Binary Search Trees

CMPT 225

Objectives

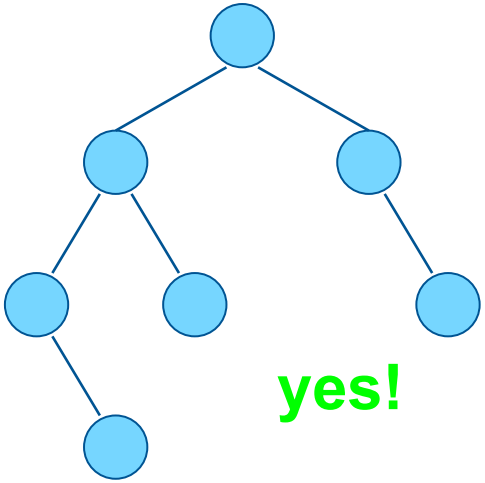
- Understand tree terminology
- Understand and implement tree traversals
- Define the binary search tree property
- Implement binary search trees
- Implement the TreeSort algorithm

Trees

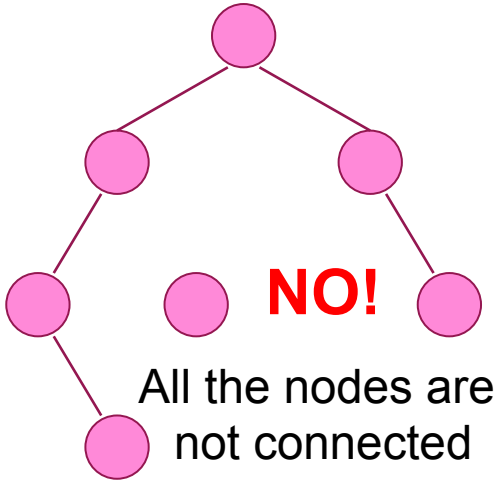
Trees

- A set of nodes (or vertices) with a single starting point
  - called the **root**
- Each node is connected by an **edge** to another node
- A tree is a connected graph
  - There is a path to every node in the tree
  - A tree has one fewer edges than the number of nodes

Is it a Tree?

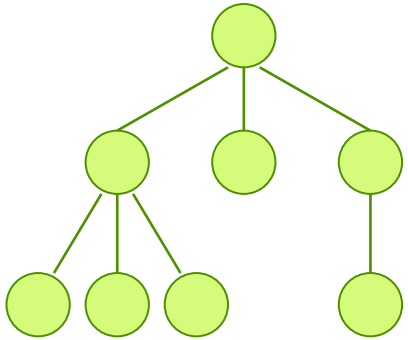


yes!

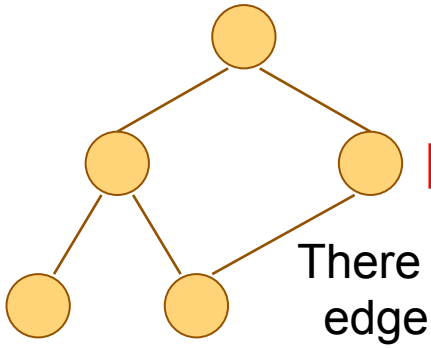


NO!

All the nodes are not connected

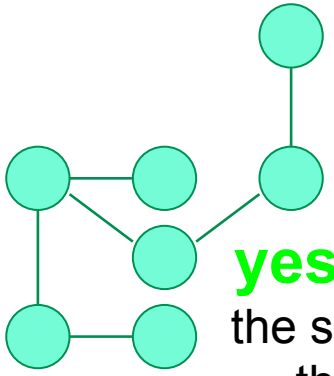


yes! (but not a binary tree)



NO!

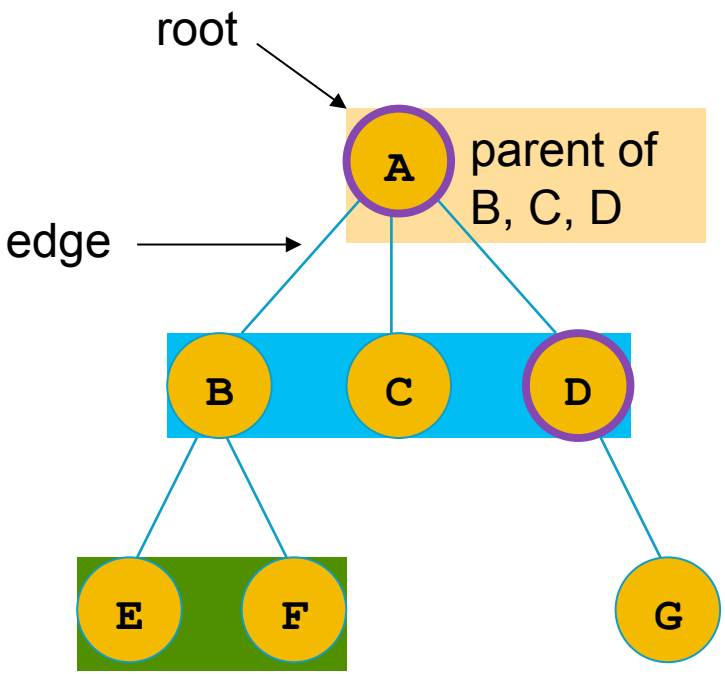
There is an extra edge (5 nodes and 5 edges)



yes! (it's actually the same graph as the blue one)

Tree Relationships

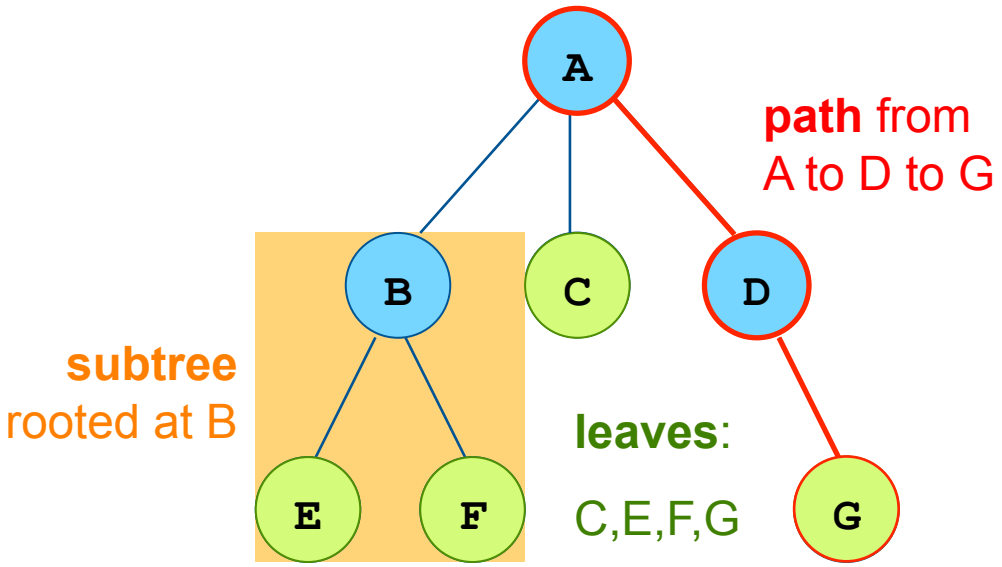
- Node **v** is said to be a **child** of **u**, and **u** the **parent** of **v** if
  - There is an edge between the nodes **u** and **v**, and
  - **u** is above **v** in the tree,
- This relationship can be generalized
  - E and F are **descendants** of A
  - D and A are **ancestors** of G
  - B, C and D are **siblings**
  - F and G are?



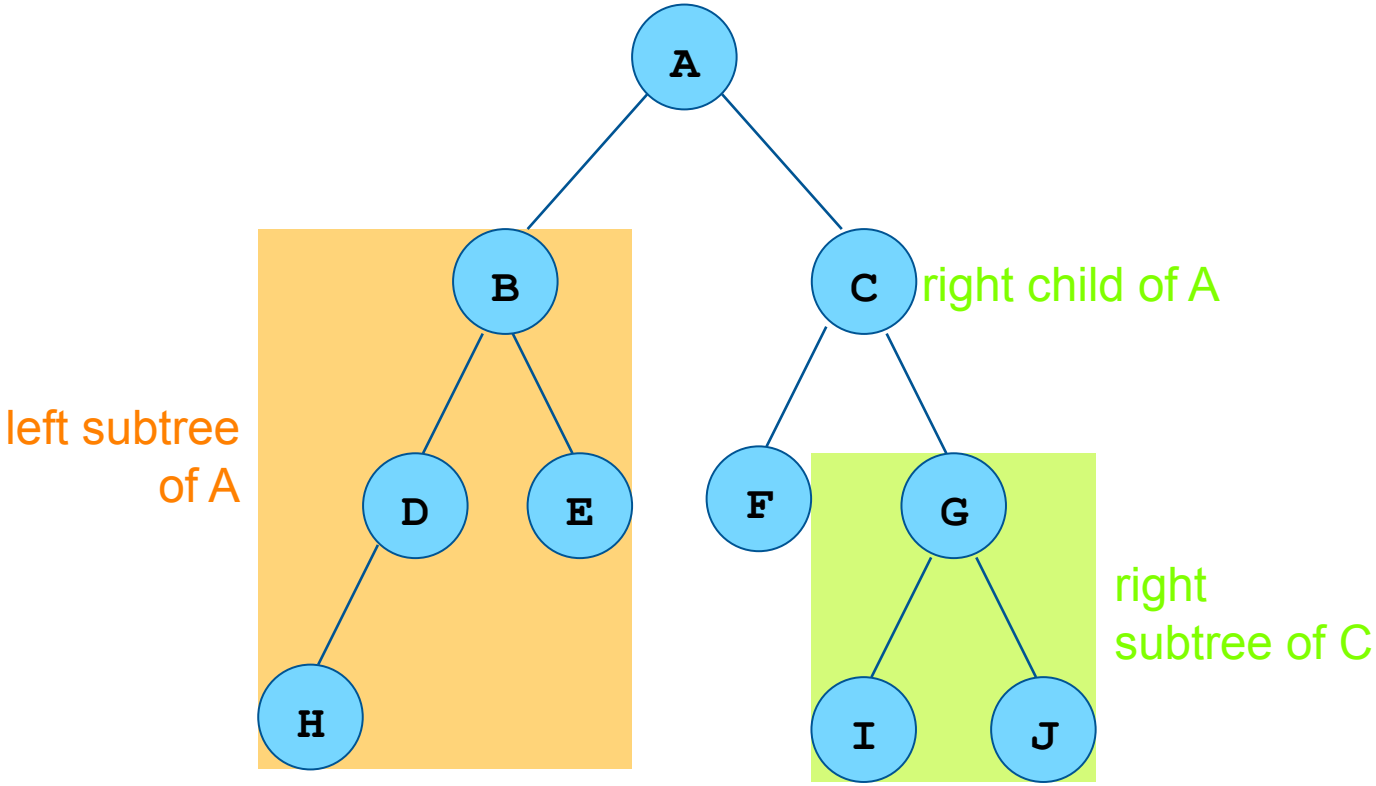
More Tree Terminology

- A **leaf** is a node with no children
- A **path** is a sequence of nodes $\mathbf{v}_1 \ldots \mathbf{v}_n$
  - where $\mathbf{v}_i$ is a parent of $\mathbf{v}_{i+1}$ ($1 \leq \mathbf{i} \leq \mathbf{n-1}$)
- A **subtree** is any node in the tree along with all of its descendants
- A **binary tree** is a tree with at most two children per node
  - The children are referred to as **left** and **right**
  - We can also refer to left and right subtrees

