Binary Search Trees

Part Two

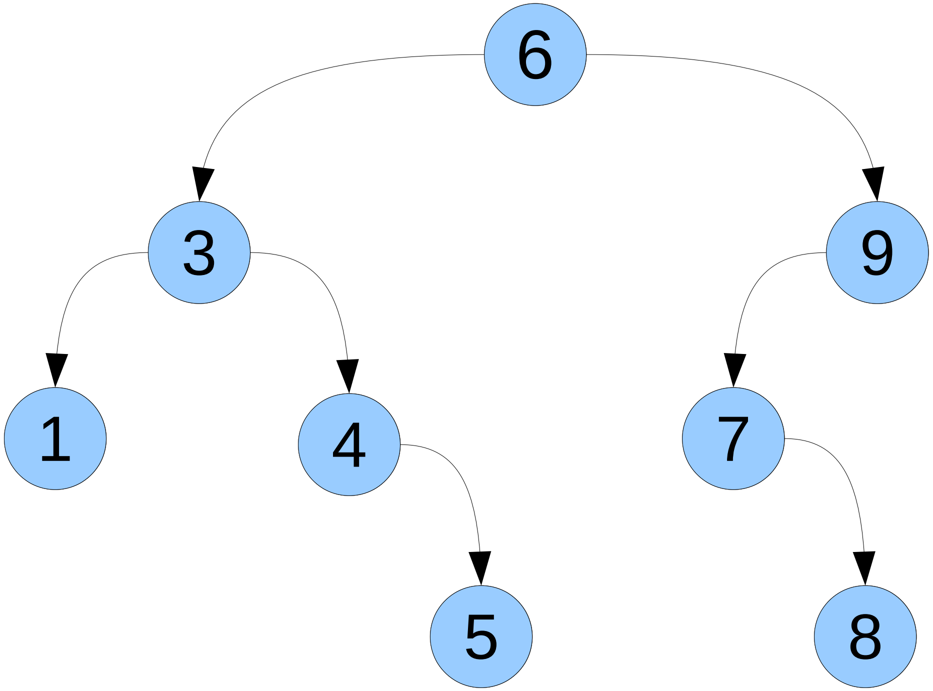
Outline for Today

- ***Freeing Trees***
    - Cleaning up our messes.
- ***Balanced Trees***
    - How fast are BST operations?
- ***Range Searches***
    - A useful hybrid algorithm.

Recap from Last Time

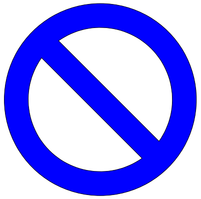
Binary Search Trees

- The data structure we have just seen is called a ***binary search tree*** (or ***BST***).
- The tree consists of a number of ***nodes***, each of which stores a value and has zero, one, or two ***children***.
- All values in a node's left subtree are ***smaller*** than the node's value, and all values in a node's right subtree are ***greater*** than the node's value.

