Structure and Insertion

Binary Search Trees 1

Binary Search Trees

- Description
- Search
- Insertion
- Efficiency
- Removal (next presentation)

Binary Tree Implementation

- The binary tree ADT can be implemented using different data structures
  - Reference structures (similar to linked lists)
  - Arrays
- Example implementations
  - Binary search trees (references)
  - Red – black trees (references again)
  - Heaps (arrays) – not a binary *search* tree
  - B trees (arrays again) – not a *binary* search tree

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

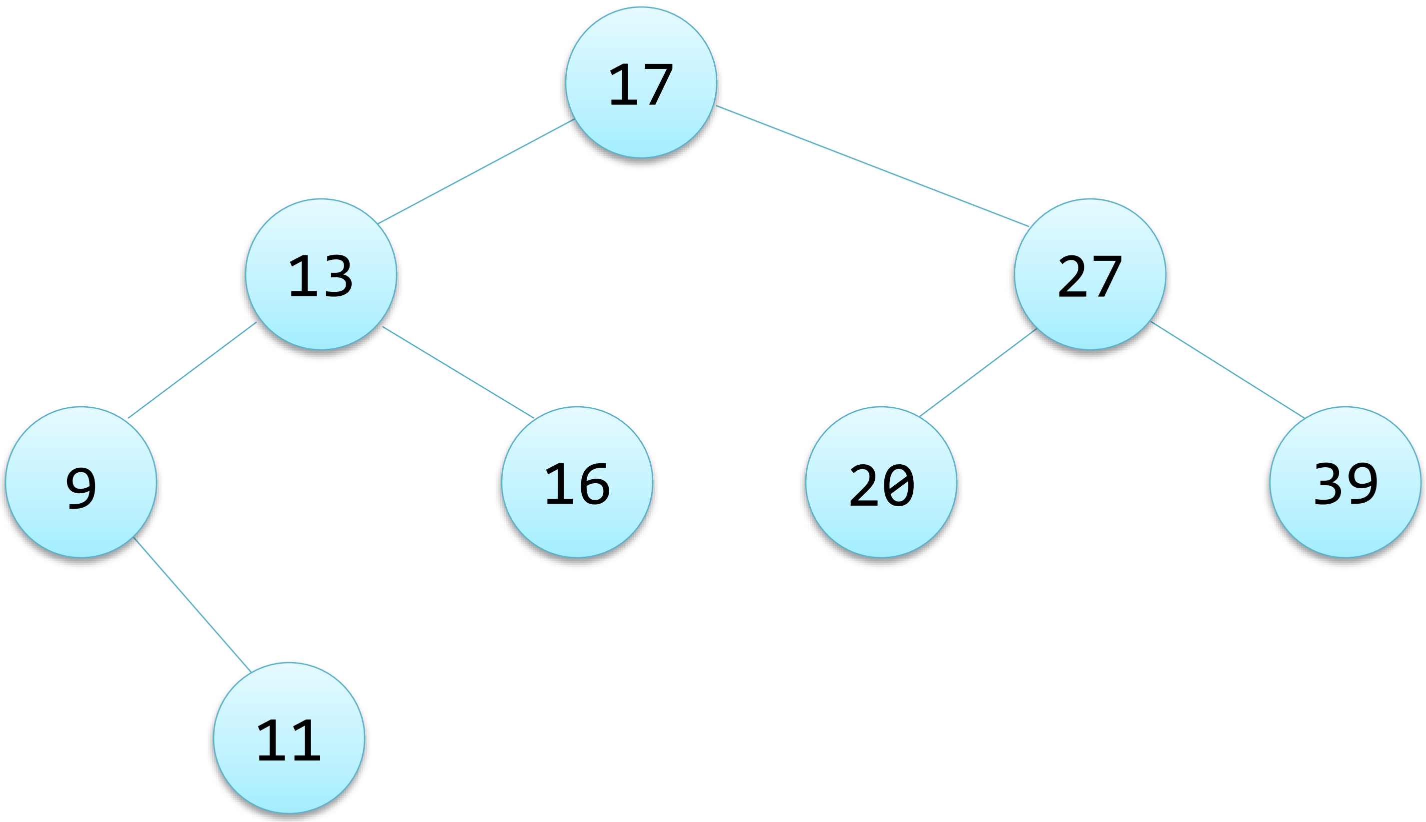
Dictionary Operations

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

Binary Search Tree Property

- A binary search tree 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 subtree's root
    - All nodes in a right subtree have labels *greater* than or equal to the label of the subtree's root
- Binary search trees are fully ordered