Tree Terminology Example



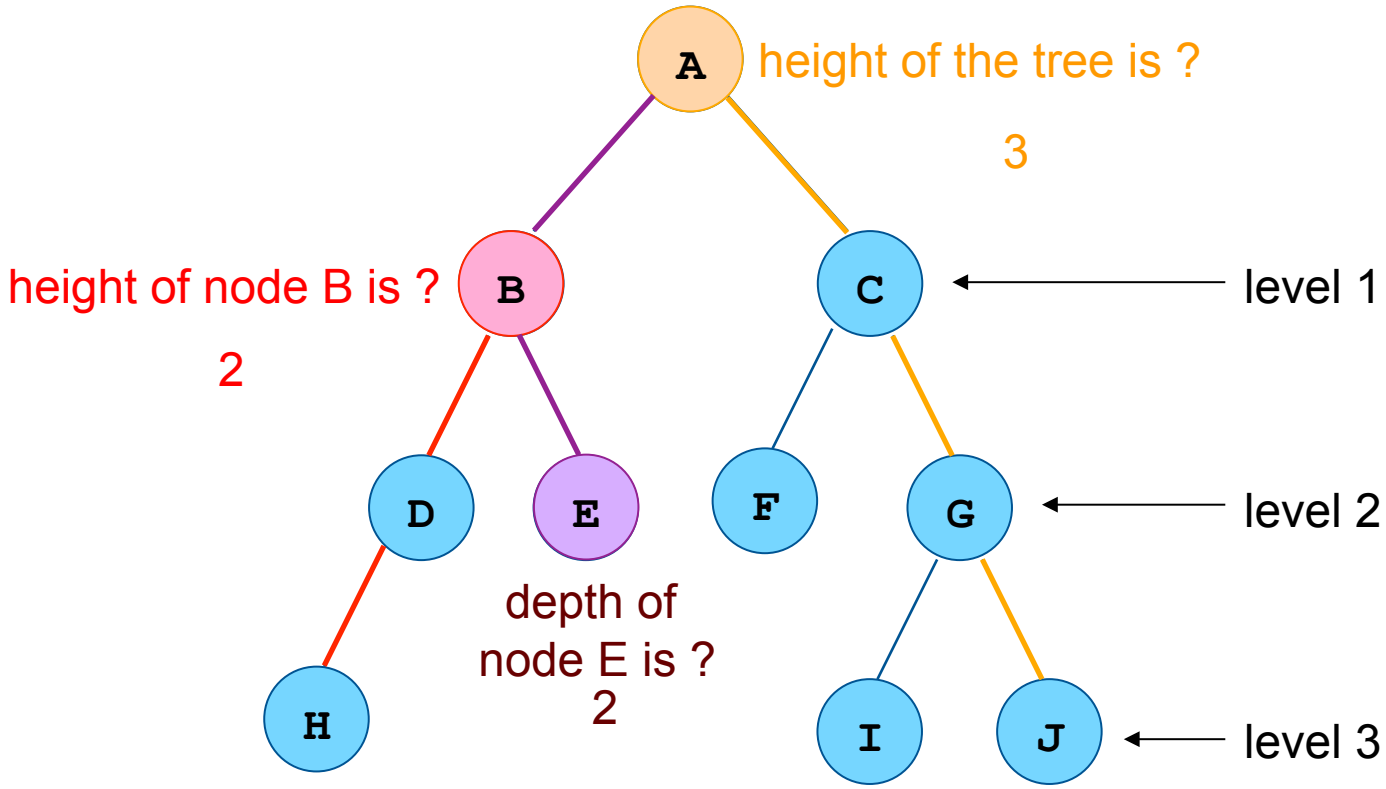
Binary Tree Terminology



Measuring Trees

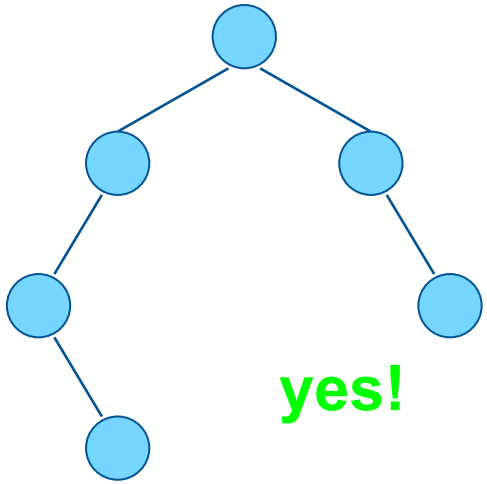
- The **height** of a node `v` is the length of the longest path from `v` to a leaf
  - The height of the tree is the height of the root
- The **depth** of a node `v` is the length of the path from `v` to the root
  - This is also referred to as the **level** of a node
- Note that there is a slightly different formulation of the height of a tree
  - Where the height of a tree is said to be the number of different **levels** of nodes in the tree (including the root)

Height of a Binary Tree

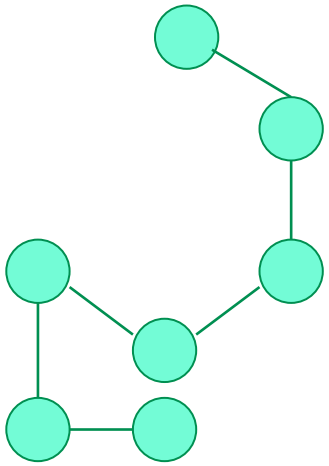


Beautiful Trees

Is it a Tree (II)?



yes!

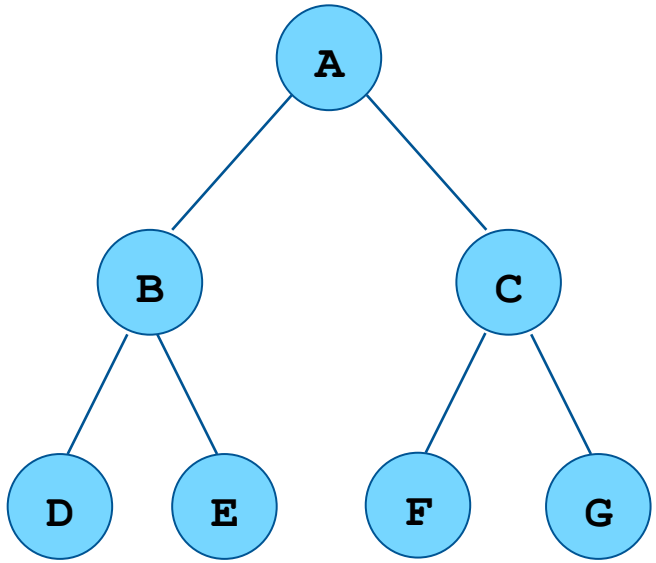


yes!

However, these trees are not “beautiful” (for some applications)

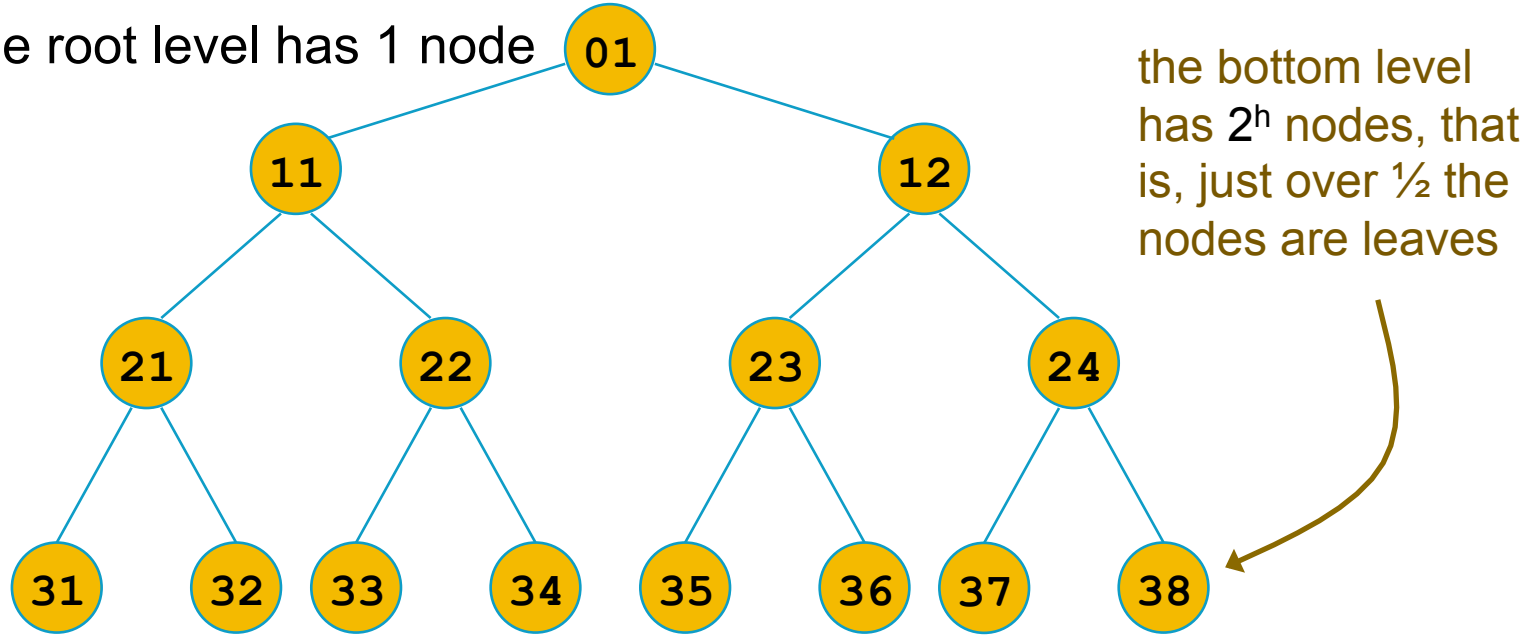
Perfect Binary Trees

- A binary tree is **perfect**, if
  - No node has only one child
  - And all the leaves have the same depth
- A perfect binary tree of height **h** has how many nodes?
  - $2^{h+1} - 1$ nodes, of which 2^{h} are leaves



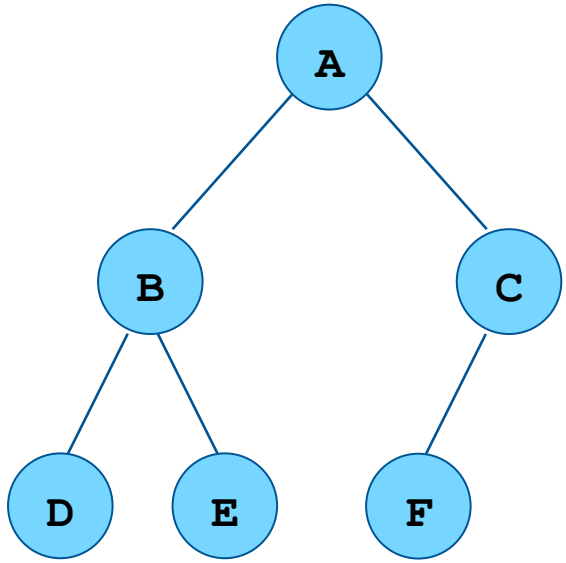
Height of a Perfect Tree

- Each level doubles the number of nodes
  - Level 1 has 2 nodes (2^1)
  - Level 2 has 4 nodes (2^2) or 2 times the number in Level 1
- Therefore a tree with h levels has 2^{h+1} - 1nodes
  - The root level has 1 node



Complete Binary Trees

- A binary tree is **complete** if
  - The leaves are on at most two different levels,
  - The second to bottom level is completely filled in, and
  - The leaves on the bottom level are as far to the left as possible