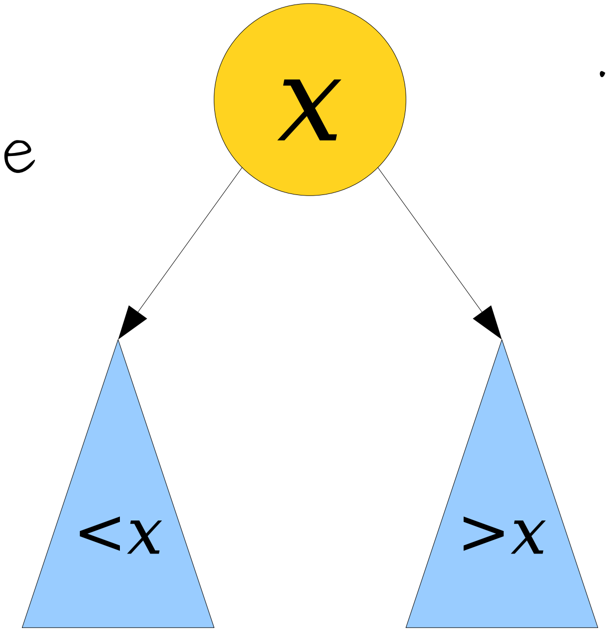


A Binary Search Tree Is Either…

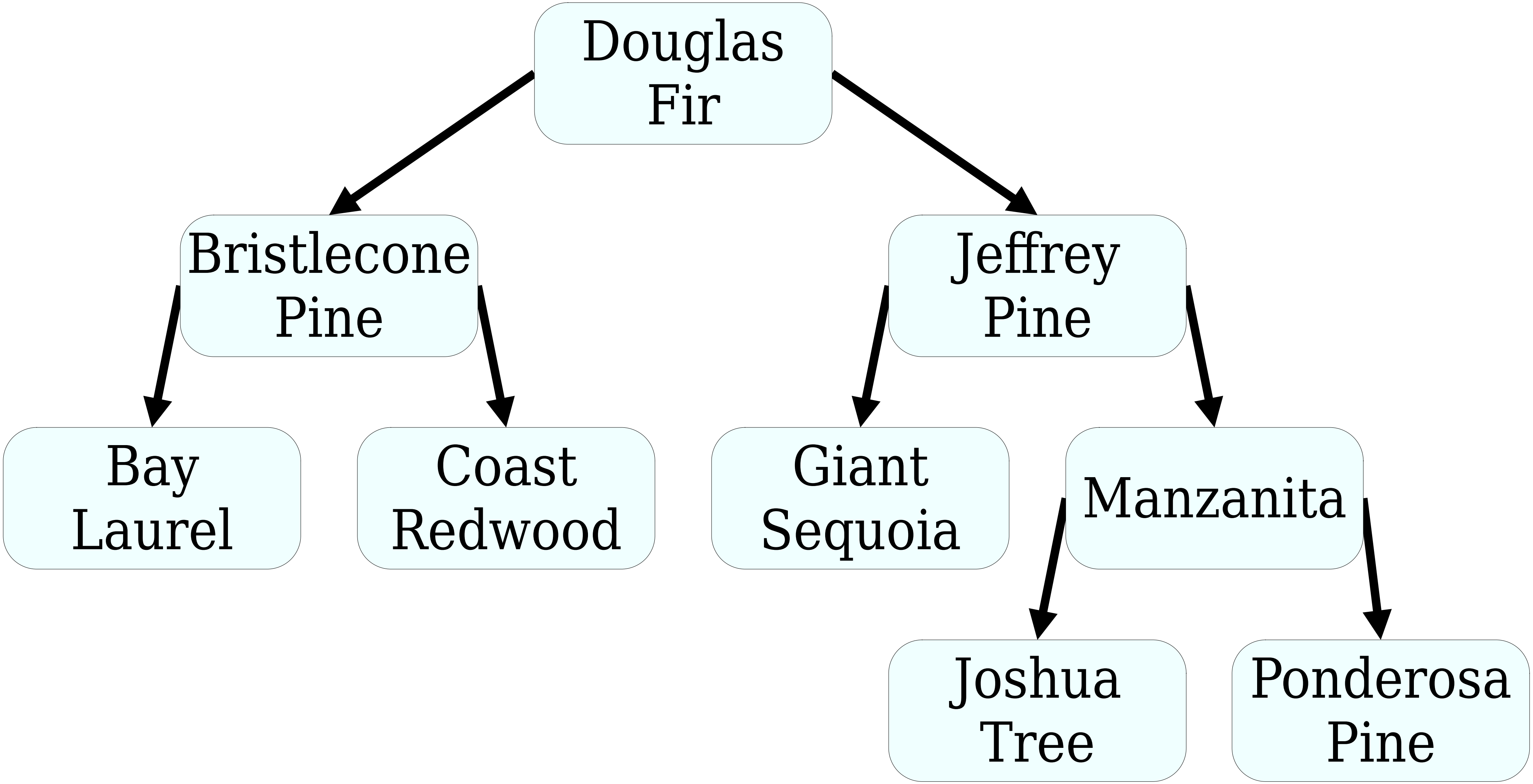
an empty tree,
represented by
`nullptr`, or…



… a single node,
whose left subtree
is a BST of
smaller values …



… and whose right
subtree is a BST
of larger values.