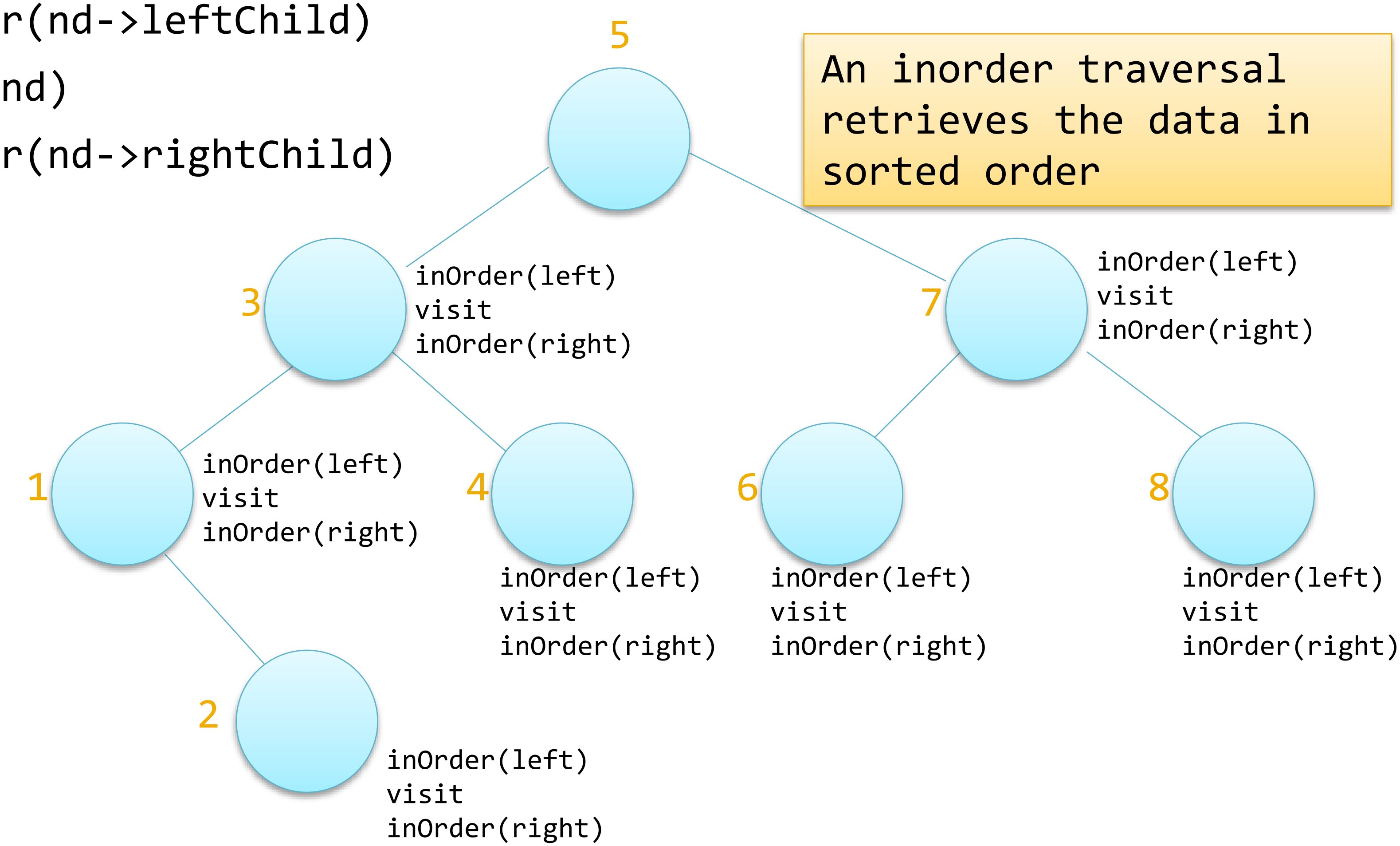
BST Example



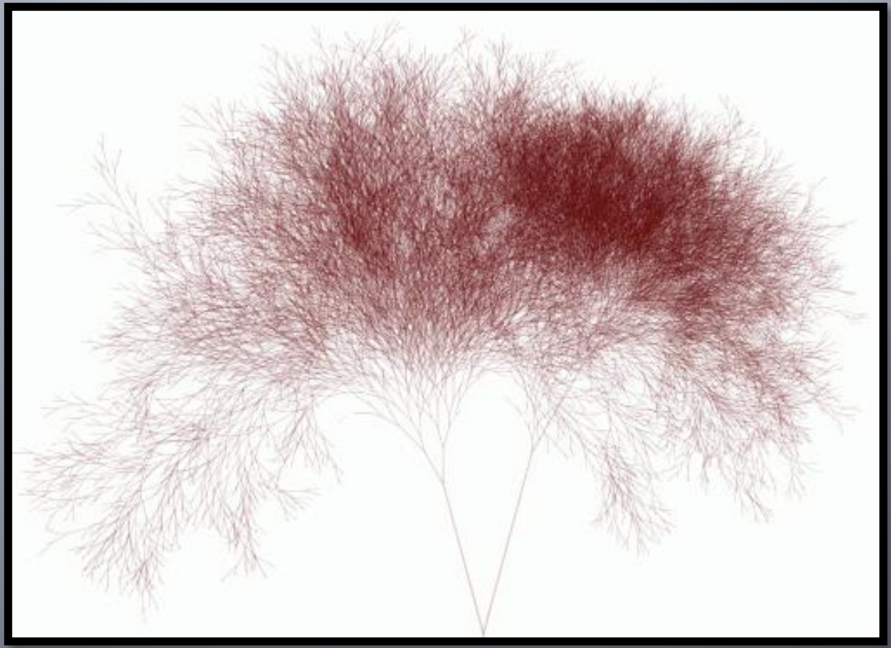
BST InOrder Traversal

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

An inorder traversal retrieves the data in sorted order

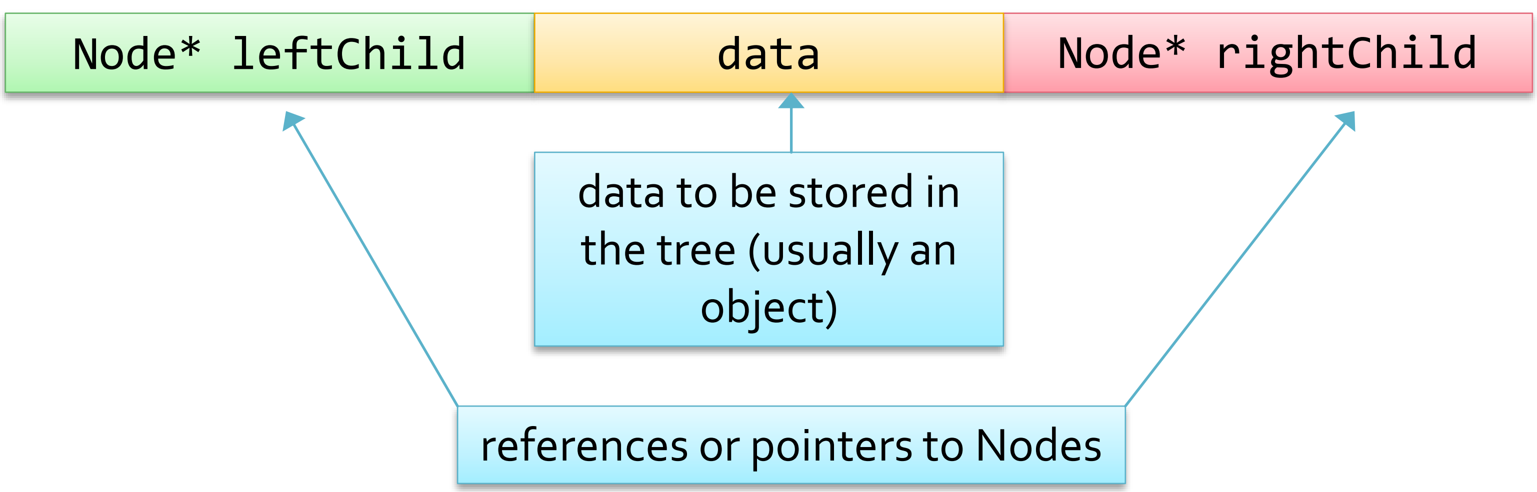


Binary Search Tree Search



BST Implementation

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