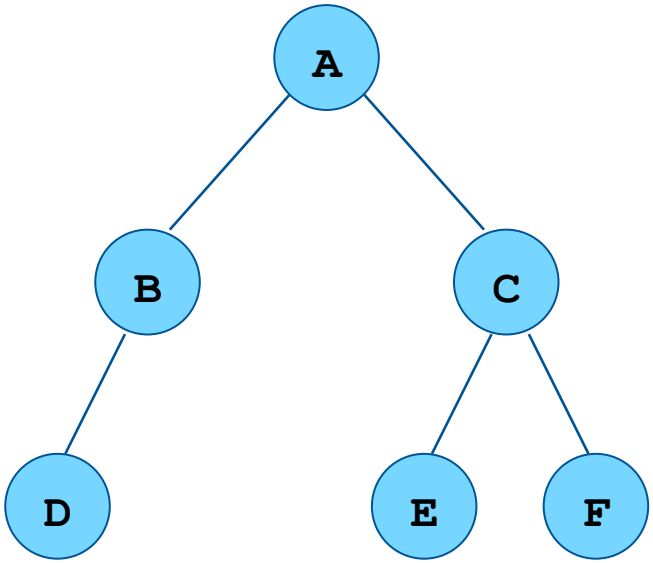


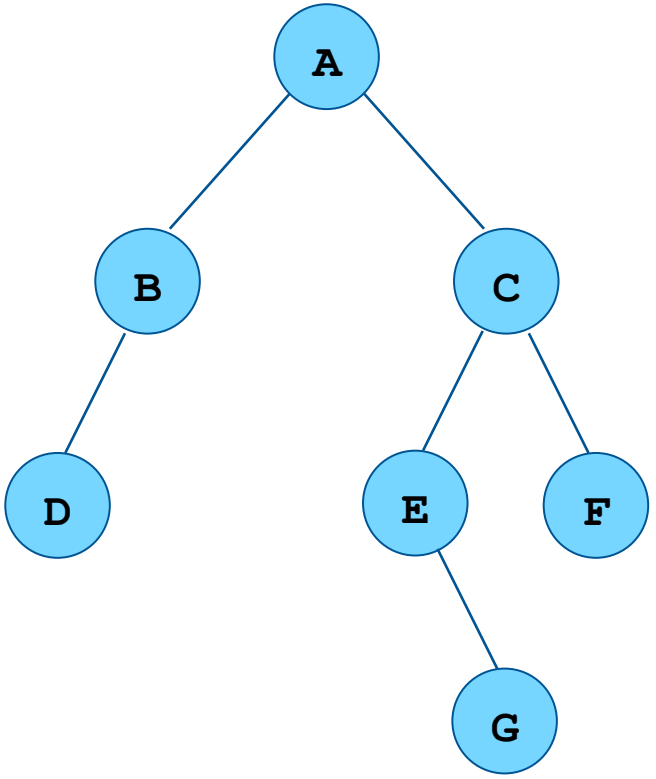
- Perfect trees are also complete

Balanced Binary Trees

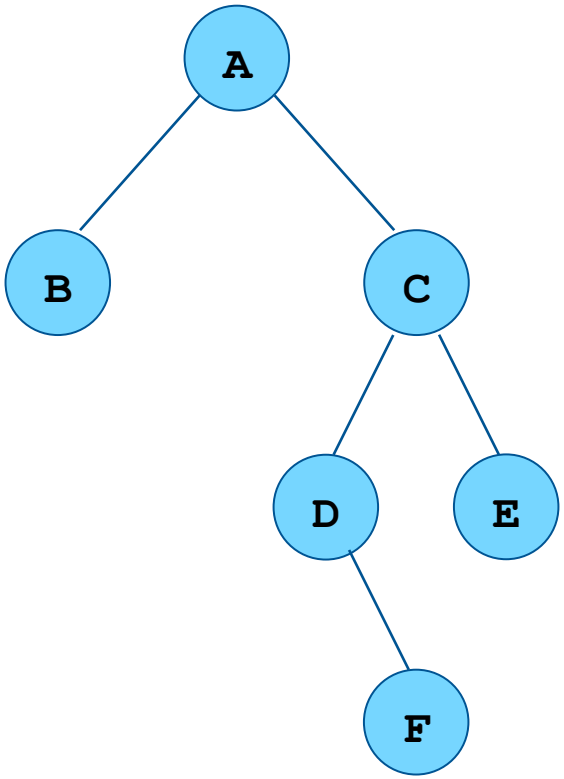
- A binary tree is **balanced** if
  - Leaves are all about the same distance from the root
  - The exact specification varies
- Sometimes trees are balanced by comparing the height of nodes
  - e.g. the height of a node's right subtree is at most one different from the height of its left subtree
- Sometimes a tree's height is compared to the number of nodes
  - e.g. red-black trees

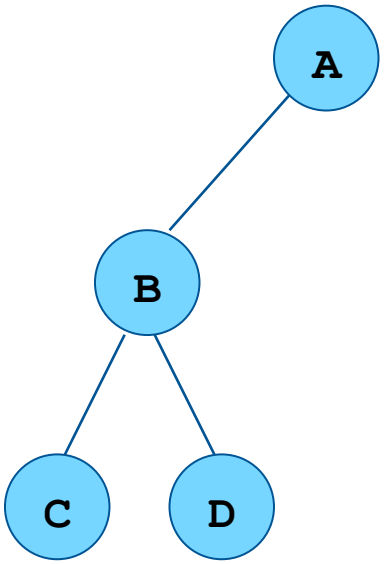
Balanced Binary Trees





Unbalanced Binary Trees





Tree Traversals

Binary Tree Traversals

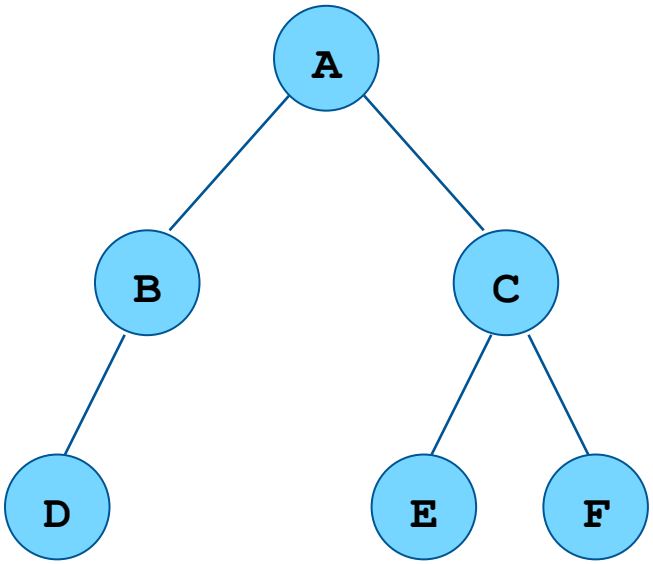
- A traversal algorithm for a binary tree visits each node in the tree
  - Typically, it will do something while visiting each node!
- Traversal algorithms are naturally recursive
- There are three traversal methods
  - Inorder
  - Preorder
  - Postorder

InOrder Traversal Algorithm

C++

```
// InOrder traversal algorithm
void inOrder(Node *n) {
    if (n != 0) {
        inOrder(n->leftChild);
        visit(n);
        inOrder(n->rightChild);
    }
}
```

InOrder Traversal



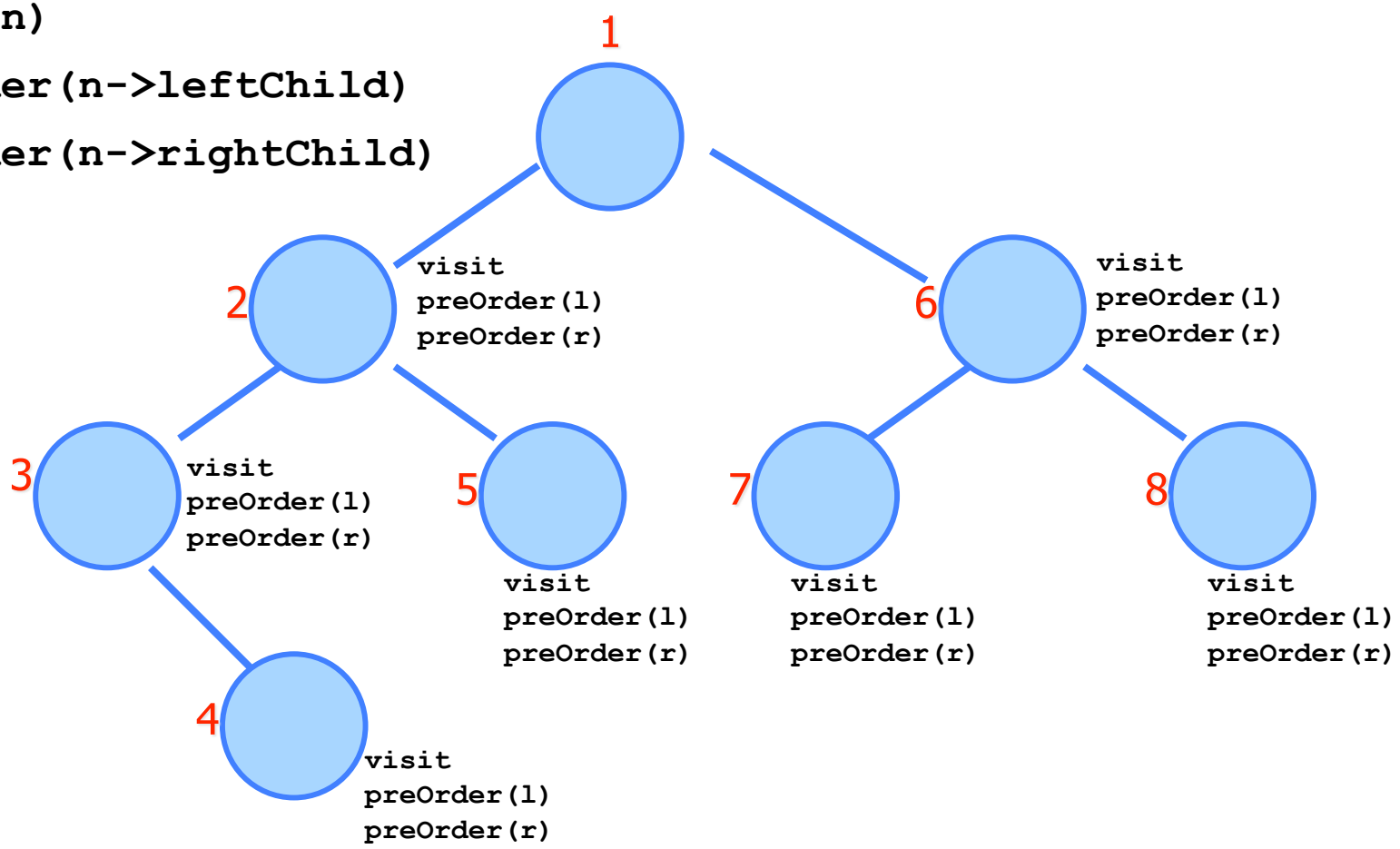
PreOrder Traversal Algorithm

C++

```
// PreOrder traversal algorithm
void preOrder(Node *n) {
    if (n != 0) {
        visit(n);
        preOrder(n->leftChild);
        preOrder(n->rightChild);
    }
}
```

PreOrder Traversal

```
visit(n)
preOrder(n->leftChild)
preOrder(n->rightChild)
```



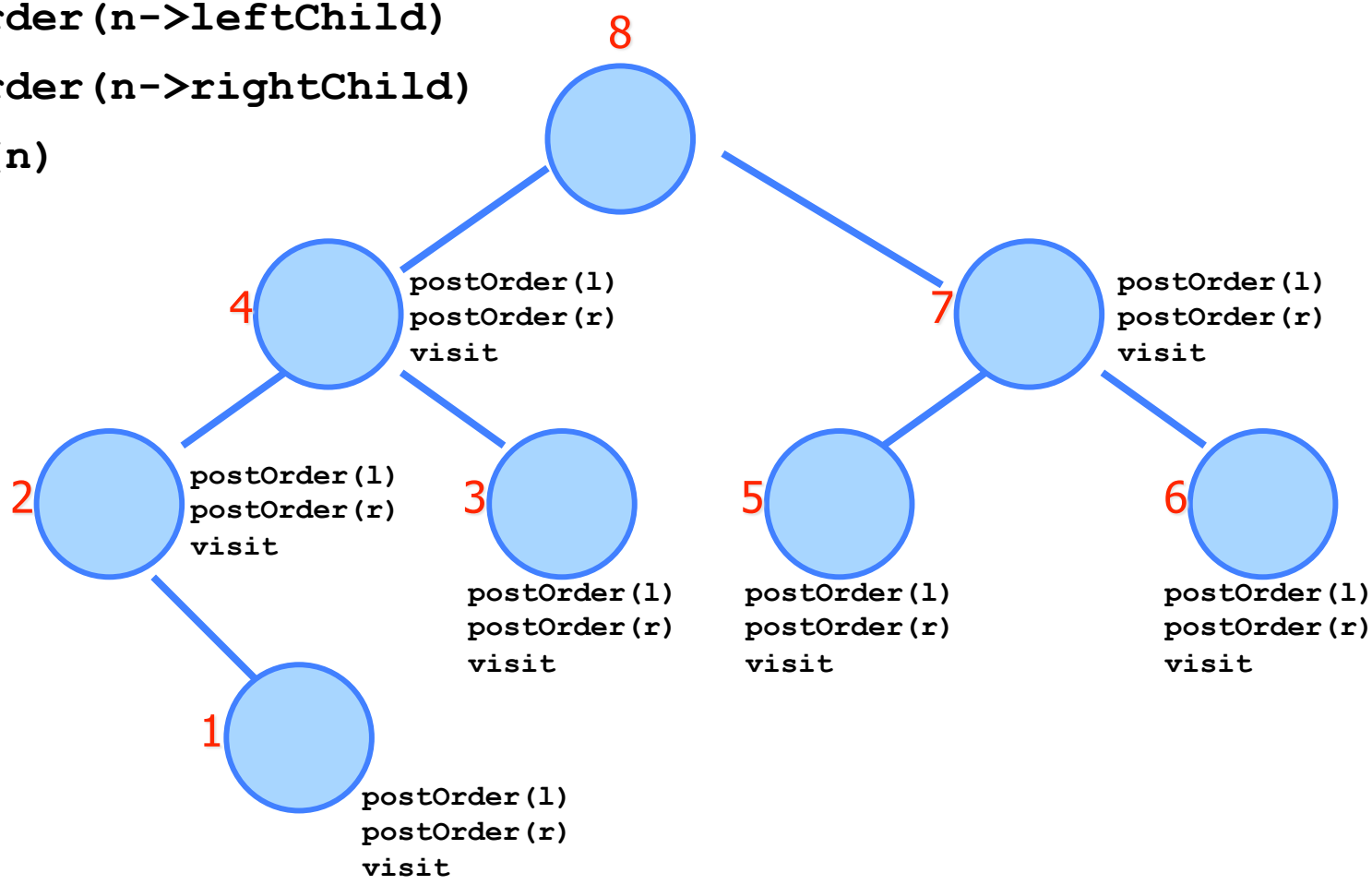
PostOrder Traversal Algorithm

C++

```
// PostOrder traversal algorithm
void postOrder(Node *n) {
    if (n != 0) {
        postOrder(n->leftChild);
        postOrder(n->rightChild);
        visit(n);
    }
}
```

PostOrder Traversal

```
postOrder(n->leftChild)
postOrder(n->rightChild)
visit(n)
```