Douglas Fir

Bristlecone Pine

Jeffrey Pine

Bay Laurel

Coast Redwood

Giant Sequoia

Manzanita

Joshua Tree

Ponderosa Pine

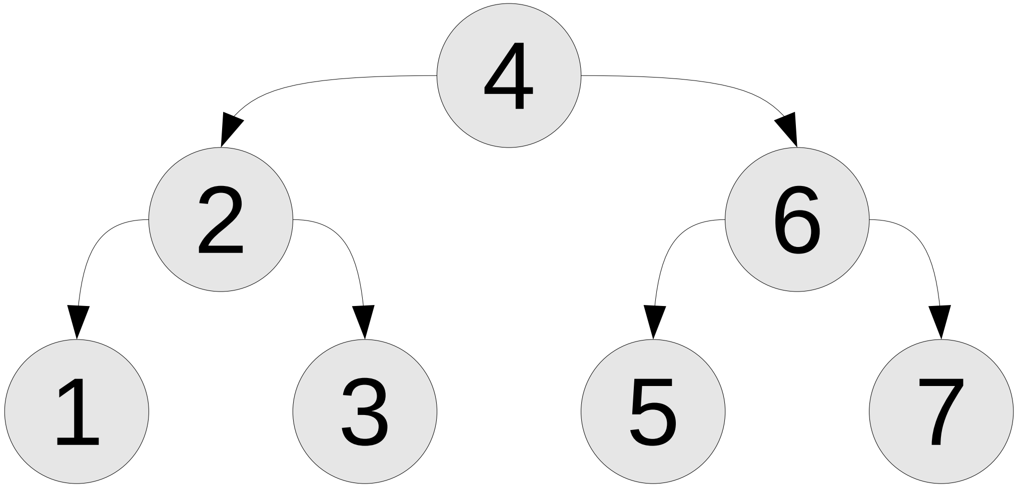
New Stuff!

Getting Rid of Trees

http://www.tigersheds.com/garden-resources/image.axd?picture=2010%2F6%2Fdeforestation1.jpg

Freeing a Tree

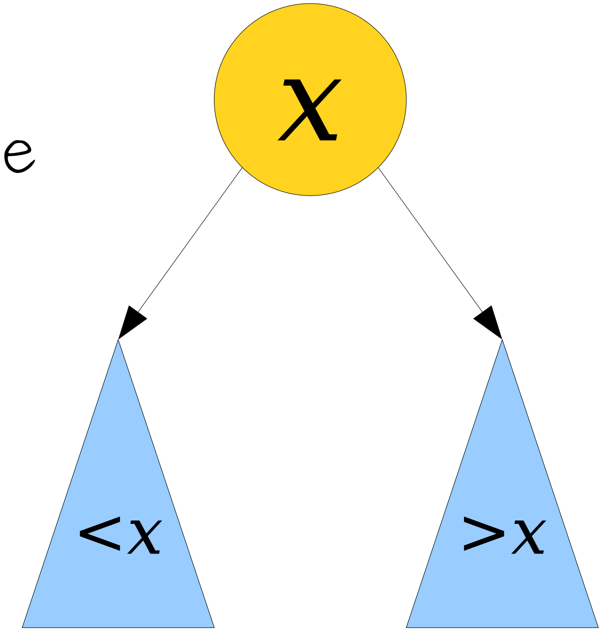
- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