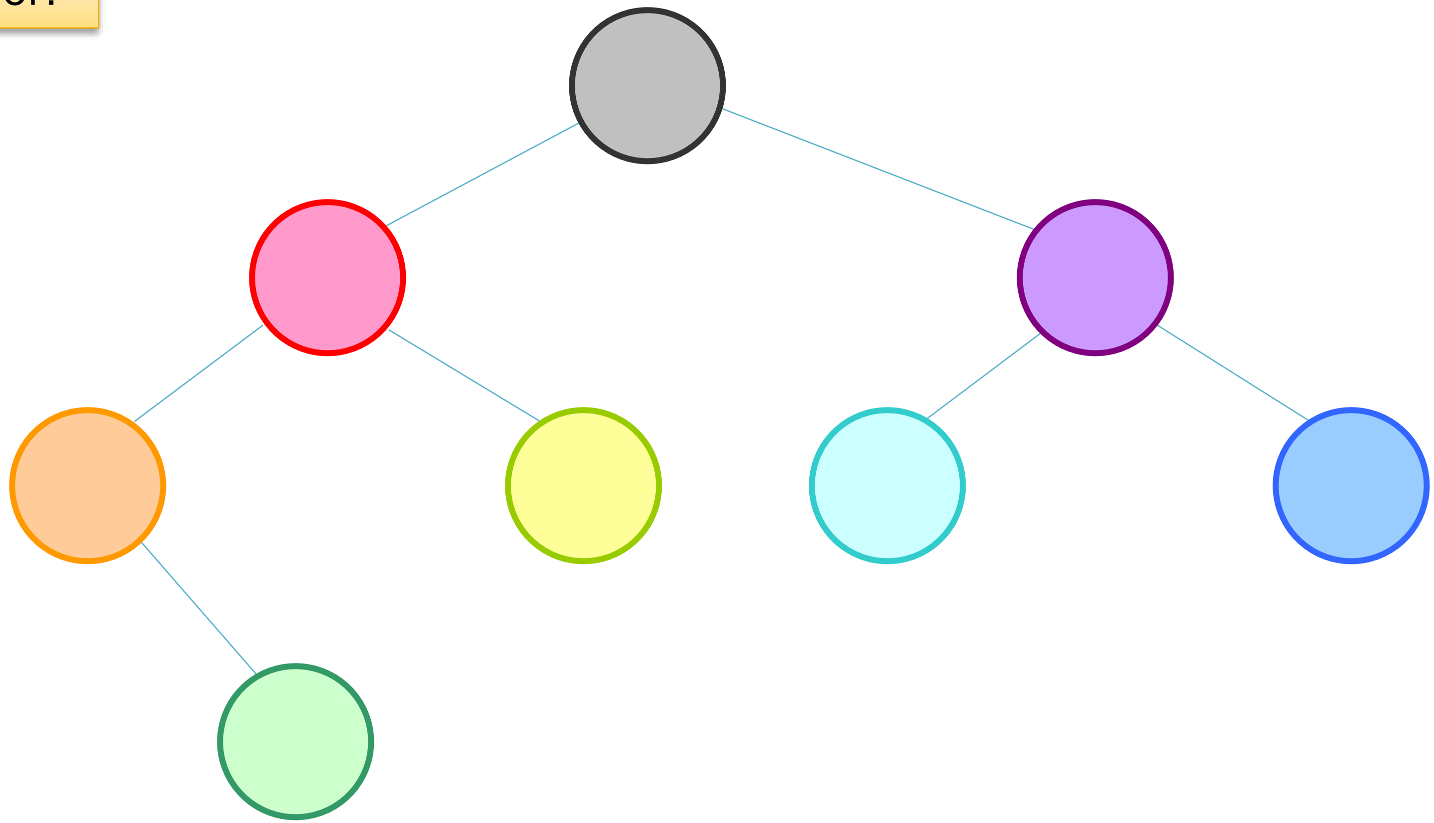


BST Search

- 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 a pointer to the data, or …)
- How many comparisons?
  - One for each node on the path
  - Worst case: height of the tree + 1

BST Search Example

Search for?



BST Search Algorithm

```
bool search(Node* nd, int x){
        if (nd == NULL){
                return false;
        }else if(x == nd->data){
                return true;
        } else if (x < nd->data){
                return search(nd->left, x);
        } else {
                return search(nd->right, x);
        }
}
```

reached the end of this path

note the similarity to binary search

called by a helper method like this: *search(root, target)*

BST Insertion

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(*height*)