Binary Tree Implementation and Binary Search Trees

Binary Tree Implementation

- The binary tree can be implemented using a number of data structures
  - Reference structures (similar to linked lists)
  - Arrays
- We will look at three implementations
  - Binary search trees (reference / pointers)
  - Red – black trees (reference / pointers)
  - Heap (arrays)

Problem: Accessing Sorted Data

- Consider maintaining data in some order
  - The data is to be frequently searched on the sort key e.g. a dictionary
- Possible solutions might be:
  - A sorted array
    - Access in O(log**n**) using binary search
    - Insertion and deletion in linear time
  - An ordered linked list
    - Access, insertion and deletion in linear time
  - Neither of these is efficient

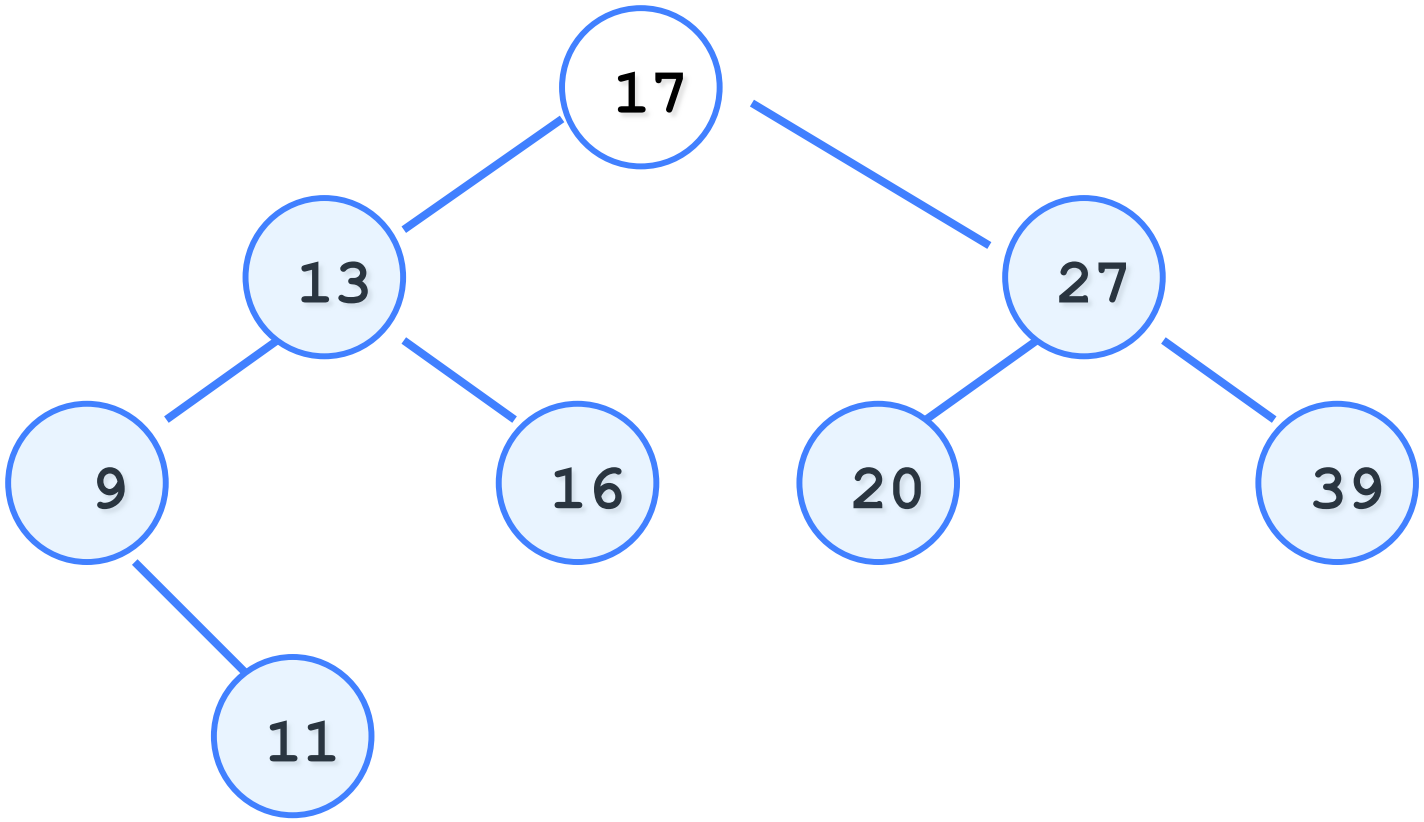
Dictionary Operations

- The data structure should be able to perform all these operations efficiently
  - Create an empty dictionary
  - Insert
  - Delete
  - Look up
- The insert, delete and look up operations should be performed in at most O(log**n**) time

Binary Search Tree Property

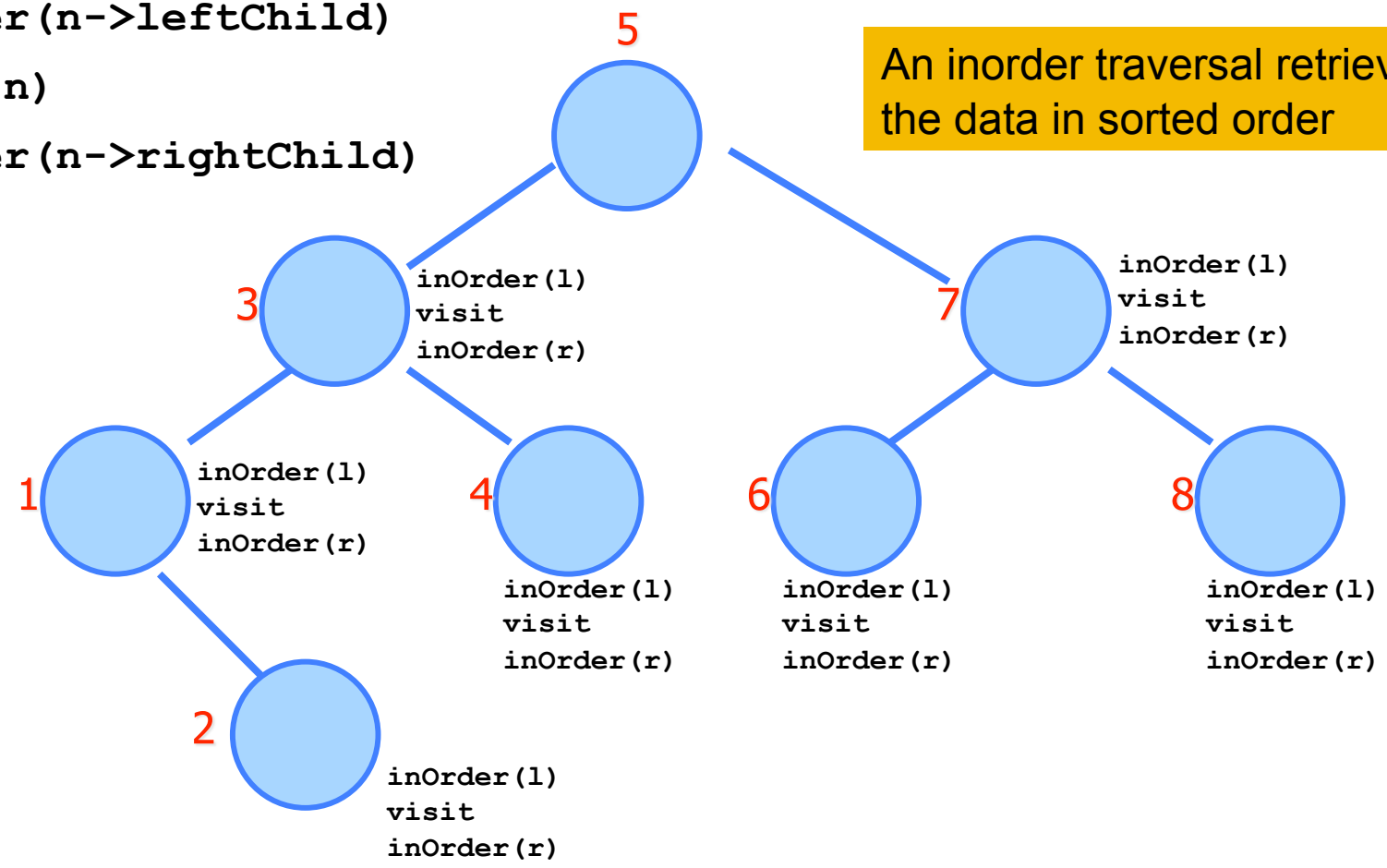
- A binary search tree (BST) is a binary tree with a special property
  - For all nodes in the tree:
    - All nodes in a left subtree have labels **less** than the label of the node
    - All nodes in a right subtree have labels **greater** than or equal to the label of the node
- Binary search trees are fully ordered

BST Example



BST InOrder Traversal

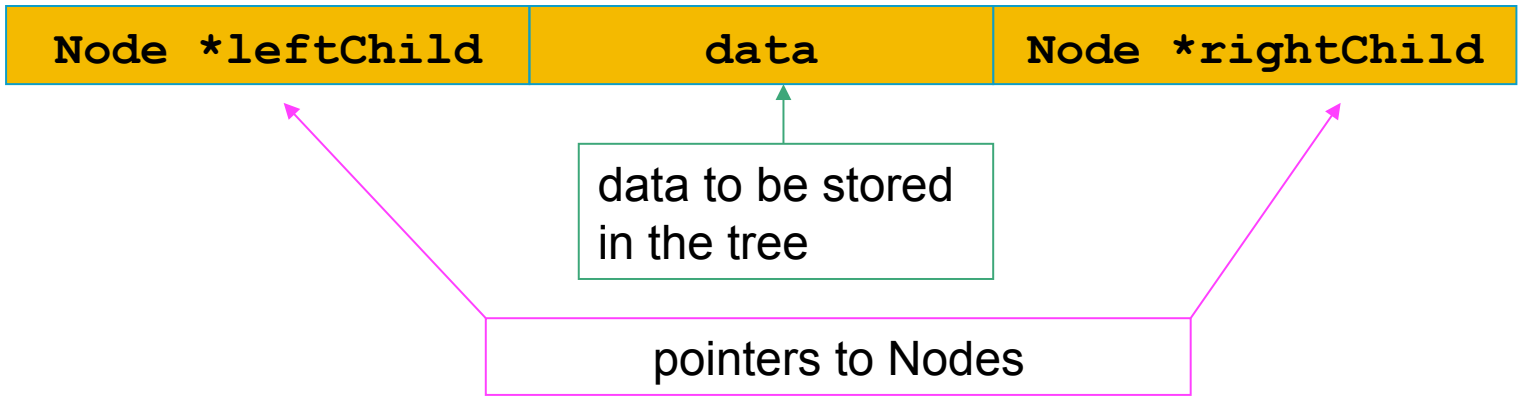
```
inOrder(n->leftChild)
visit(n)
inOrder(n->rightChild)
```



An inorder traversal retrieves the data in sorted order

BST Implementation

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



BST Search

- To find a value in a BST search from the root node:
  - 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
  - Otherwise return true, or return data, etc.
- How many comparisons?
  - One for each node on the path
  - Worst case: height of the tree + 1

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 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(**height**)

