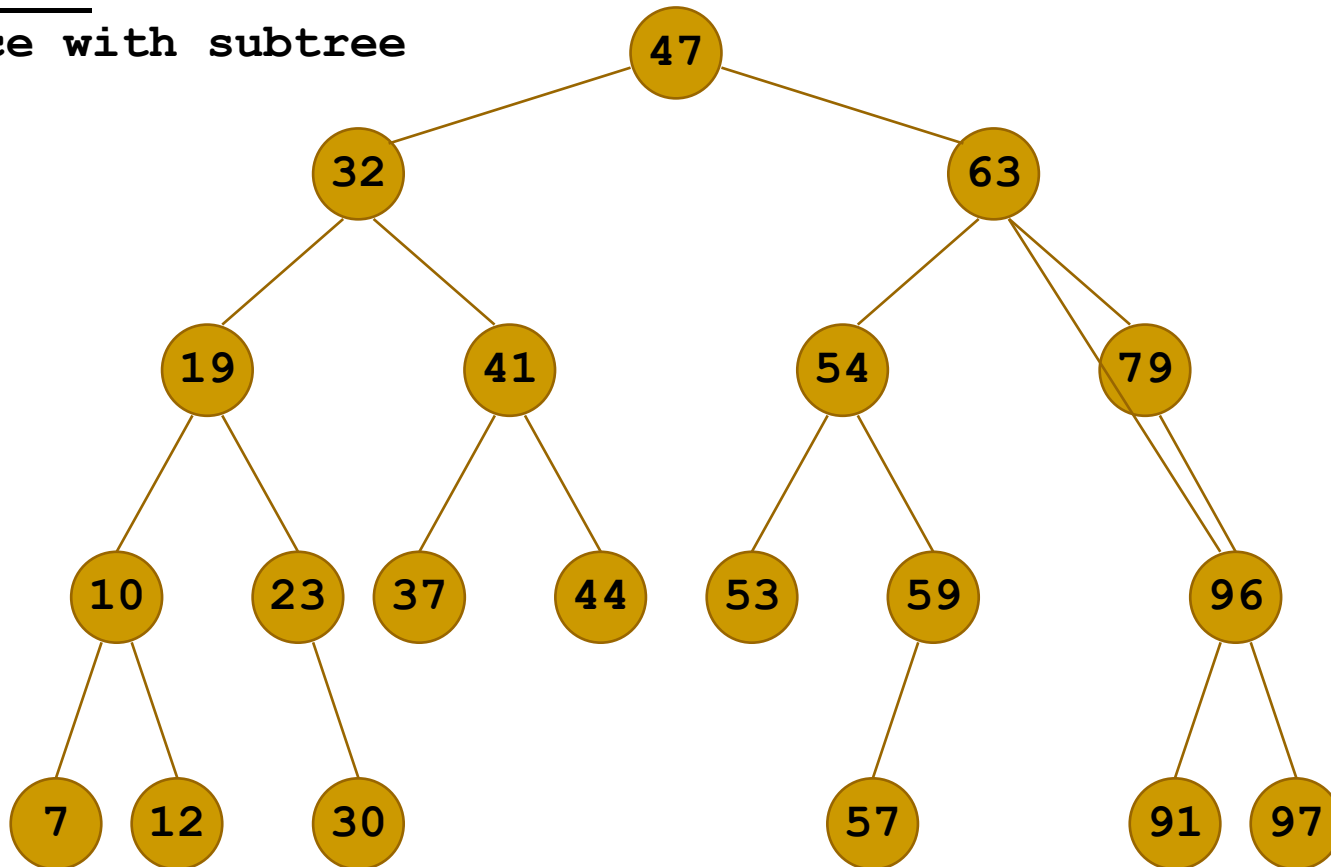
BST Deletion – target is a leaf

delete 30



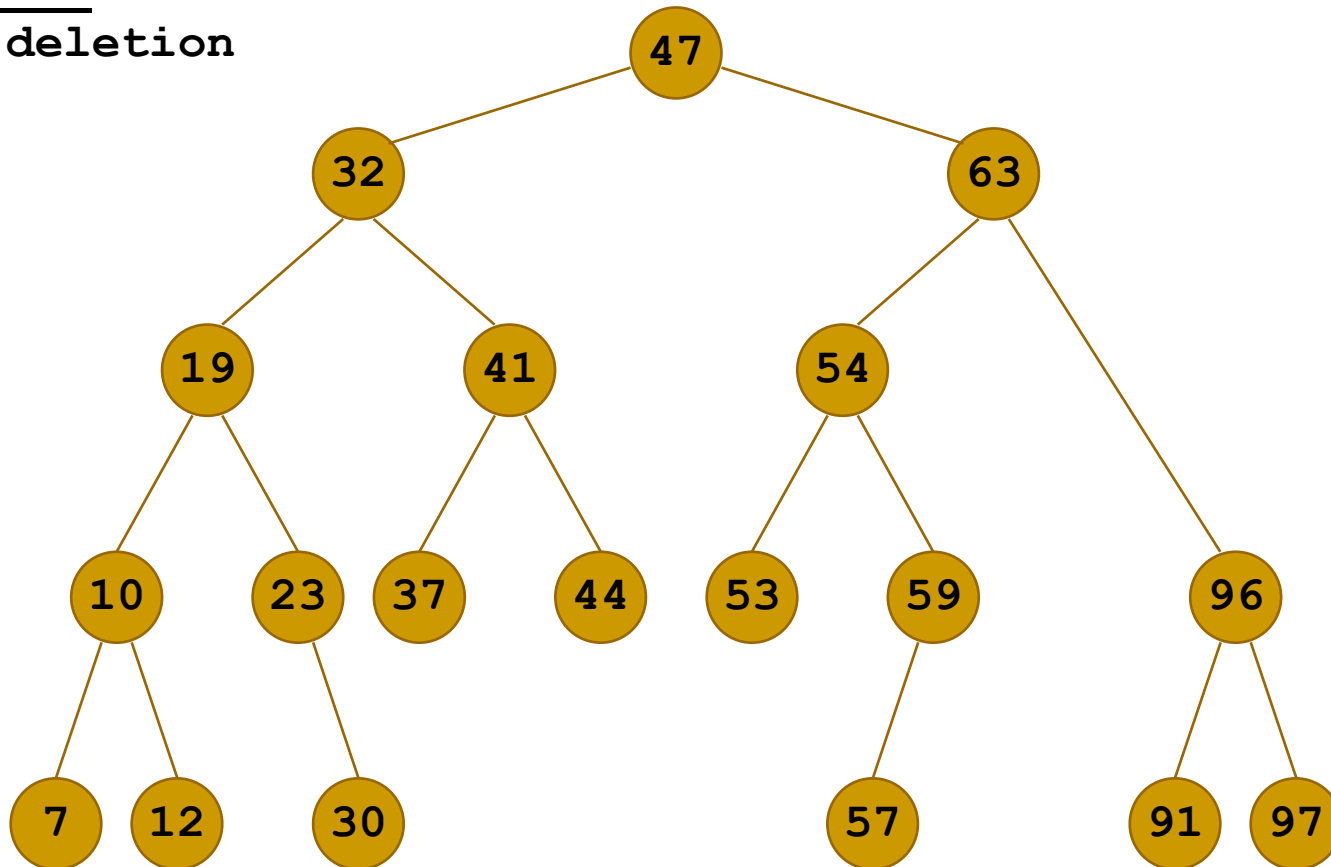
BST Deletion – target has one child

delete 79
replace with subtree



BST Deletion – target has one child

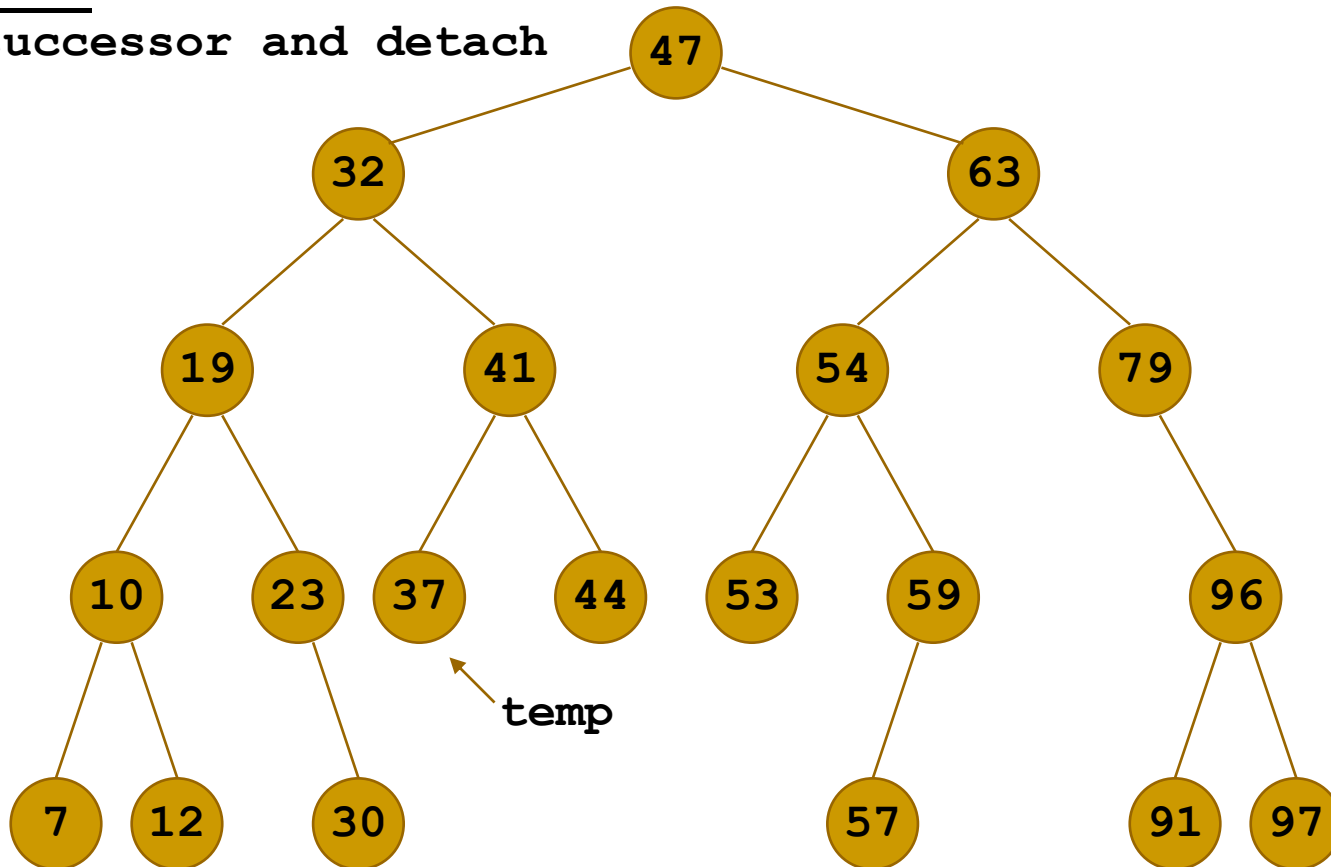
delete 79