A Binary Search Tree Is Either…

an empty tree,
represented by
`nullptr`, or… 🚫

… a single node,
whose left subtree
is a BST of
smaller values …



… and whose right
subtree is a BST
of larger values.

```
void deleteTree(Node* root) {
  if (root == nullptr) return;

  delete root;
  deleteTree(root->left);
  deleteTree(root->right);
}
```

A

```
void deleteTree(Node* root) {
  if (root == nullptr) return;

  delete root;
  deleteTree(root->right);
  deleteTree(root->left);
}
```

B

```
void deleteTree(Node* root) {
  if (root == nullptr) return;

  deleteTree(root->left);
  delete root;
  deleteTree(root->right);
}
```

C

```
void deleteTree(Node* root) {
  if (root == nullptr) return;

  deleteTree(root->right);
  delete root;
  deleteTree(root->left);
}
```

D

```
void deleteTree(Node* root) {
  if (root == nullptr) return;

  deleteTree(root->left);
  deleteTree(root->right);
  delete root;
}
```

E

```
void deleteTree(Node* root) {
  if (root == nullptr) return;

  deleteTree(root->right);
  deleteTree(root->left);
  delete root;
}
```

F

Which of these options work?
Formulate a hypothesis!

```
void deleteTree(Node* root) {
  if (root == nullptr) return;

  delete root;
  deleteTree(root->left);
  deleteTree(root->right);
}
```

A

```
void deleteTree(Node* root) {
  if (root == nullptr) return;

  delete root;
  deleteTree(root->right);
  deleteTree(root->left);
}
```

B

```
void deleteTree(Node* root) {
  if (root == nullptr) return;

  deleteTree(root->left);
  delete root;
  deleteTree(root->right);
}
```

C

```
void deleteTree(Node* root) {
  if (root == nullptr) return;

  deleteTree(root->right);
  delete root;
  deleteTree(root->left);
}
```

D

```
void deleteTree(Node* root) {
  if (root == nullptr) return;

  deleteTree(root->left);
  deleteTree(root->right);
  delete root;
}
```

E

```
void deleteTree(Node* root) {
  if (root == nullptr) return;

  deleteTree(root->right);
  deleteTree(root->left);
  delete root;
}
```

F

Which of these options work?
Discuss with your neighbors!

```
void deleteTree(Node* root) {
  if (root == nullptr) return;

  deleteTree(root->left);
  deleteTree(root->right);
  delete root;
}
```

✓

```
void deleteTree(Node* root) {
  if (root == nullptr) return;

  deleteTree(root->right);
  deleteTree(root->left);
  delete root;
}
```

✓

Which of these options work?

Discuss with your neighbors!

Postorder Traversals

- The particular recursive pattern we just saw is called a ***postorder traversal*** of a binary tree.
- Specifically:
  - Recursively visit all the nodes in the two subtrees, in whichever order you'd like.
  - Visit the node itself.
- This contrasts with the ***inorder traversal*** we used to print the contents of a BST.
  - That's where we recursively visit the left subtree, then the node itself, then the right subtree.

Tree Efficiency

How fast are BST lookups?

How fast are BST insertions?

Building a BST

- First, draw the BST formed by inserting the values 1, 3, 5, 7, 2, 4, 6 into an empty tree.
- Then draw what you get if you insert the values 4, 6, 5, 2, 1, 7, and 3 into an empty tree.

Formulate a hypothesis!