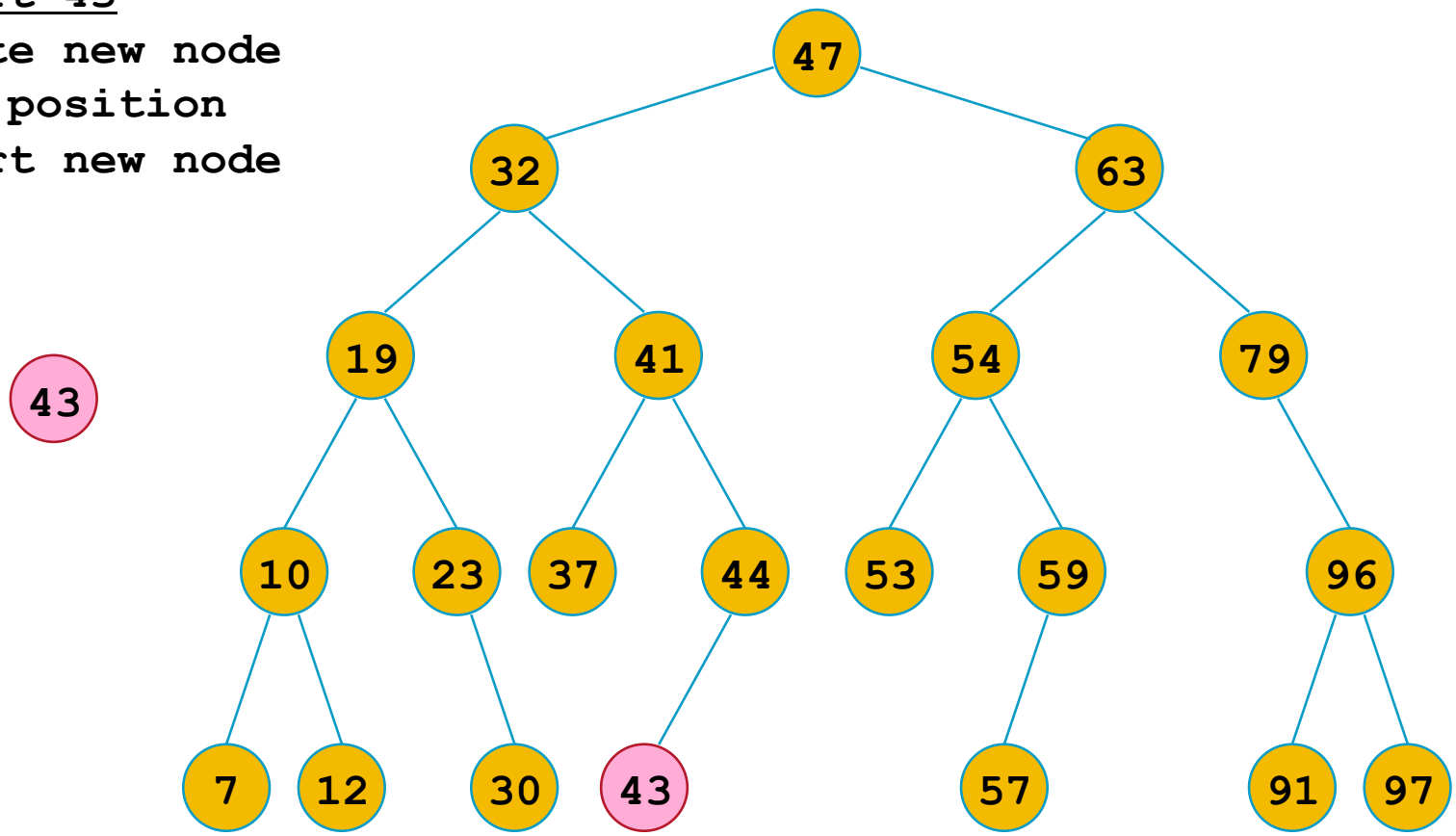
BST Insertion Example

insert 43

create new node

find position

insert new node



BST Deletion

- After deletion the BST property must hold
- Deletion is not as straightforward as search or insertion
  - So much so that sometimes it is not even implemented!
  - Deleted nodes are marked as deleted in some way
- There are a number of different cases that must be considered

BST Deletion Cases

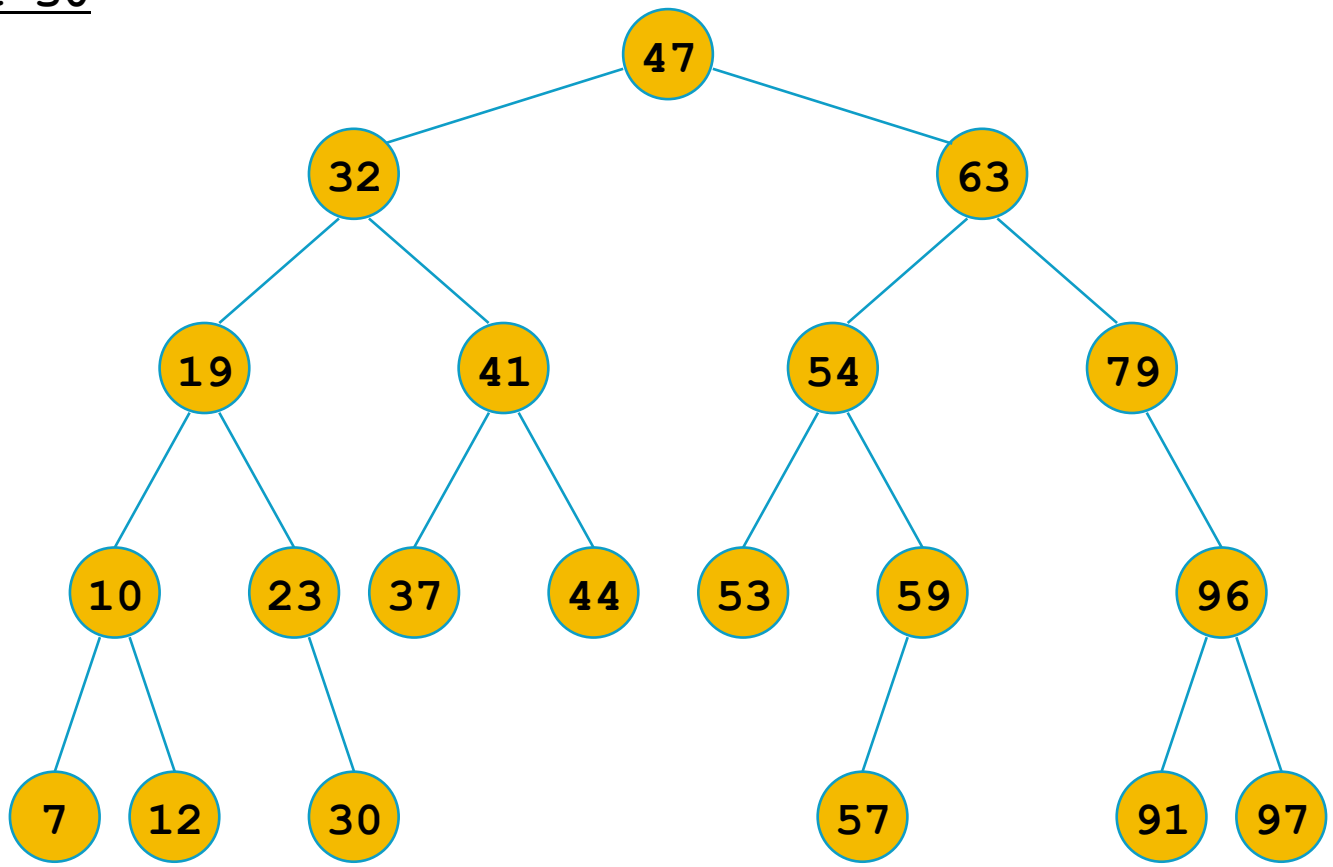
- The node to be deleted has no children
- The node to be deleted has one child
- The node to be deleted has two children

BST Deletion Cases

- The node to be deleted has no children
  - Remove it (assigning null to its parent's reference)