after deletion



BST Deletion – target has 2 children

delete 32

find successor and detach

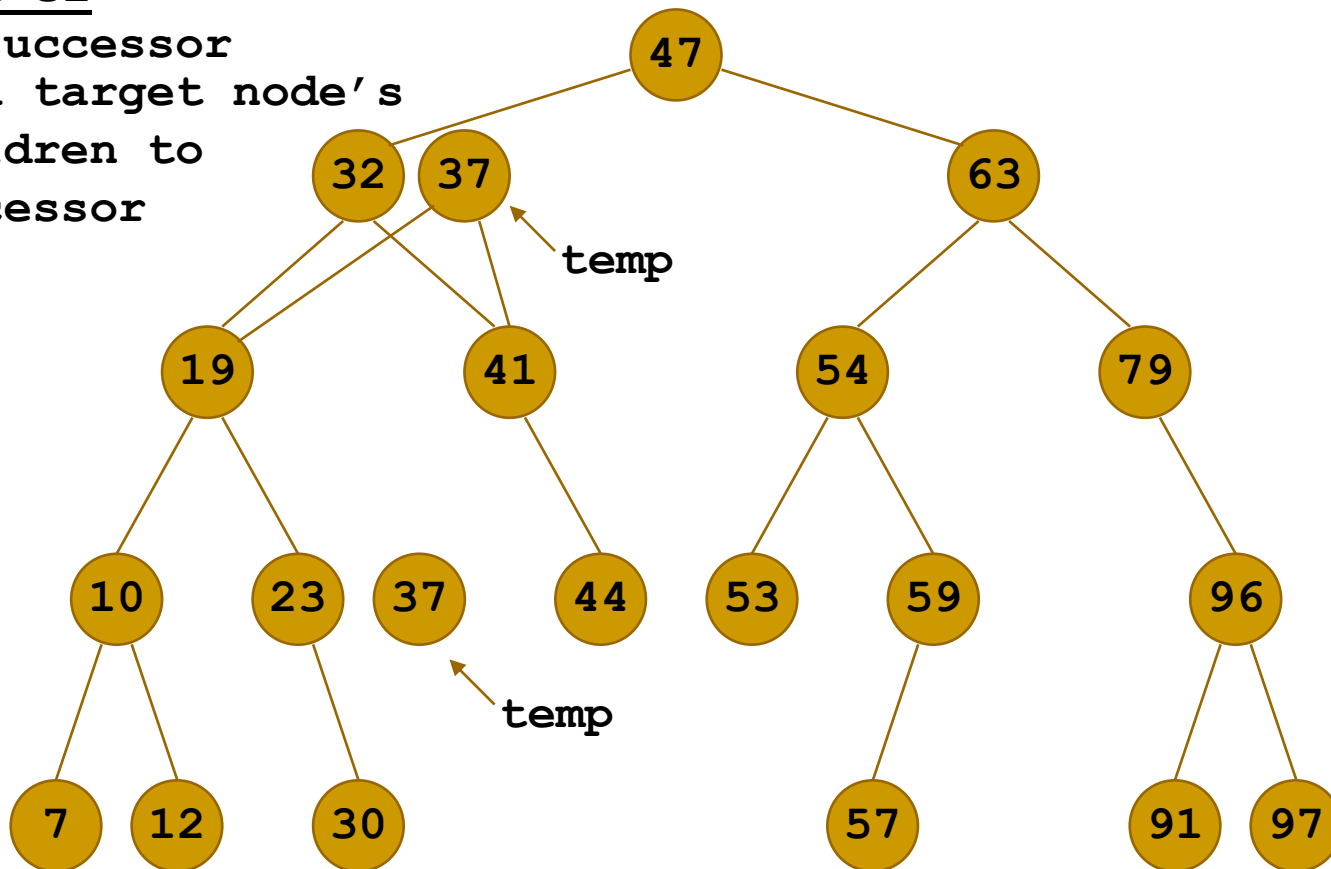


BST Deletion – target has 2 children

delete 32

find successor

attach target node's
children to
successor



BST Deletion – target has 2 children