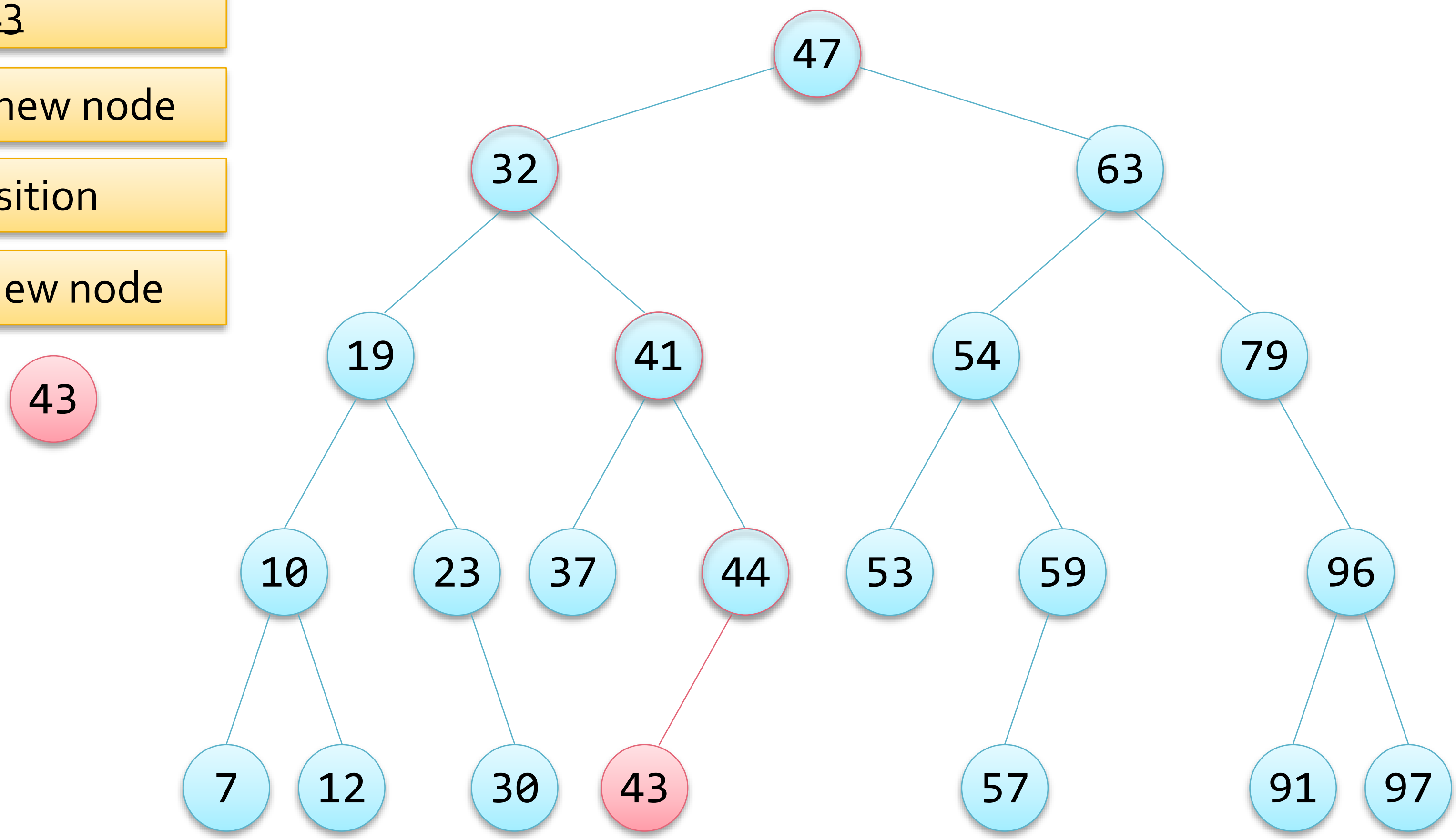
BST Insertion Example

insert 43

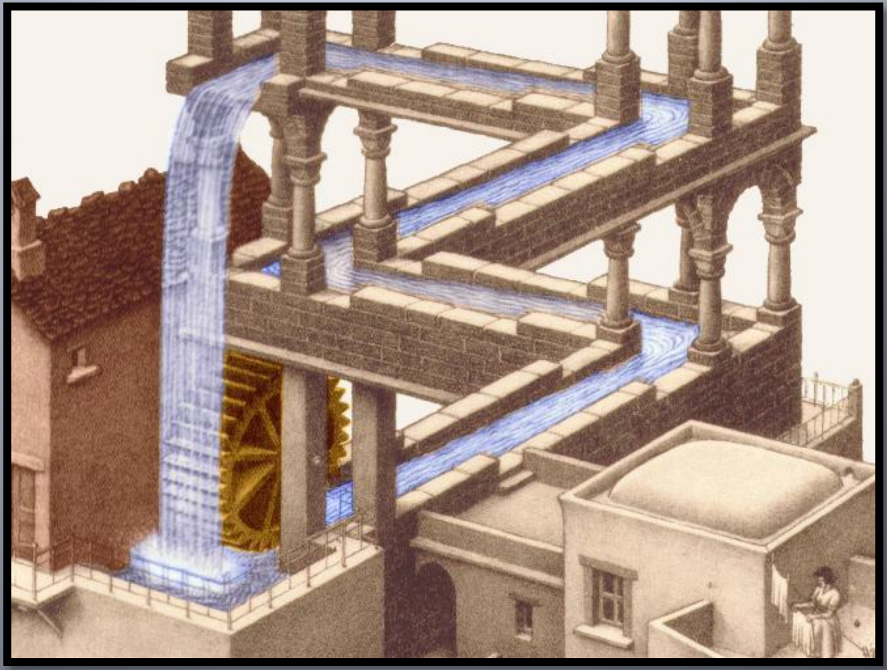
create new node

find position

insert new node



BST Efficiency

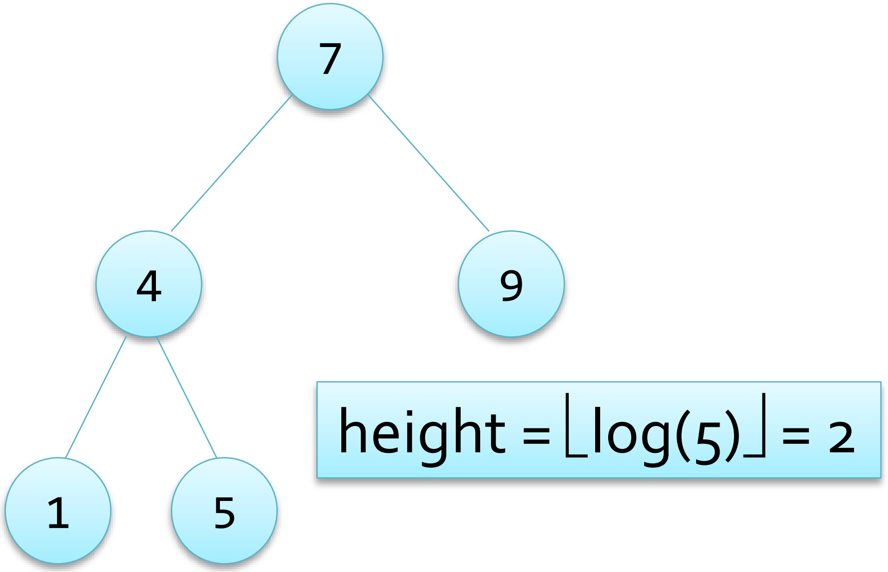