BST Deletion – target is a leaf

`delete 30`



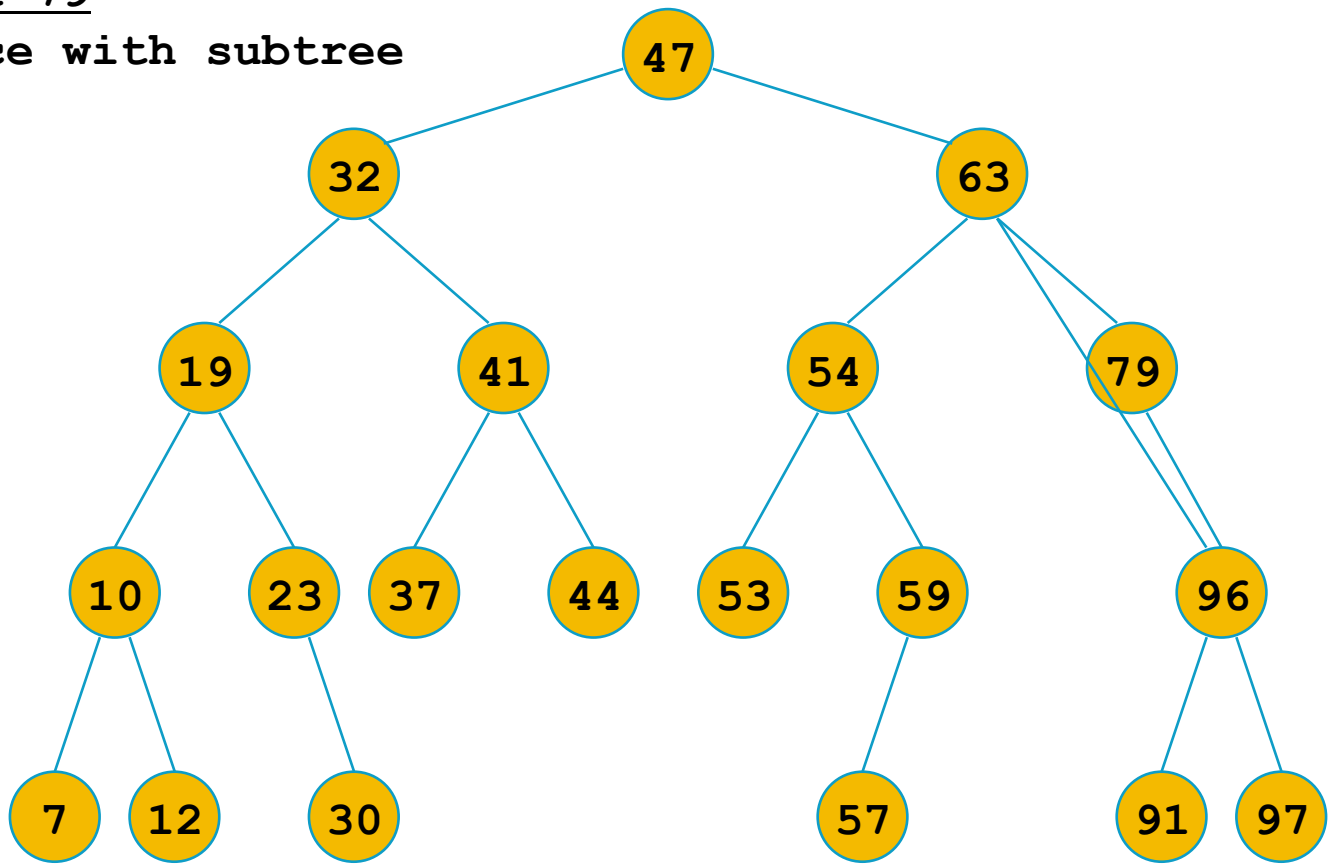
BST Deletion Cases

- The node to be deleted has one child
  - Replace the node with its subtree

BST Deletion – target has one child

delete 79

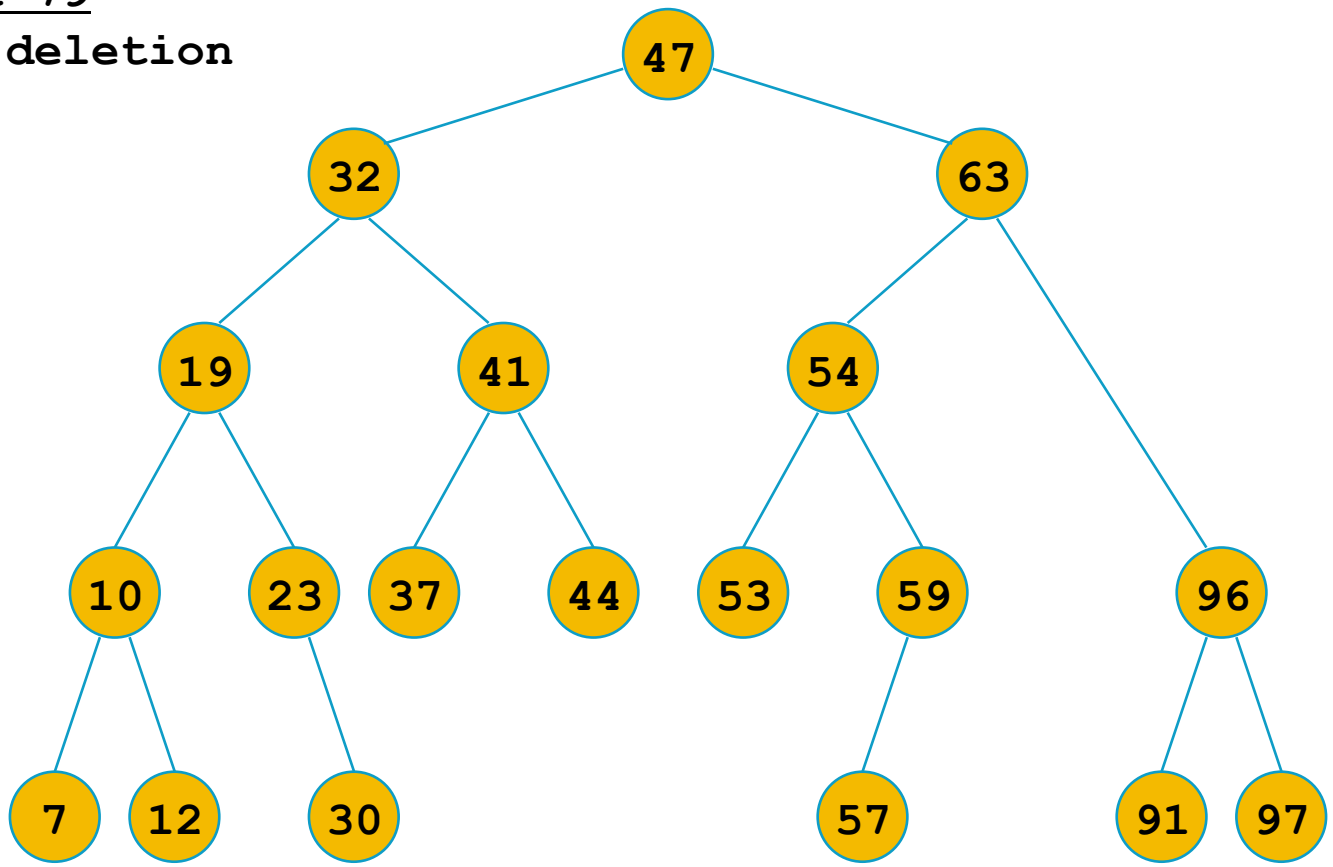
replace with subtree



BST Deletion – target has one child

delete 79

after deletion



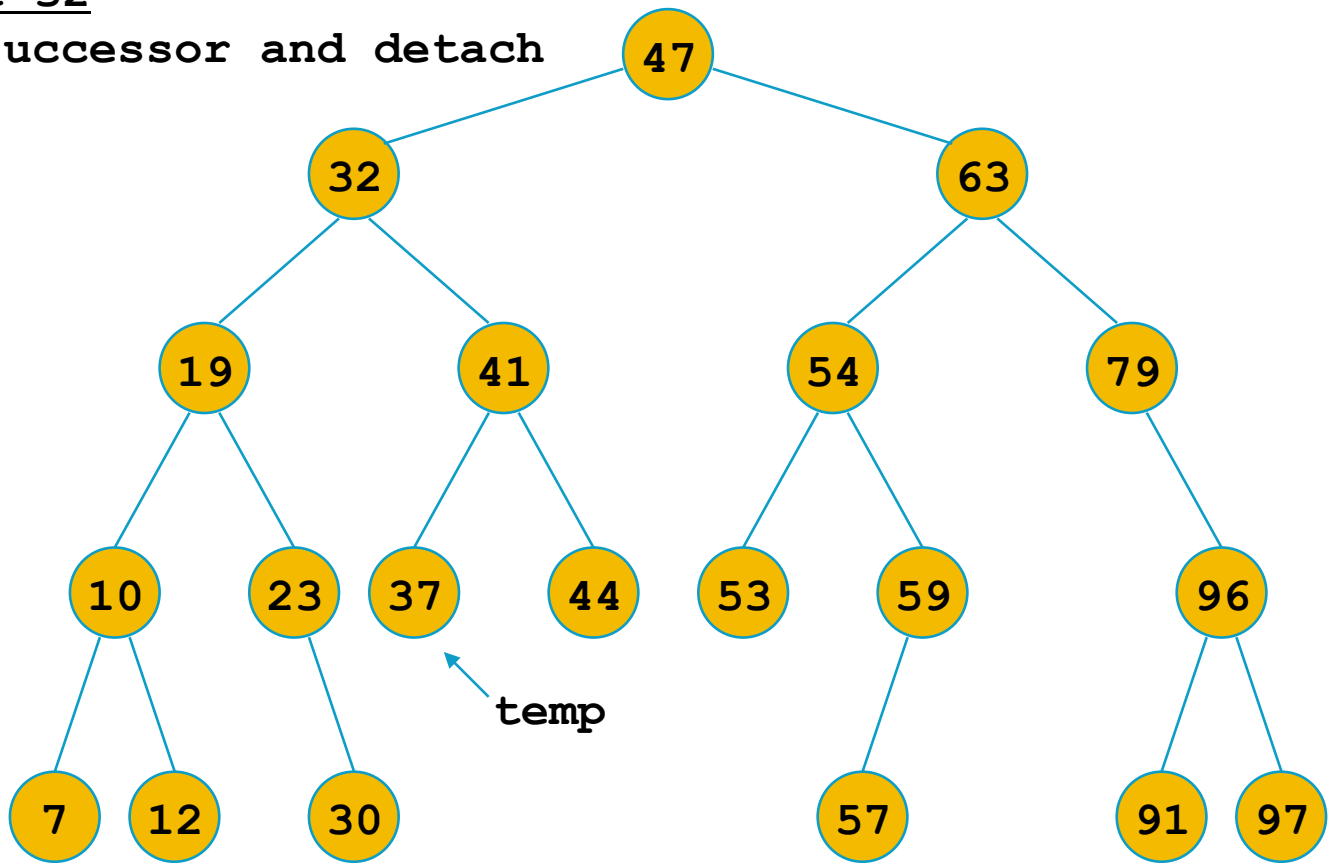
BST Deletion Cases

- The node to be deleted has two children
  - Replace the node with its **successor**, the **left most node** of its **right subtree**
    - It is also possible to replace the node with its **predecessor**, the **right most node** of its **left** subtree
  - If that node has a child (and it can have at most one child) attach it to the node's parent
    - Why can a predecessor or successor have at most one child?

BST Deletion – target has 2 children

delete 32

find successor and detach

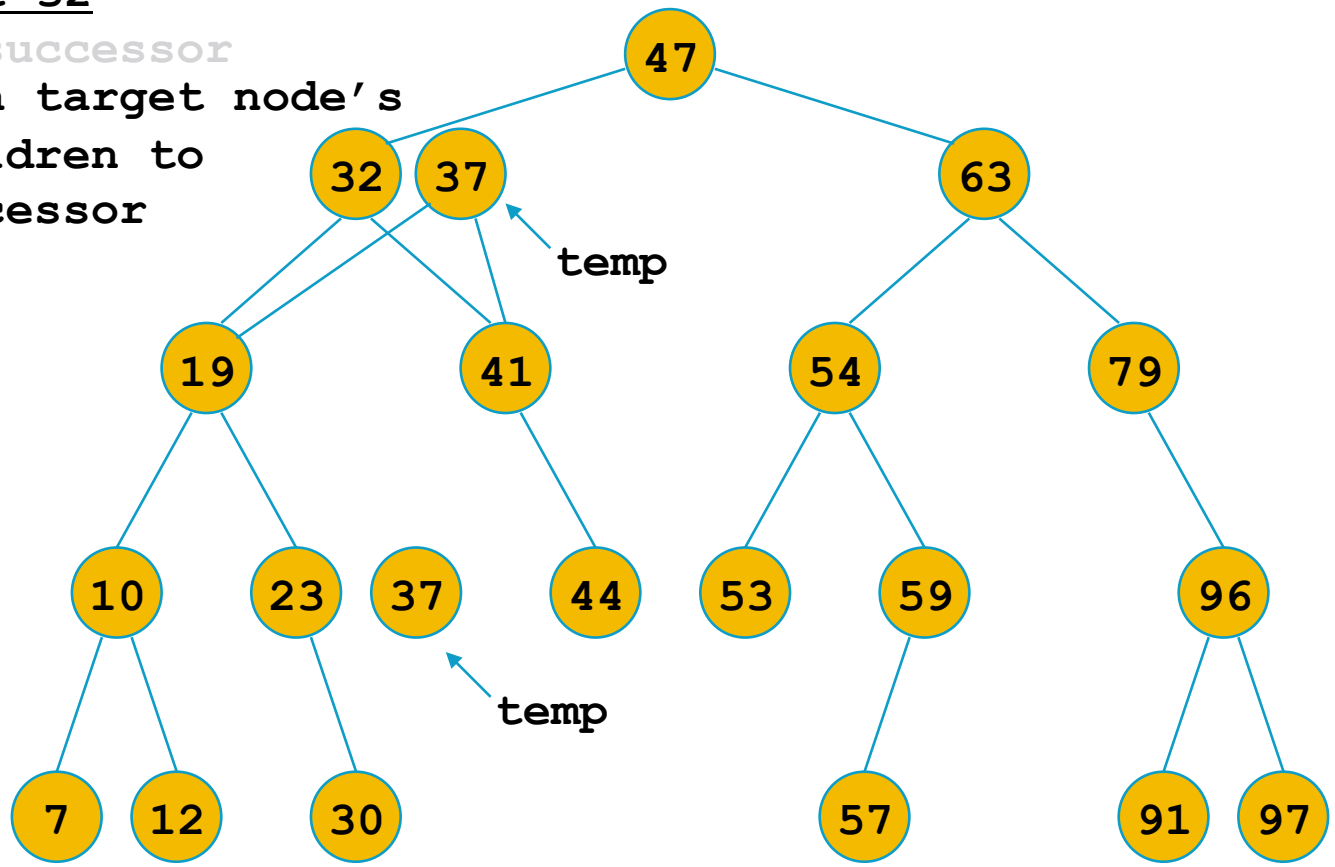


BST Deletion – target has 2 children

delete 32

find successor

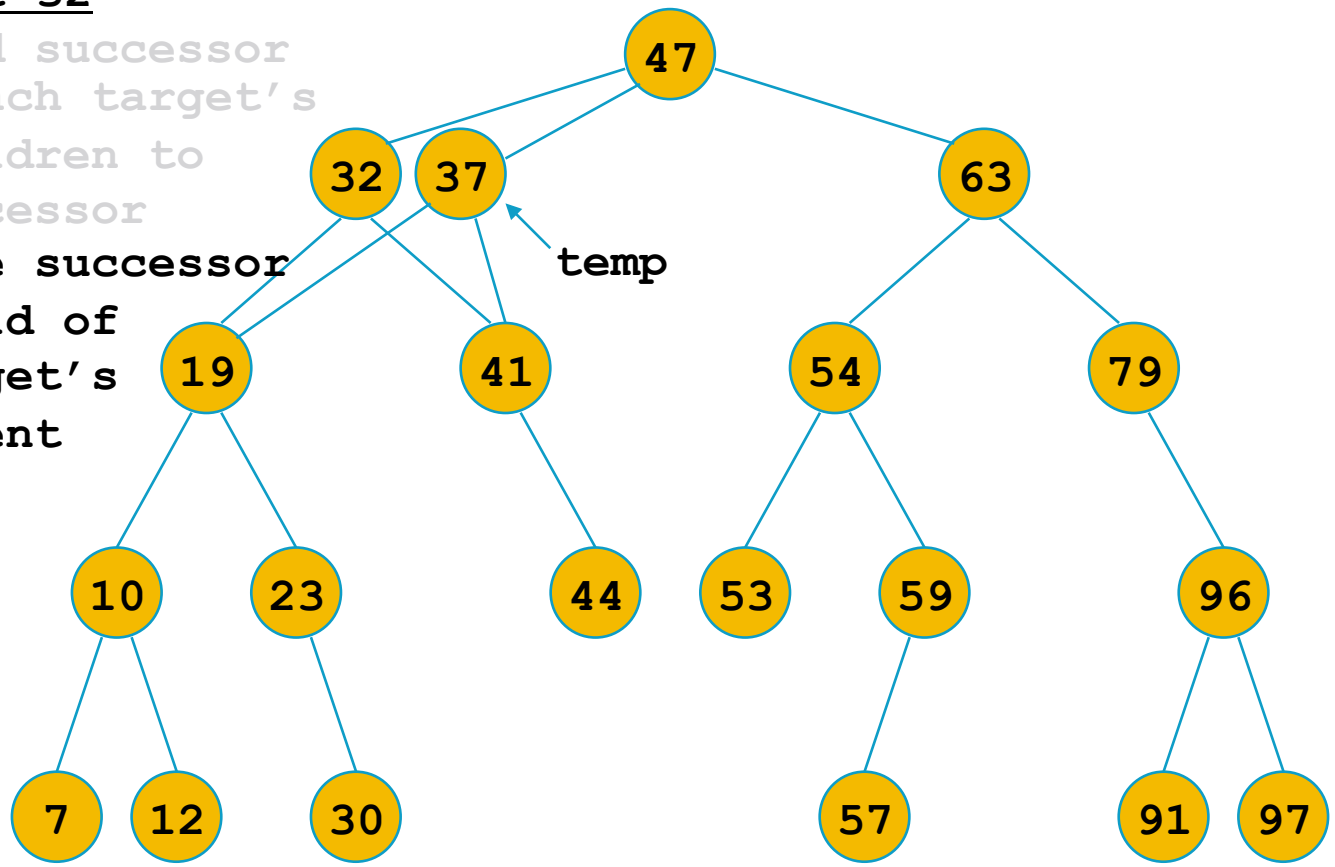
attach target node's children to successor