delete 32

find successor

attach target node's

children to

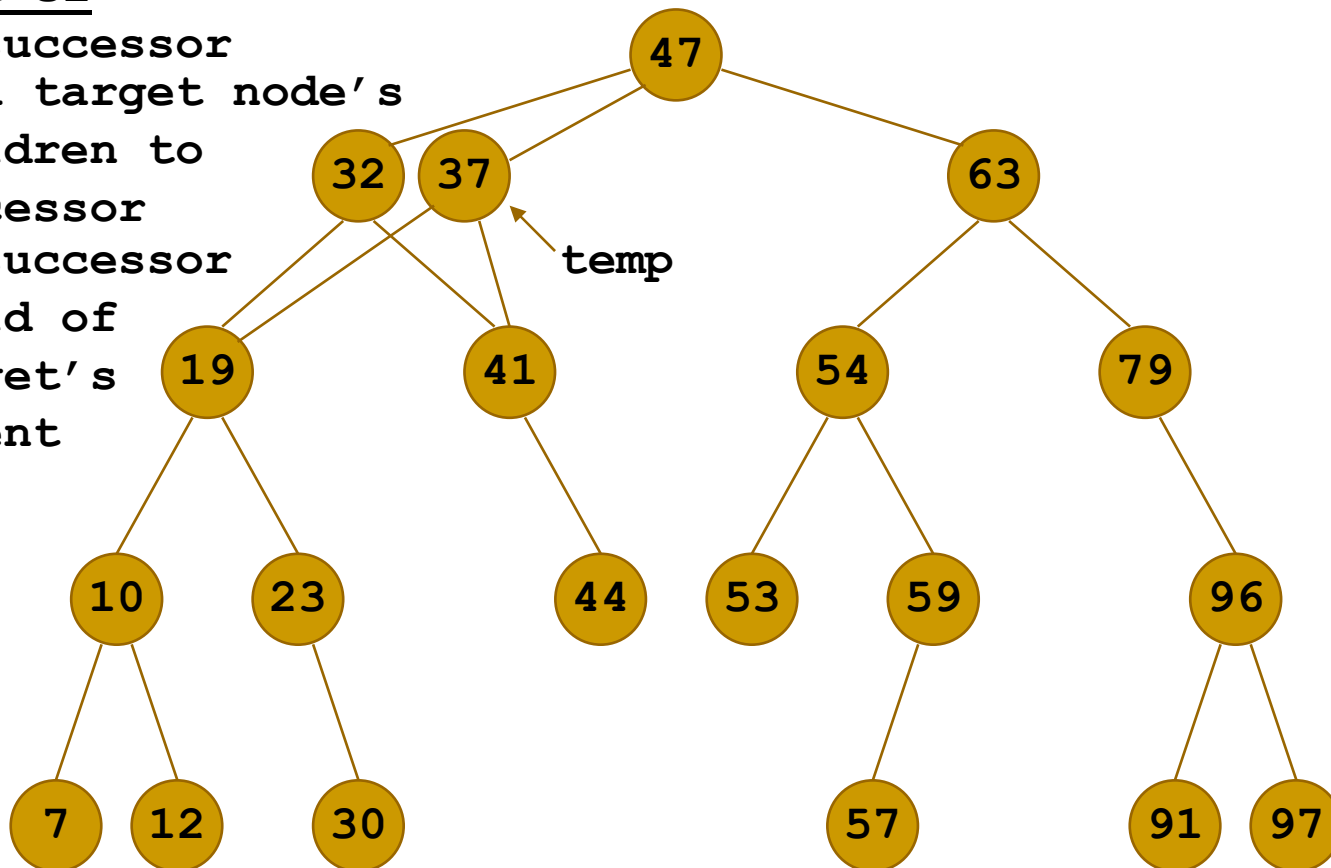
successor

make successor

child of

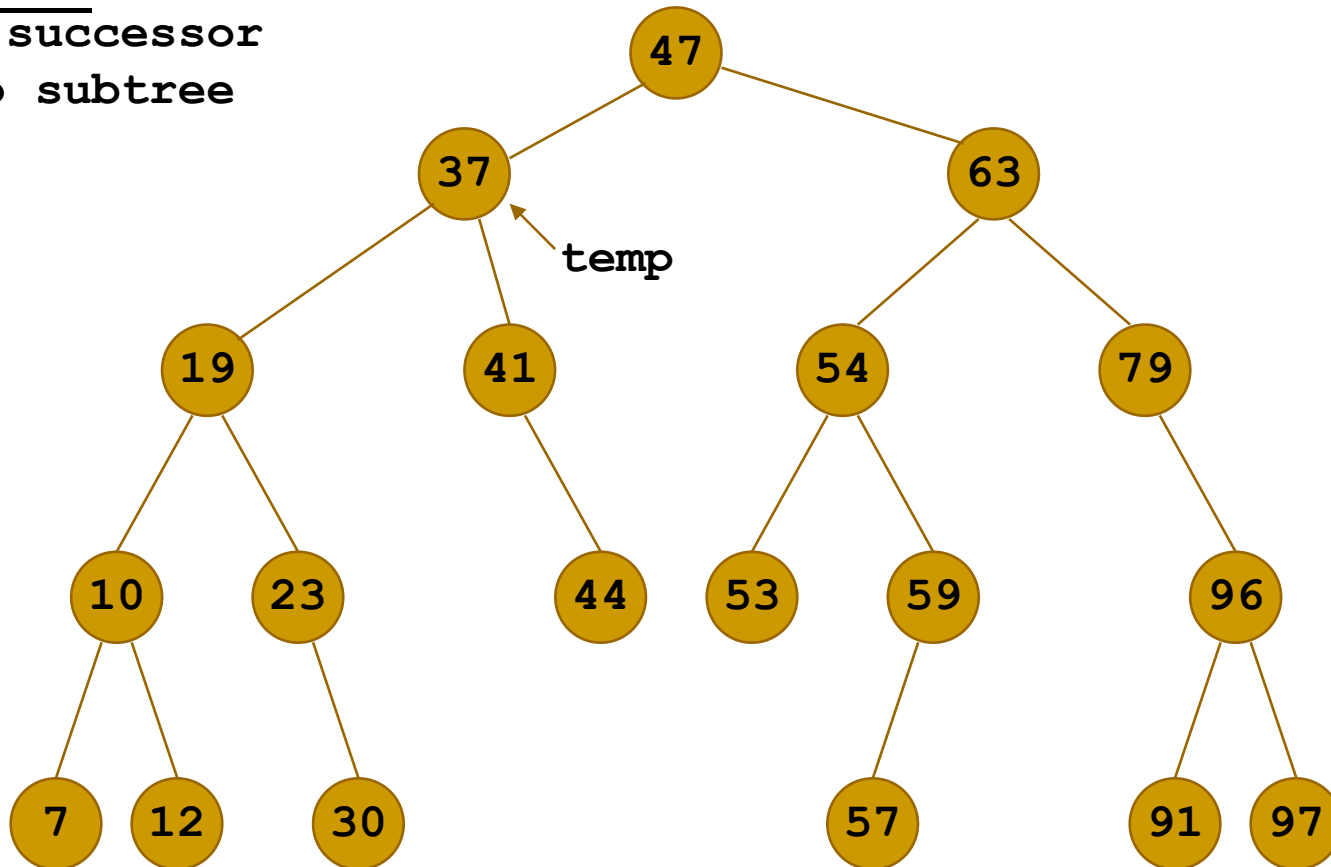
target's

parent



BST Deletion – target has 2 children

delete 32
note: successor
had no subtree