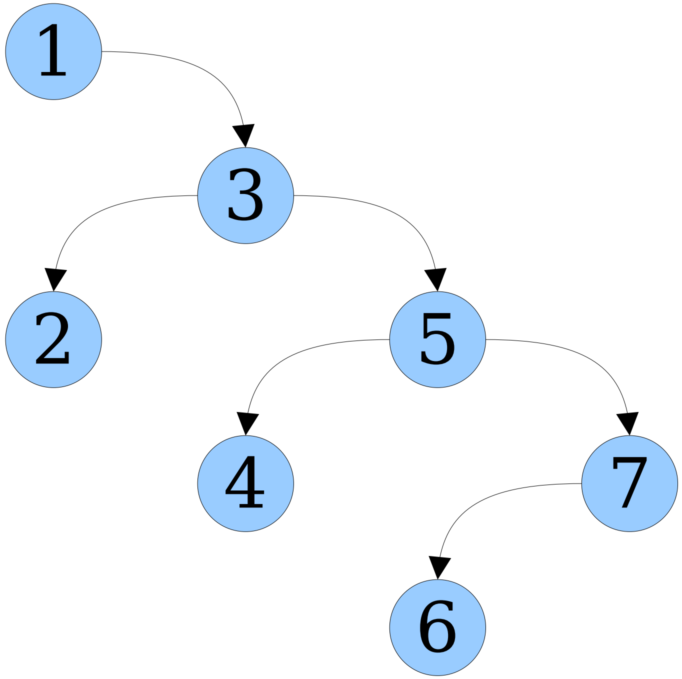
Building a BST

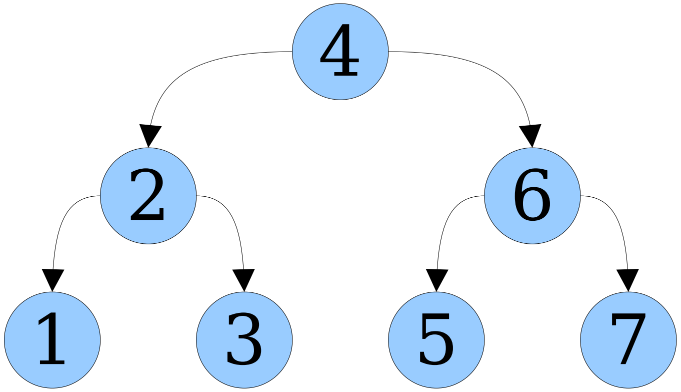
- First, draw the BST formed by inserting the values 1, 3, 5, 7, 2, 4, 6 into an empty tree.
- Then draw what you get if you insert the values 4, 6, 5, 2, 1, 7, and 3 into an empty tree.

Discuss with your neighbors!