BST Deletion – target has 2 children

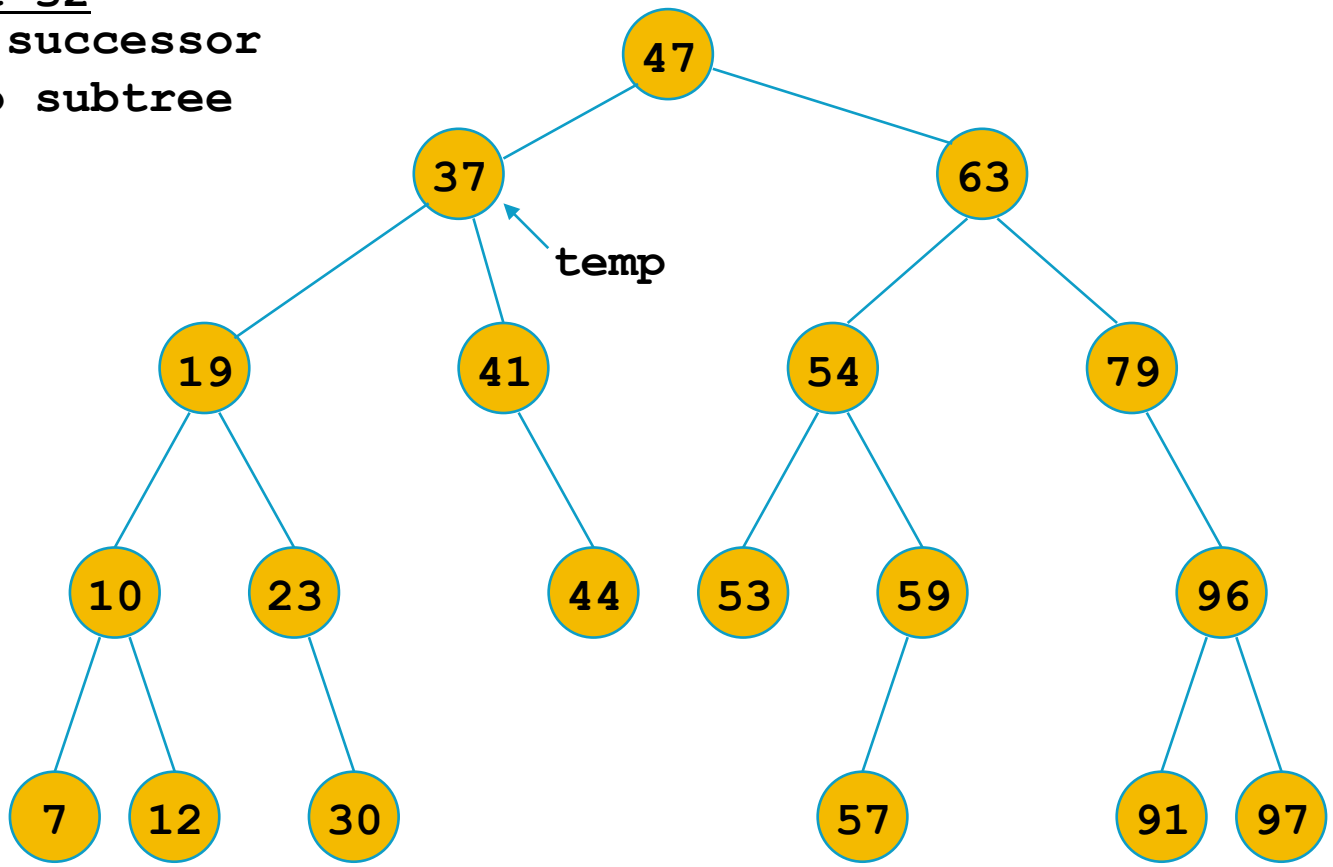
delete 32

- find successor
- attach target's children to successor
- **make successor child of target's parent**



BST Deletion – target has 2 children

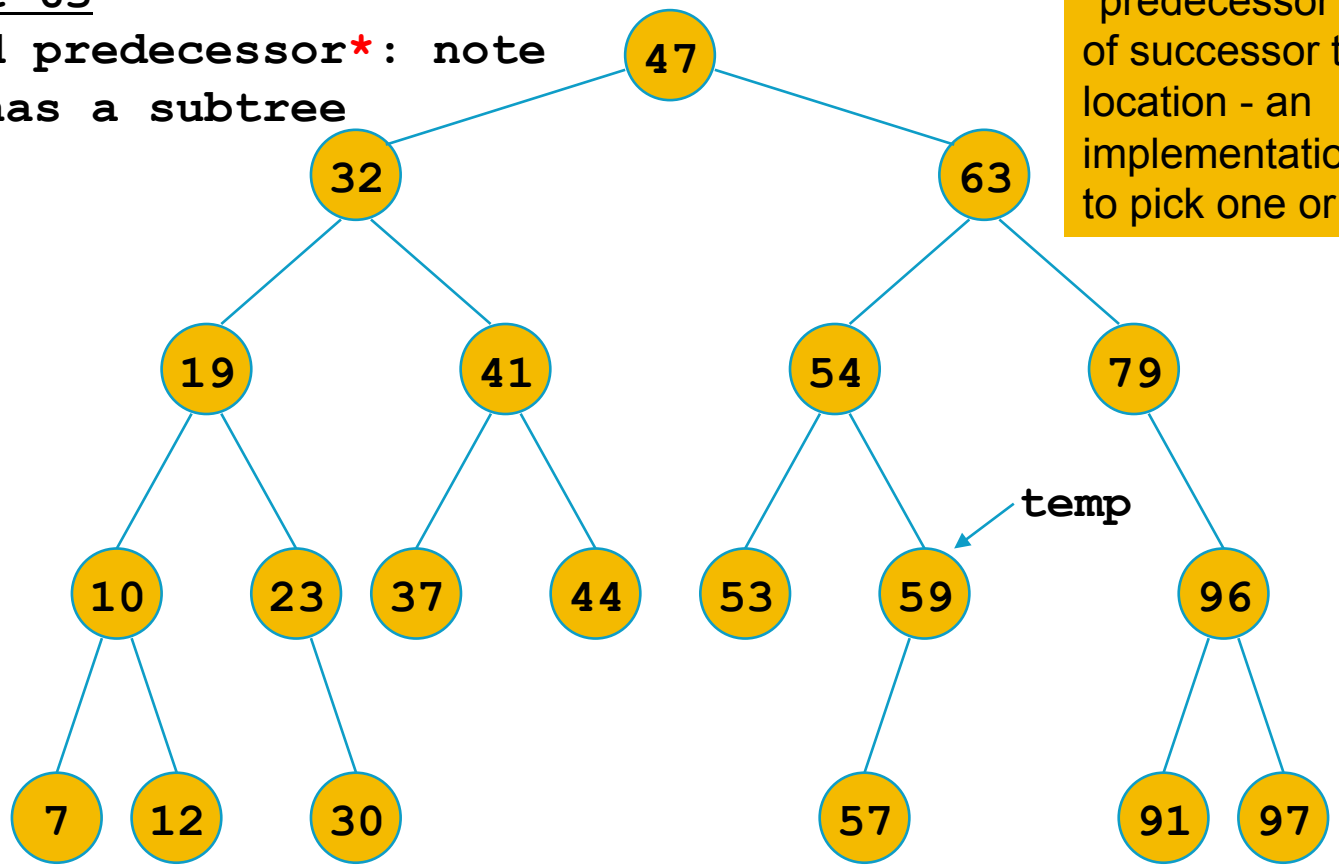
delete 32
note: successor had no subtree



BST Deletion – target has 2 children

delete 63

- find predecessor*: note it has a subtree

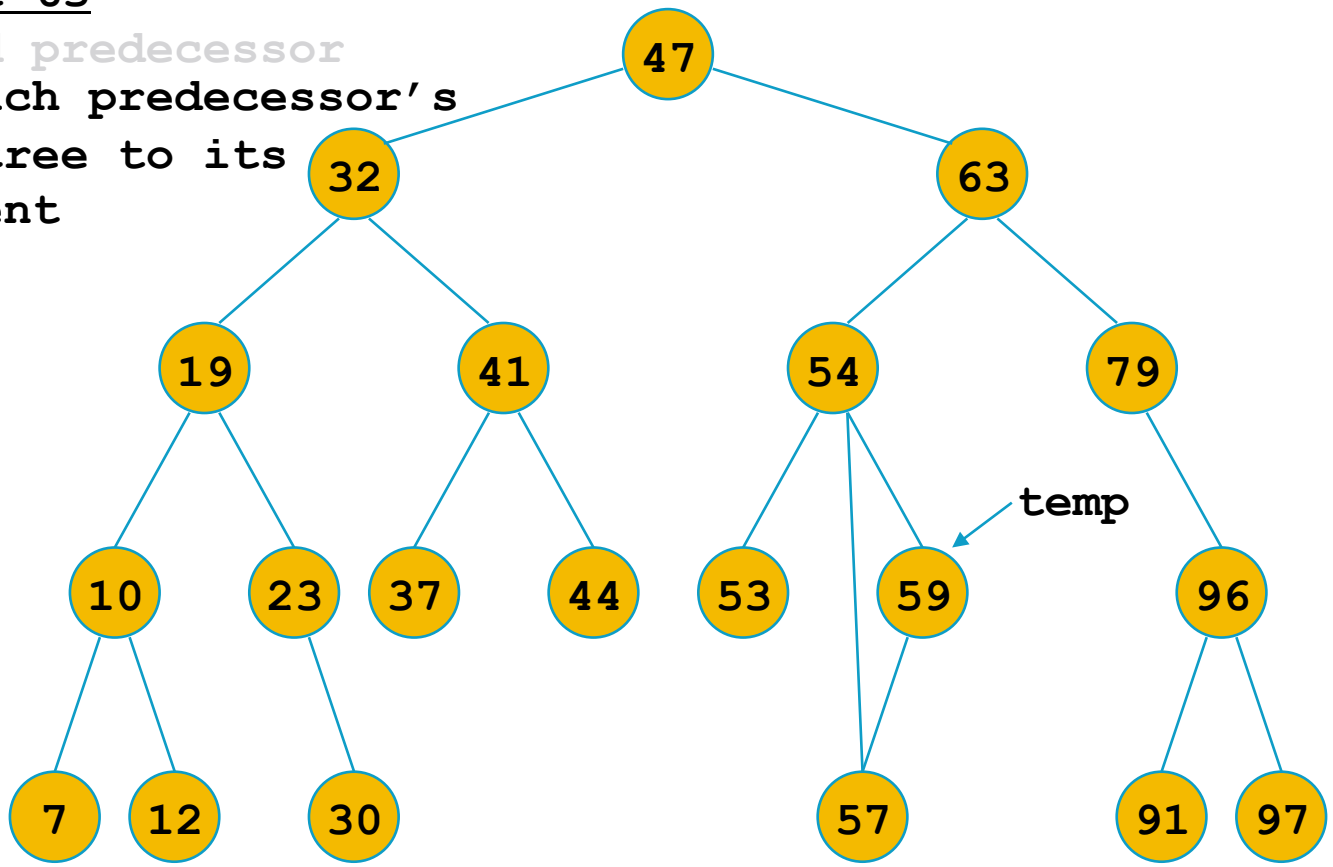


*predecessor used instead of successor to show its location - an implementation would have to pick one or the other

BST Deletion – target has 2 children

delete 63

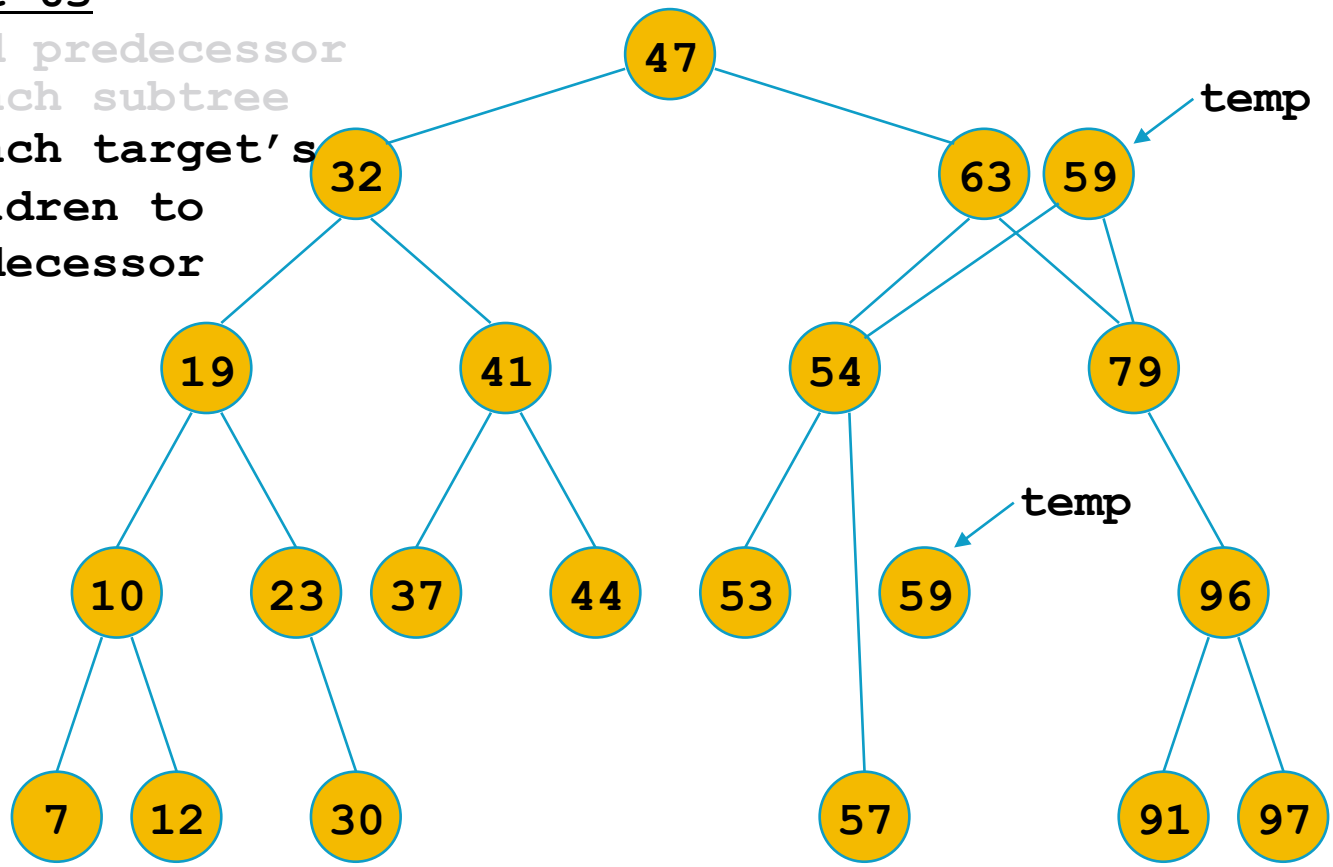
- find predecessor
- attach predecessor's subtree to its parent