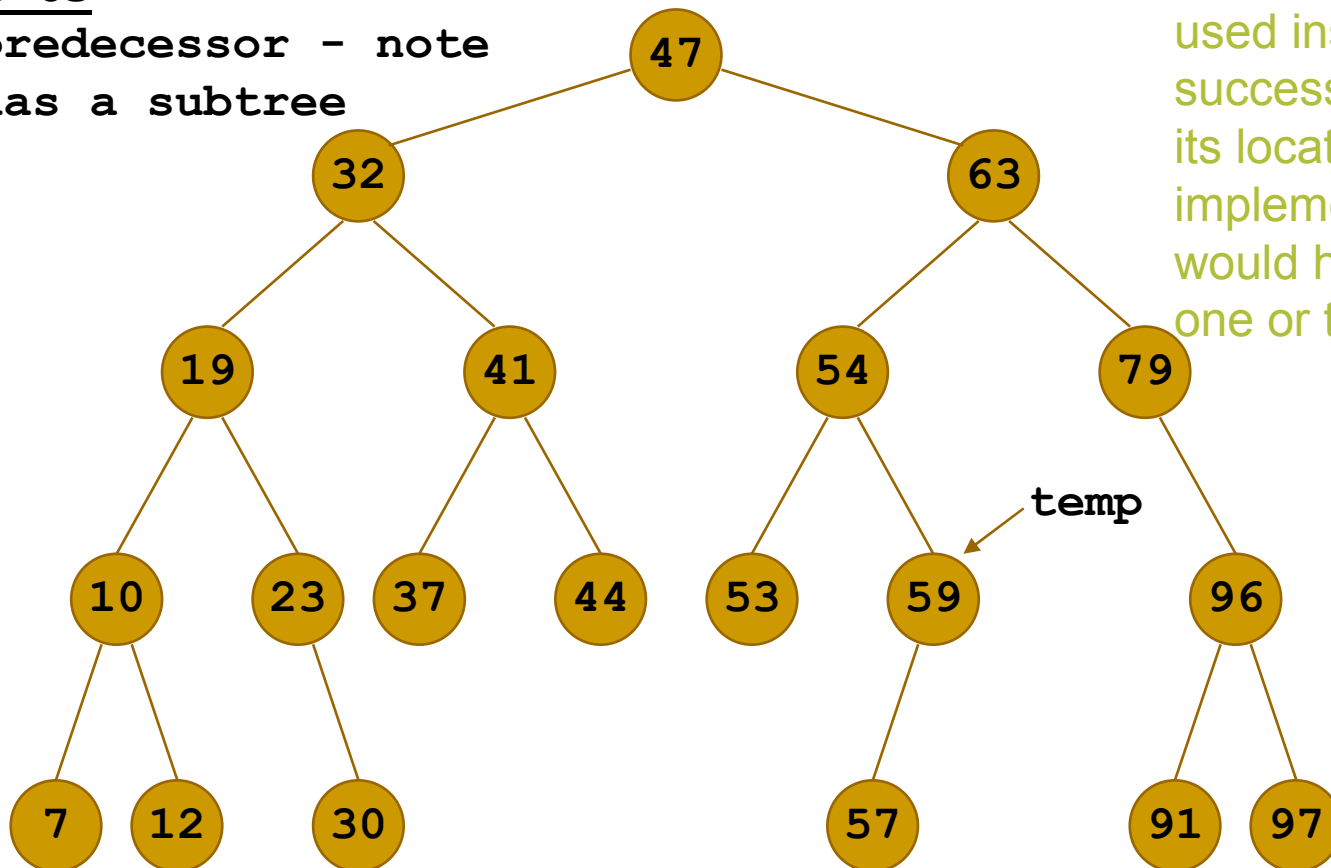


BST Deletion – target has 2 children

delete 63

find predecessor - note
it has a subtree

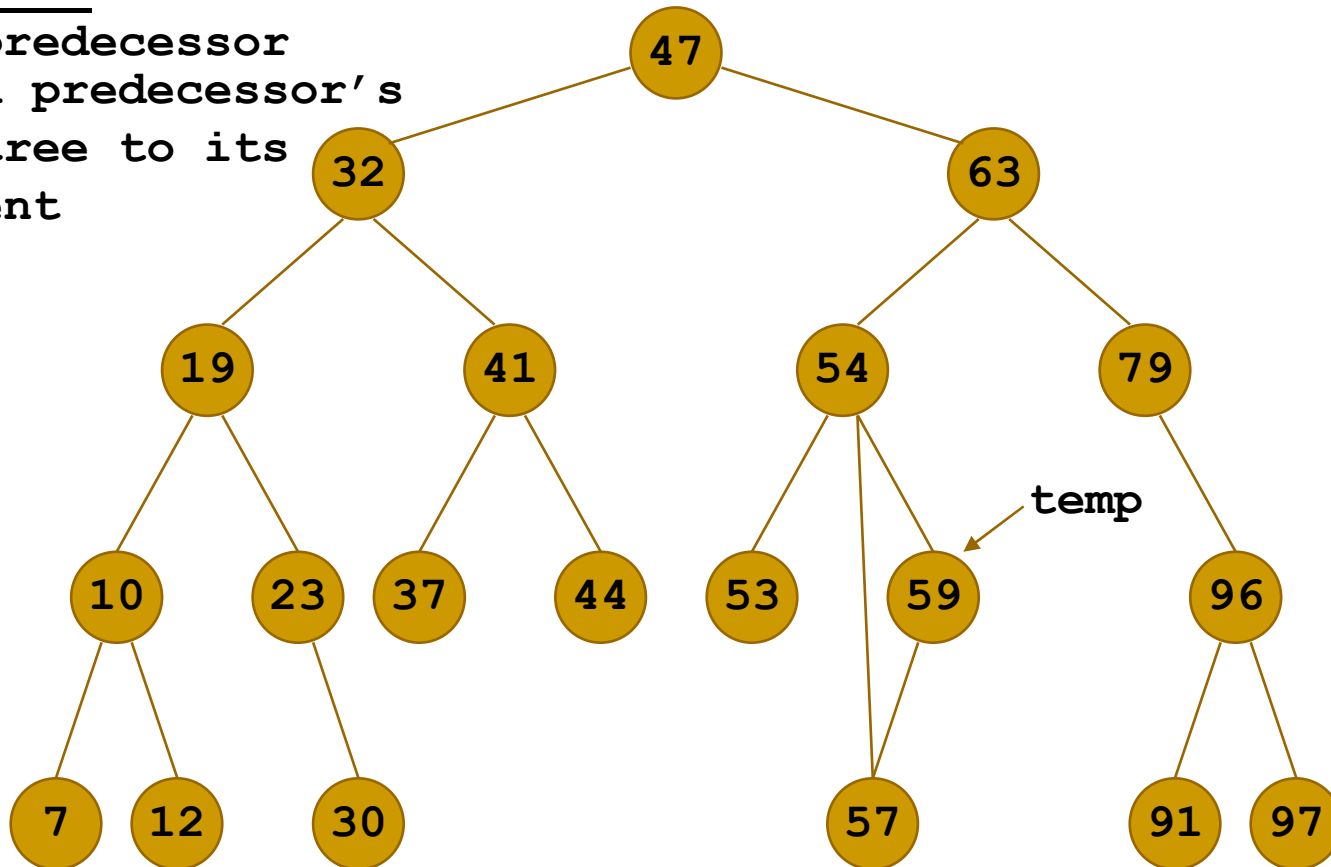
Note: predecessor
used instead of
successor to show
its location - an
implementation
would have to pick
one or the other