BST Efficiency

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

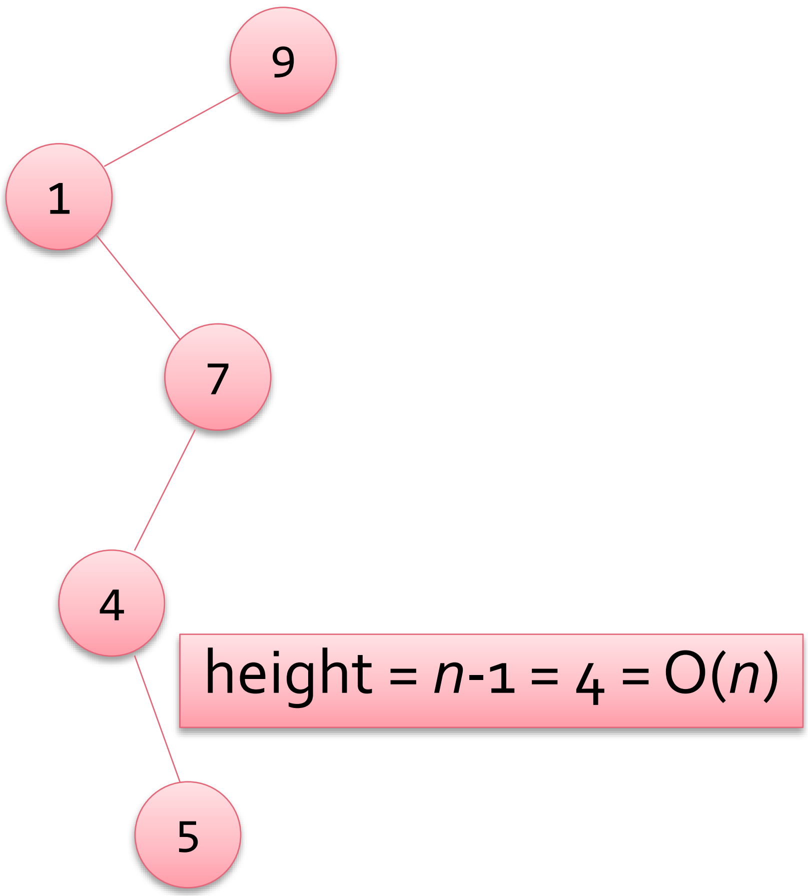
Height of a BST

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



Height of a BST

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



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 structure and / or the insertion and removal 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))