BST Deletion – target has 2 children

delete 63

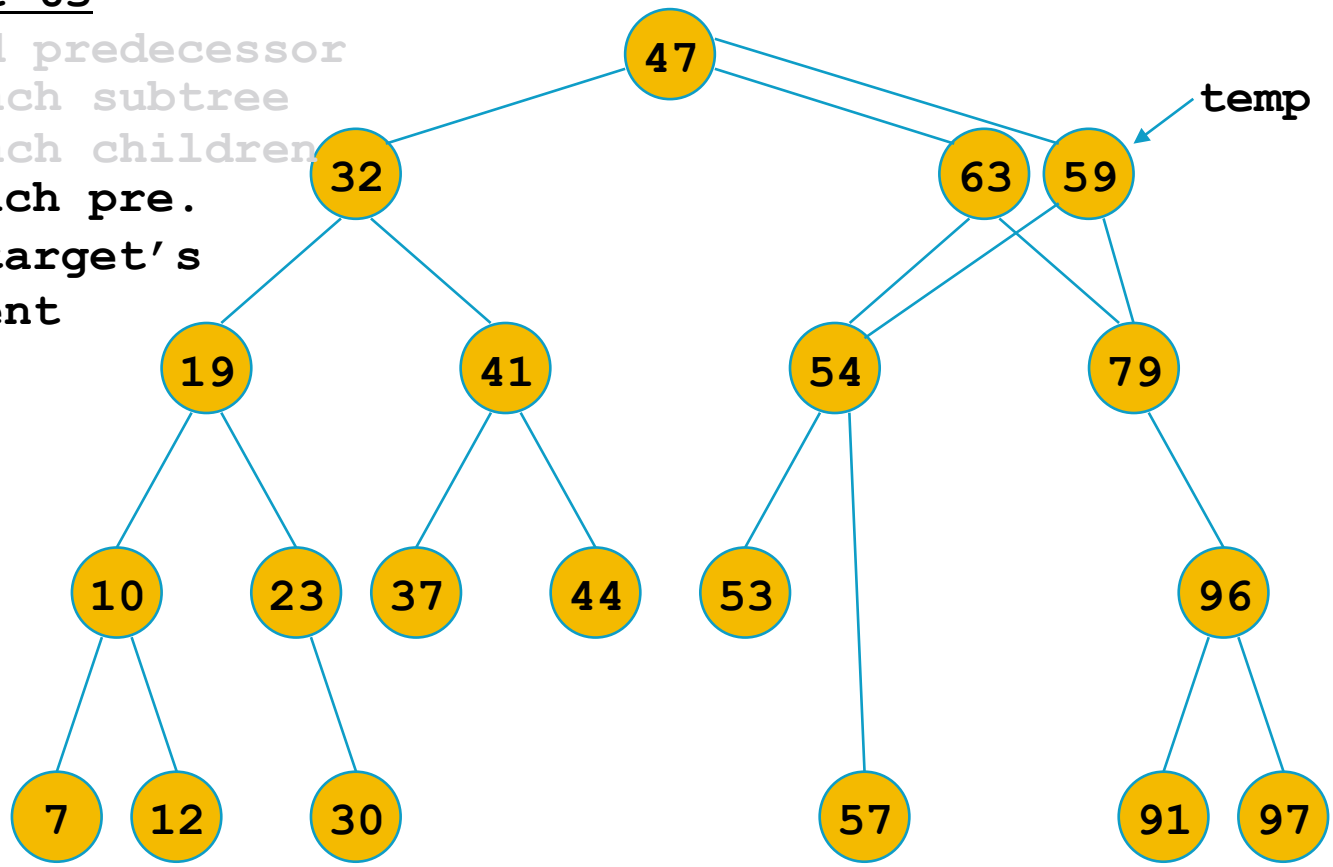
- find predecessor
- attach subtree
- attach target's children to predecessor



BST Deletion – target has 2 children

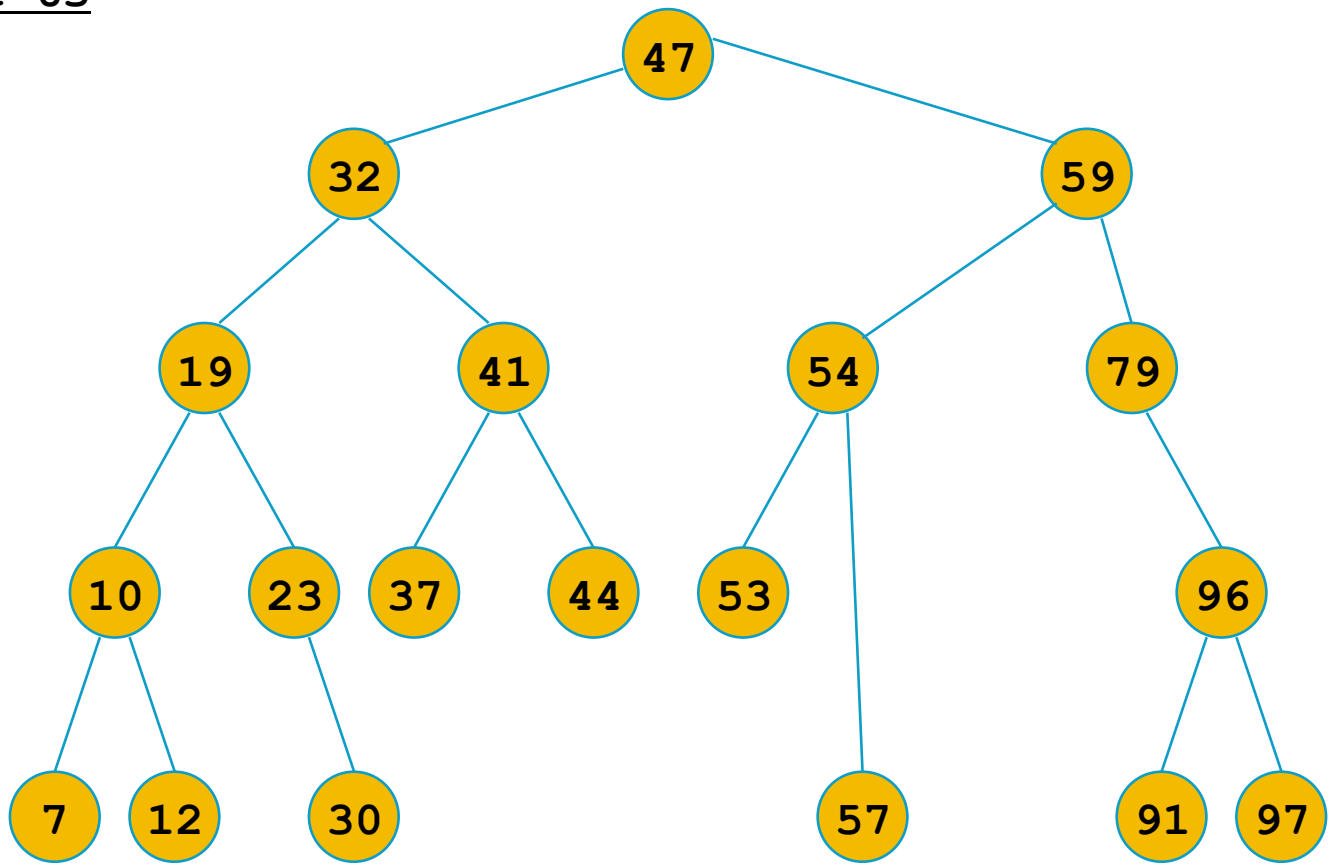
delete 63

- find predecessor
- attach subtree
- attach children
- **attach pre. to target's parent**



BST Deletion – target has 2 children

`delete 63`

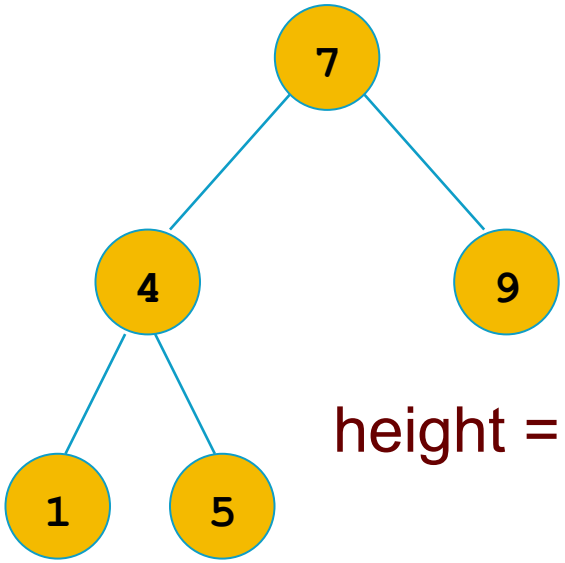


BST Efficiency

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

Height of a BST

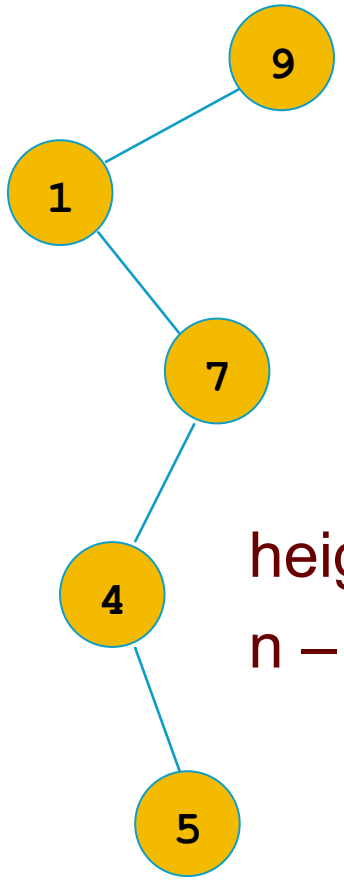
- Insert 7
- Insert 4
- Insert 1
- Insert 9
- Insert 5
- It's a complete tree!



height = $\lfloor \log(5) \rfloor = 2$

Height of a BST

- Insert 9
- Insert 1
- Insert 7
- Insert 4
- Insert 5
- It's a linked list with a lot of extra pointers!



height =
n – 1 = 4 = O(**n**)

Balanced BSTs

- It would be ideal if a BST was always close to complete
  - i.e. balanced
- How do we guarantee a balanced BST?
  - We have to make the insertion and deletion algorithms more complex
    - e.g. **`red - black`** trees.

Sorting and Binary Search Trees

- It is possible to sort an array using a binary search tree
  - Insert the array items into an empty tree
  - Write the data from the tree back into the array using an InOrder traversal
- Running time = **n***(insertion cost) + traversal
  - Insertion cost is O(**h**)
  - Traversal is O(**n**)
  - Total = O(**n**) * O(**h**) + O(**n**), i.e. O(**n** * **h**)
  - If the tree is balanced = O(**n** * log(**n**))

Tree Quiz

Tree Quiz I

- Write a recursive function to print the items in a BST in *descending* order

```
class Node {
  public:
     int data;
     Node *leftc;
     Node *rightc;
};
```

Tree Quiz II

- Write a recursive function to **delete** a BST stored in dynamic memory

```
class Node {
  public:
     int data;
     Node *leftc;
     Node *rightc;
};
```

Summary

Summary

- Trees
  - Terminology: paths, height, node relationships, …
- Binary search trees
  - Traversal
    - Post-order, pre-order, in-order
  - Operations
    - Insert, delete, search
- Balanced trees
  - Binary search tree operations are efficient for balanced trees

Readings

- Carrano Ch. 10