BST Deletion – target has 2 children

delete 63

find predecessor
attach predecessor's
subtree to its
parent



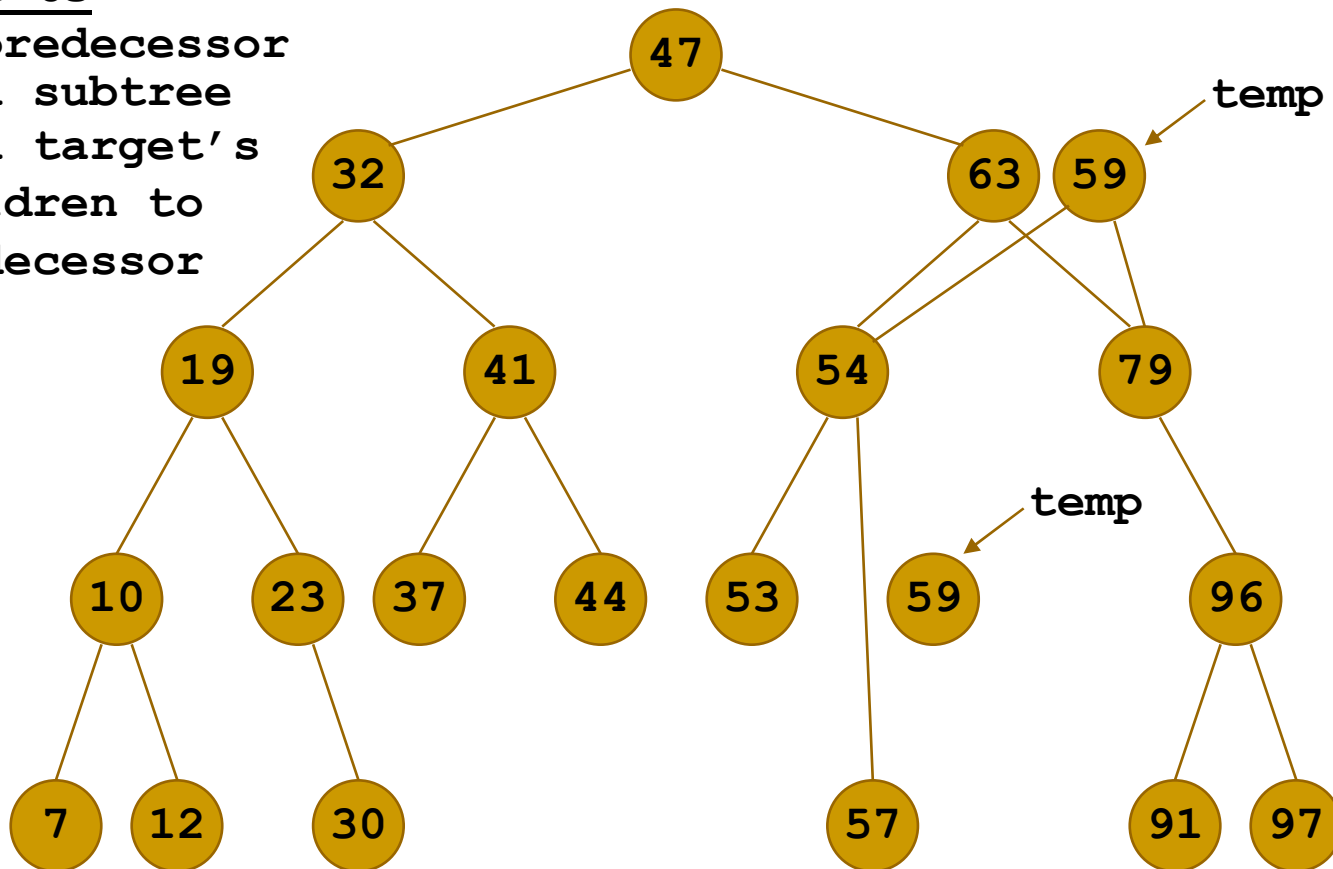
BST Deletion – target has 2 children

delete 63

find predecessor

attach subtree

attach target's
children to
predecessor



BST Deletion – target has 2 children

delete 63

find predecessor

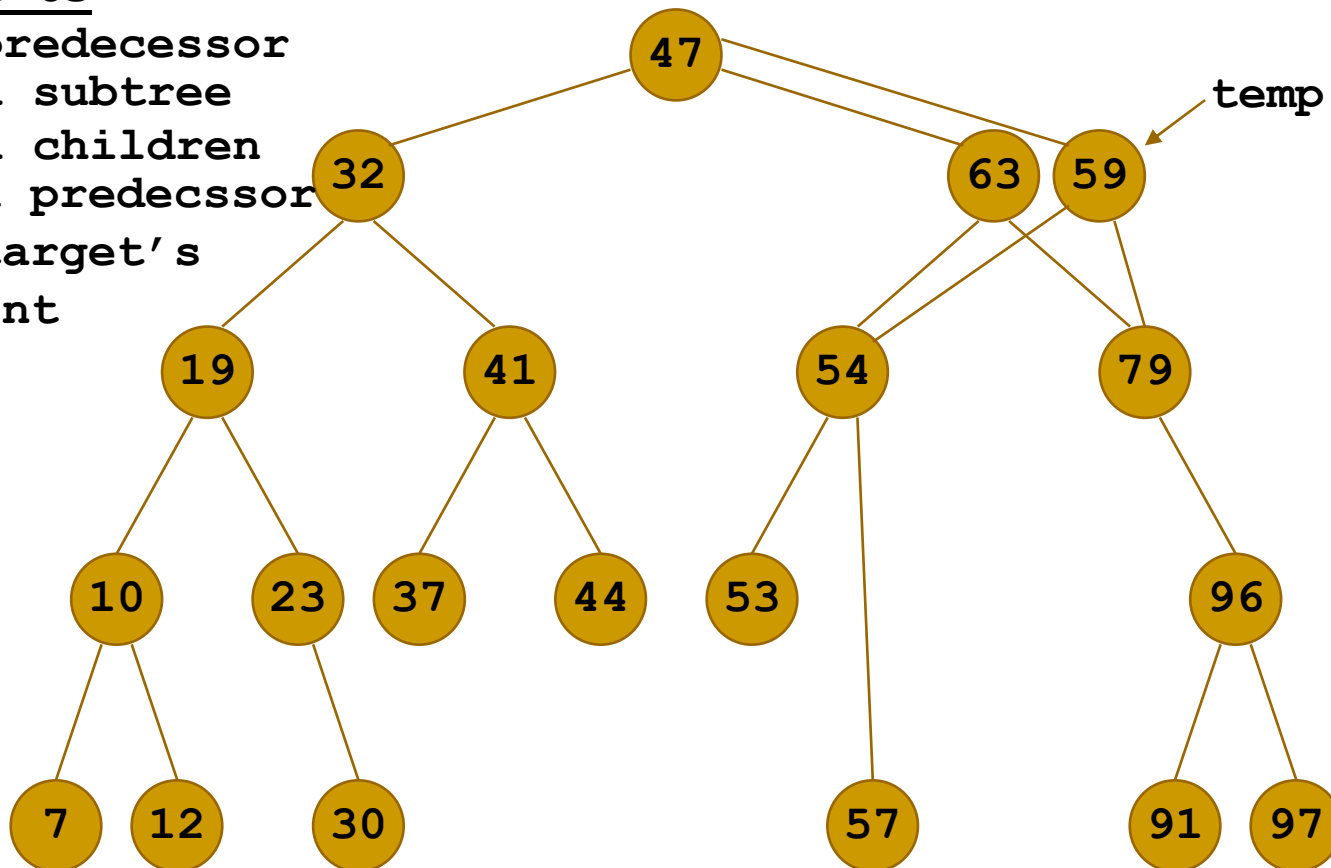
attach subtree

attach children

attach predecessor

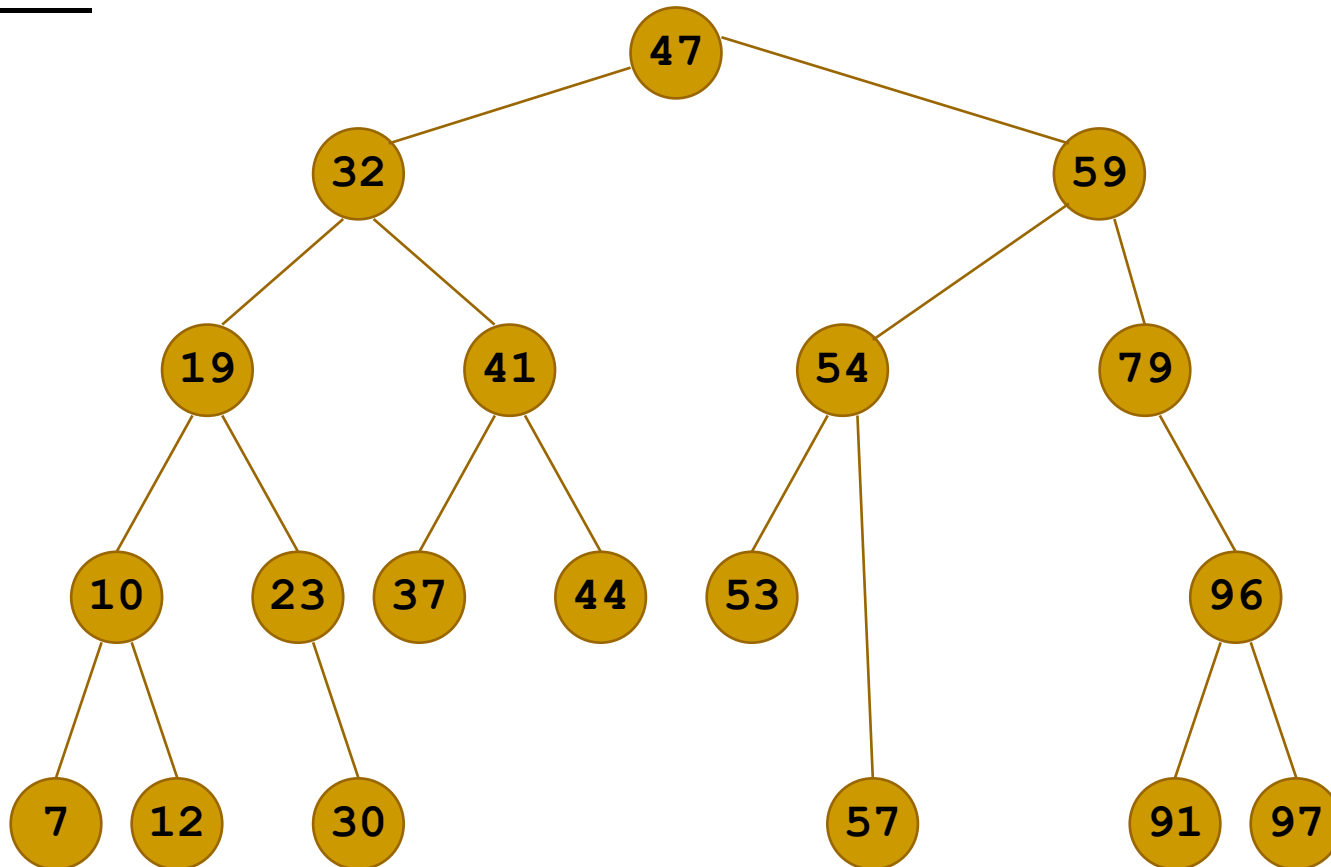
to target's

parent



BST Deletion – target has 2 children

delete 63



Deletion algorithm – Phase 1: Finding Node

```
TreeNode<T> deleteItem(TreeNode<T> n, long searchKey) {
    // Returns a reference to the new root.
    // Calls: deleteNode.
    TreeNode<T> newSubtree;
    if (n == null) {
        throw new TreeException("TreeException: Item not found");
    }
    else {
        if (searchKey==n.getItem().getKey()) {
            // item is in the root of some subtree
            n = deleteNode(n); // delete the node n
        }
        // else search for the item
        else if (searchKey<n.getItem().getKey()) {
            // search the left subtree
            newSubtree = deleteItem(n.getLeft(), searchKey);
            n.setLeft(newSubtree);
        }
        else { // search the right subtree
            newSubtree = deleteItem(n.getRight(), searchKey);
            n.setRight(newSubtree);
        } // end if
    } // end if
    return n;
} // end deleteItem
```