Building a BST

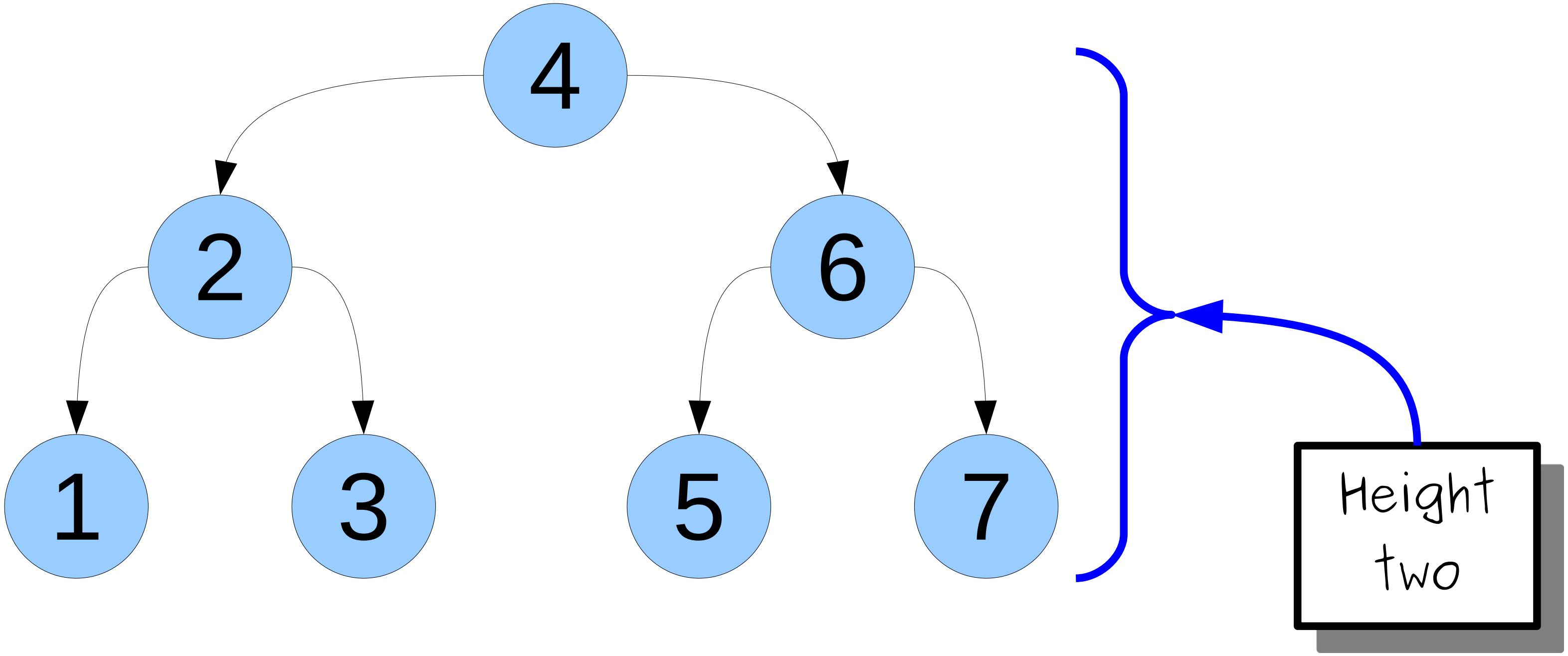
- First, draw the BST formed by inserting the values 1, 3, 5, 7, 2, 4, 6 into an empty tree.
- Then draw what you get if you insert the values 4, 6, 5, 2, 1, 7, and 3 into an empty tree.