Deletion algorithm – Phase 2: Remove node or replace its with successor

```
TreeNode<T> deleteNode(TreeNode<T> n) {  
    // Returns a reference to a node which replaced n.  
    // Algorithm note: There are four cases to consider:  
    // 1. The n is a leaf.  
    // 2. The n has no left child.  
    // 3. The n has no right child.  
    // 4. The n has two children.  
    // Calls: findLeftmost and deleteLeftmost  
  
    // test for a leaf  
    if (n.getLeft() == null && n.getRight() == null)  
        return null;  
  
    // test for no left child  
    if (n.getLeft() == null)  
        return n.getRight();  
  
    // test for no right child  
    if (n.getRight() == null)  
        return n.getLeft();  
  
    // there are two children: retrieve and delete the inorder successor  
    T replacementItem = findLeftMost(n.getRight()).getItem();  
    n.setItem(replacementItem);  
    n.setRight(deleteLeftMost(n.getRight()));  
    return n;  
} // end deleteNode
```

Deletion algorithm – Phase 3: Remove successor

```
TreeNode<T> findLeftmost(TreeNode<T> n) {  
    if (n.getLeft() == null) {  
        return n;  
    }  
    else {  
        return findLeftmost(n.getLeft());  
    } // end if  
} // end findLeftmost
```

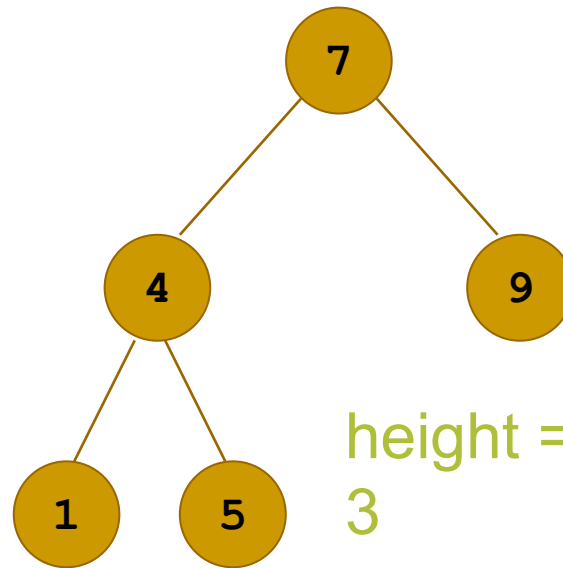
```
TreeNode<T> deleteLeftmost(TreeNode<T> n) {  
    // Returns a new root.  
    if (n.getLeft() == null) {  
        return n.getRight();  
    }  
    else {  
        n.setLeft(deleteLeftmost(n.getLeft()));  
        return n;  
    } // end if  
} // end deleteLeftmost
```

BST Efficiency

- The efficiency of BST operations depends on the height of the tree
 - All three operations (search, insert and delete) are $O(\text{height})$
 - If the tree is complete/full the height is $\lfloor \log(n) \rfloor + 1$
 - What if it isn't complete/full?
-

Height of a BST

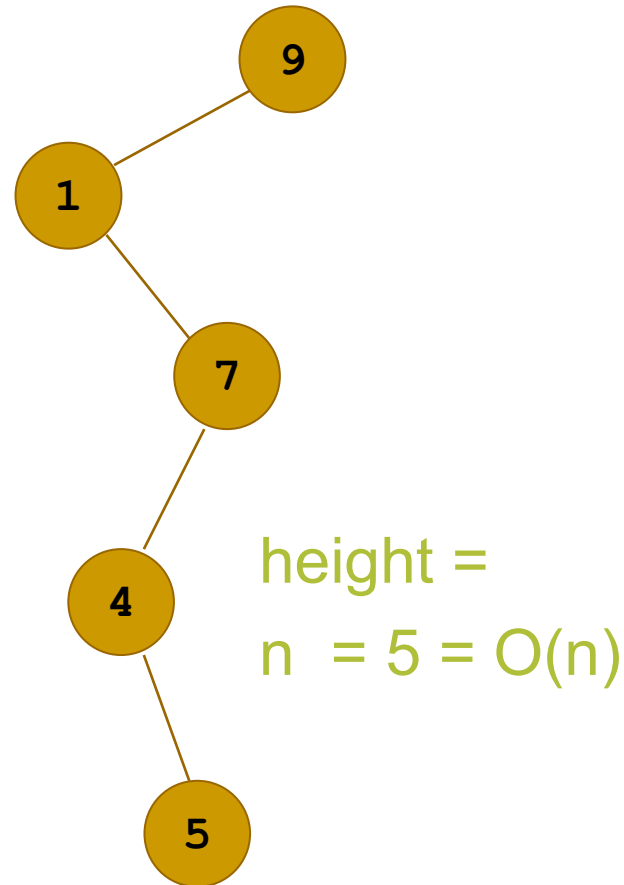
- Insert 7
- Insert 4
- Insert 1
- Insert 9
- Insert 5
- It's a complete tree!



$$\text{height} = \lfloor \log(5) \rfloor + 1 = 3$$

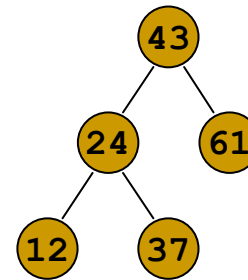
Height of a BST

- Insert 9
- Insert 1
- Insert 7
- Insert 4
- Insert 5
- It's a linked list!

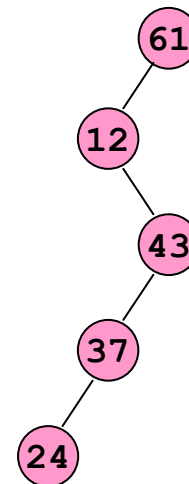


Binary Search Trees – Performance

- Items can be inserted in and removed and removed from BSTs in $O(\text{height})$ time
- So what is the height of a BST?
 - If the tree is complete it is $O(\log n)$ [best case]
 - If the tree is not balanced it may be $O(n)$ [worst case]



complete BST
height = $O(\log n)$



incomplete BST
height = $O(n)$

The Efficiency of Binary Search Tree Operations

<u>Operation</u>	<u>Average case</u>	<u>Worst case</u>
Retrieval	$O(\log n)$	$O(n)$
Insertion	$O(\log n)$	$O(n)$
Deletion	$O(\log n)$	$O(n)$
Traversal	$O(n)$	$O(n)$

Figure 11-34

The order of the retrieval, insertion, deletion, and traversal operations for the reference-based implementation of the ADT binary search tree

BSTs with heights $O(\log n)$

- It would be ideal if a BST was always close to a full binary tree
 - It's enough to guarantee that the height of tree is $O(\log n)$
 - To guarantee that we have to make the structure of the tree and insertion and deletion algorithms more complex
 - e.g. AVL trees (balanced), 2-3 trees, 2-3-4 trees (full but not binary), **red-black** trees (if red vertices are ignored then it's like a full tree)
-

Tree sort

- We can sort an array of elements using BST ADT.
 - Start with an empty BST and insert the elements one by one to the BST.
 - Traverse the tree in an in-order manner.
 - Cost: on average is $O(n \log (n))$ and worst case $O(n^2)$.
-

Saving a BST in a file.