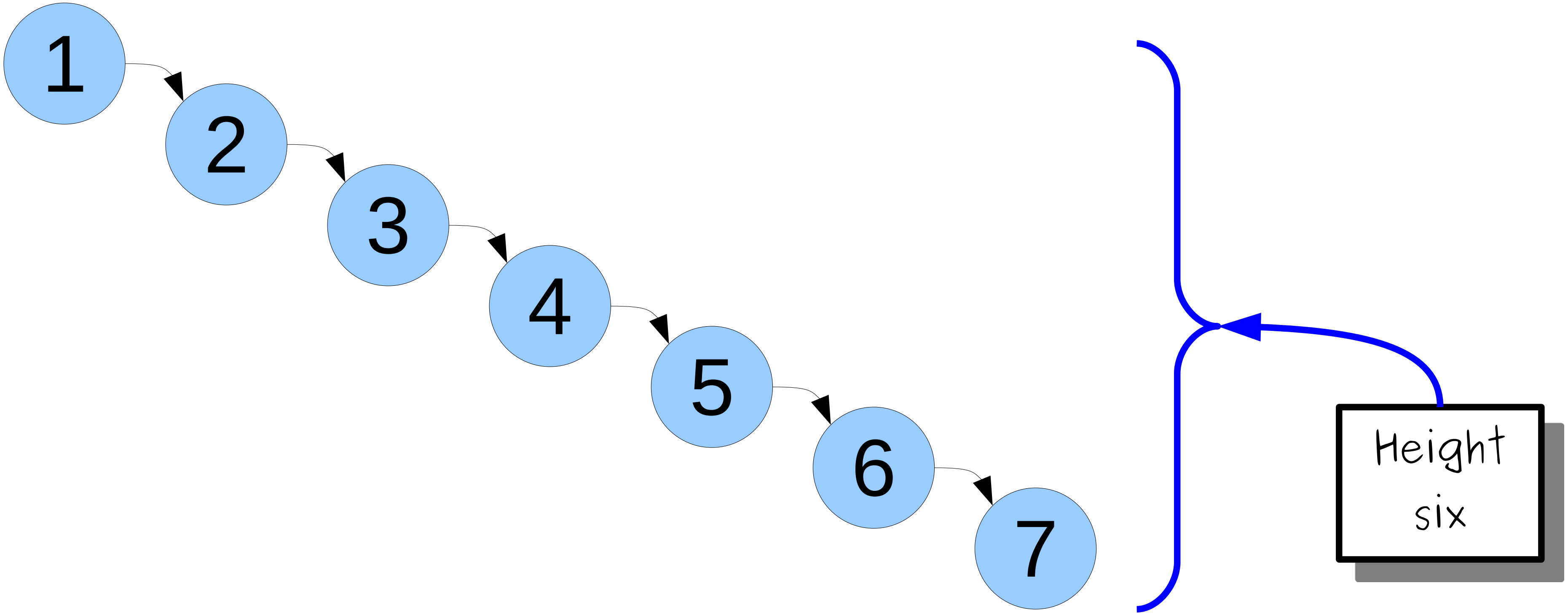
Tree Terminology

- The ***height*** of a tree is the number of links in the longest path from the root to a leaf.



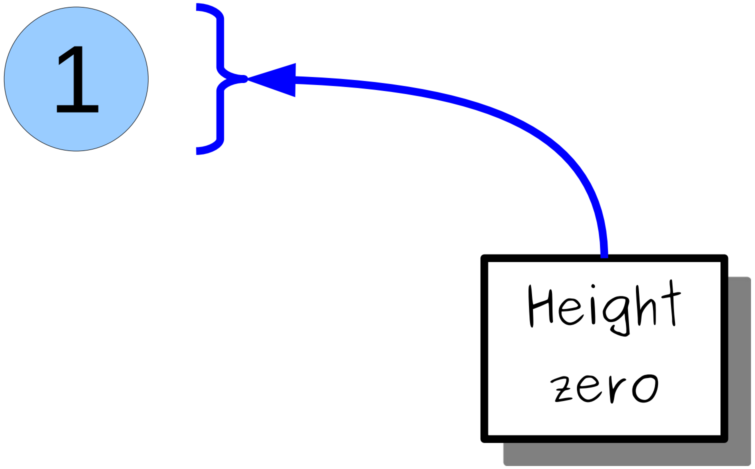
Tree Terminology

- The ***height*** of a tree is the number of links in the longest path from the root to a leaf.



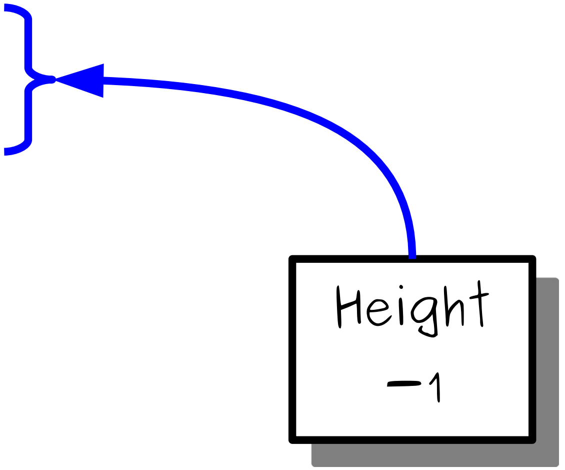
Tree Terminology

- The ***height*** of a tree is the number of links in the longest path from the root to a leaf.