- Sometimes we need to save a BST in a file and restore it later.
 - There are two options:
 - Saving the BST and restoring it to its original format.
 - Save the elements in the BST in a pre-order manner to the file. R
 - Saving the BST and restoring it to a balanced shape.
-

General Trees

- An n -ary tree
 - A generalization of a binary tree whose nodes each can have no more than n children

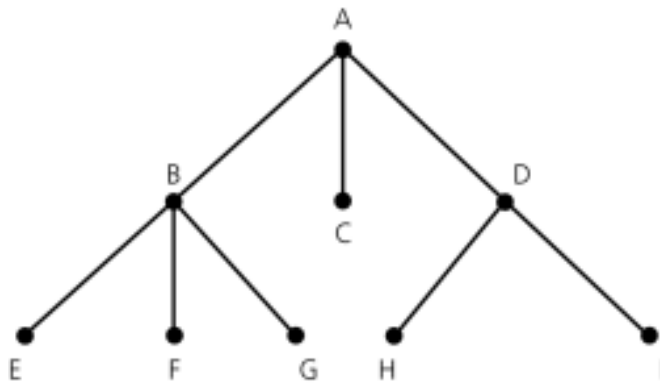


Figure 11-38

A general tree

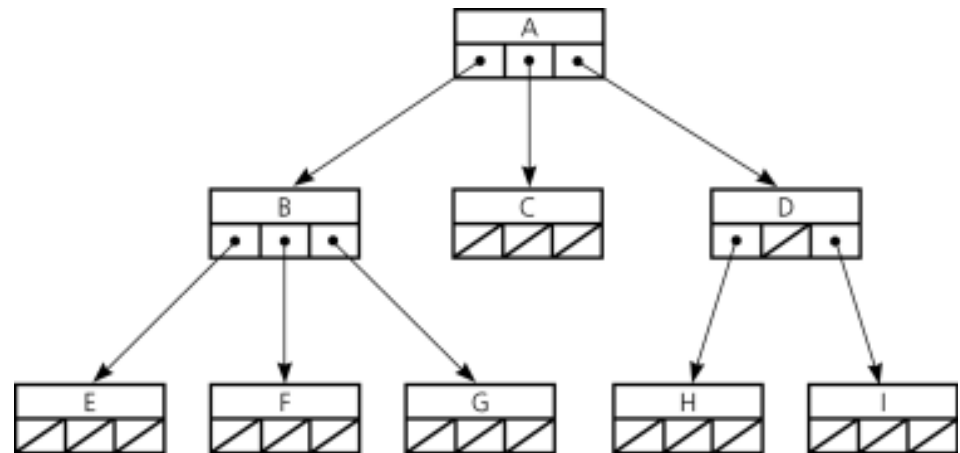


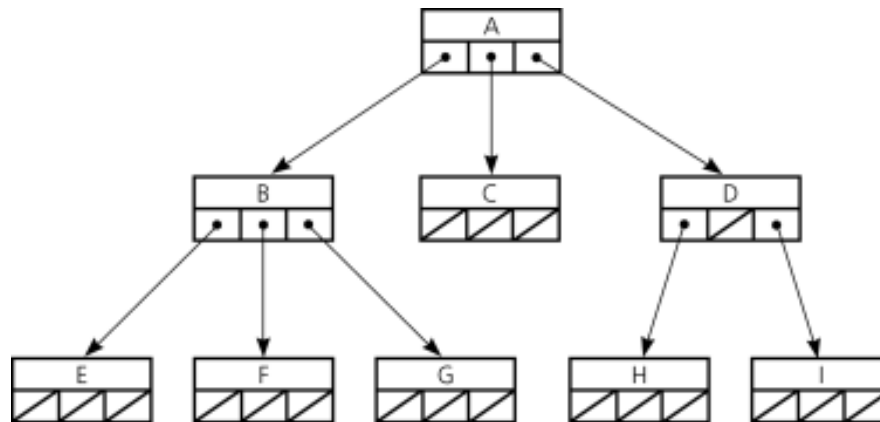
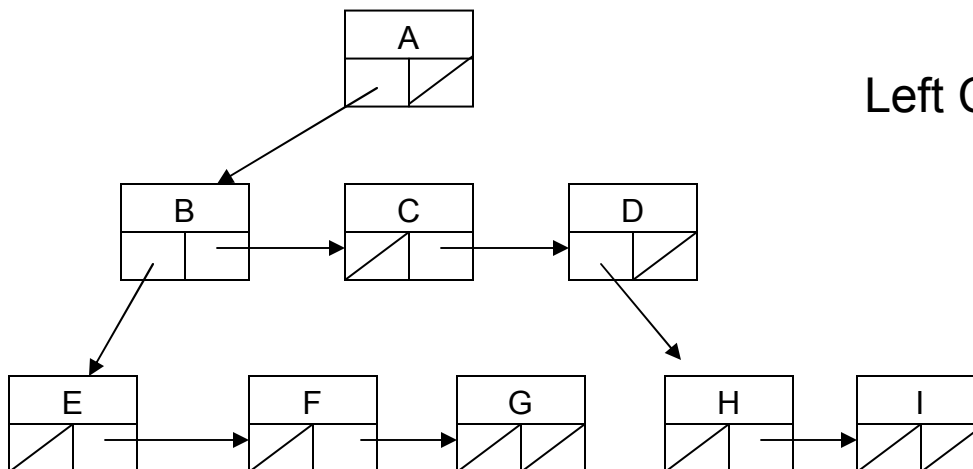
Figure 11-41

An implementation of the n -ary tree in Figure 11-38

```
public class GeneralTreeNode<T> {  
    private T item; // data item in the tree  
    private ArrayList<GeneralTreeNode<T>> child;  
    private static final int degree=3;  
  
    // constructors and methods appear here  
    public GeneralTreeNode(){  
        child = new ArrayList<GeneralTreeNode<T>>(degree);  
    }  
    public GeneralTreeNode getChild(int i){  
        return child.get(i);  
    }  
} // end TreeNode
```

-
- The problem with this implementation is that the number of null references is large (memory waste is huge).
 - **Null Pointer Theorem**
 - given a regular m-ary tree (a tree that each node has at most m children), the number of nodes n is related to the number of null pointers p in the following way:
$$p = (m - 1).n + 1$$
 - **Proof:** the total number of references is m.n .
 - the number of used references is equal to the number of edges which is n - 1.
 - Hence the number of unused (null) references is:
$$p = m.n - (n - 1) = (m-1)n + 1$$
 - This shows that the number of wasted references is minimum in a binary tree (m=2) right after a linked list (m=1).
-

-
- We can represent an m-ary tree using a binary tree.
 - This way we use less memory to store the tree.
 - To convert a general tree into a binary tree we make each node store a pointer to its right sibling and its left child.
-



Left Child Right sibling representation
of the above tree

-
- The only problem with the LC-RS (left child right sibling) representation is that accessing the children of a node is a constant time operation anymore. It is actually $O(m)$.
-