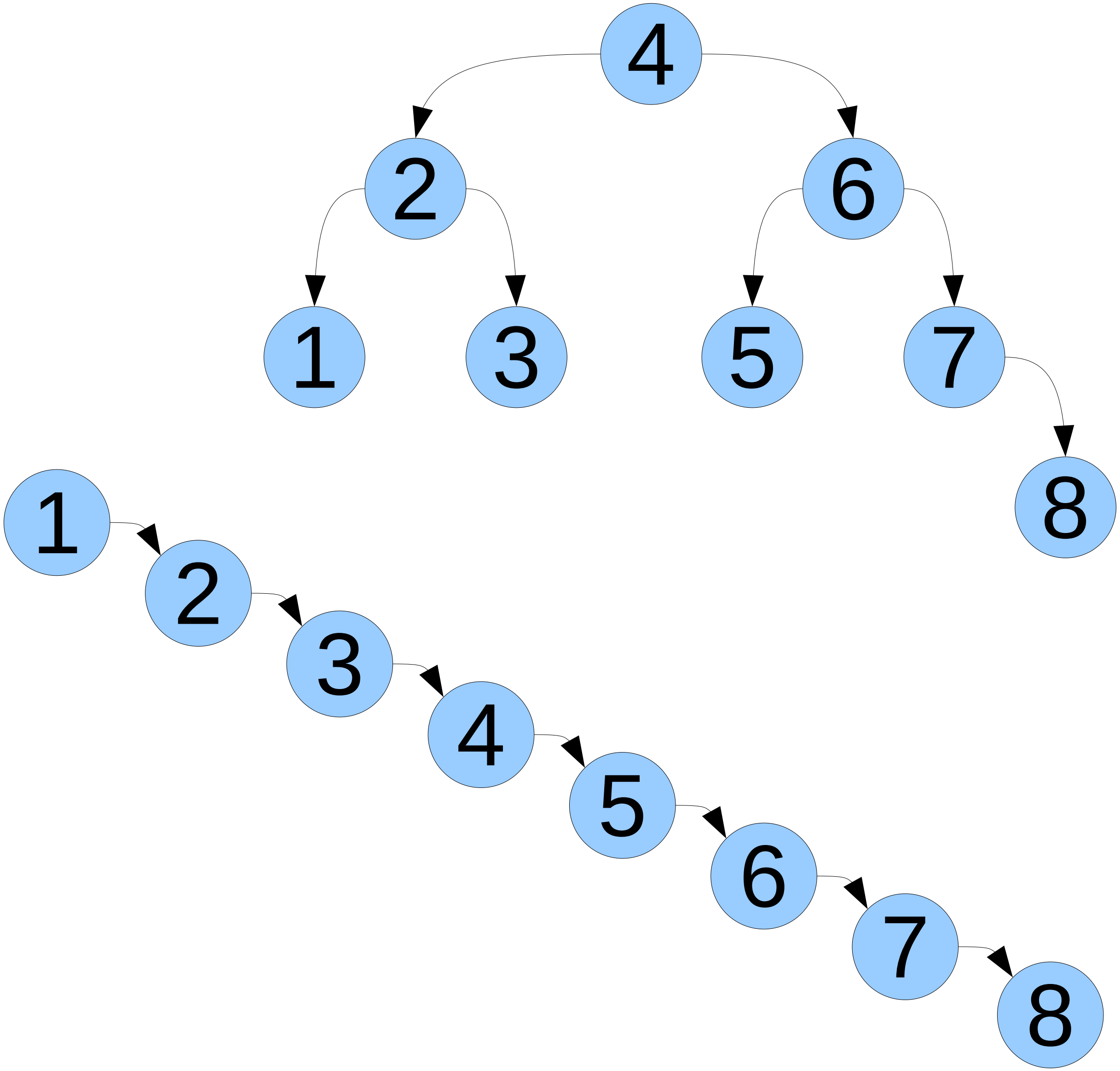
Tree Terminology

- The ***height*** of a tree is the number of links in the longest path from the root to a leaf.
- By convention, an empty tree has height -1.



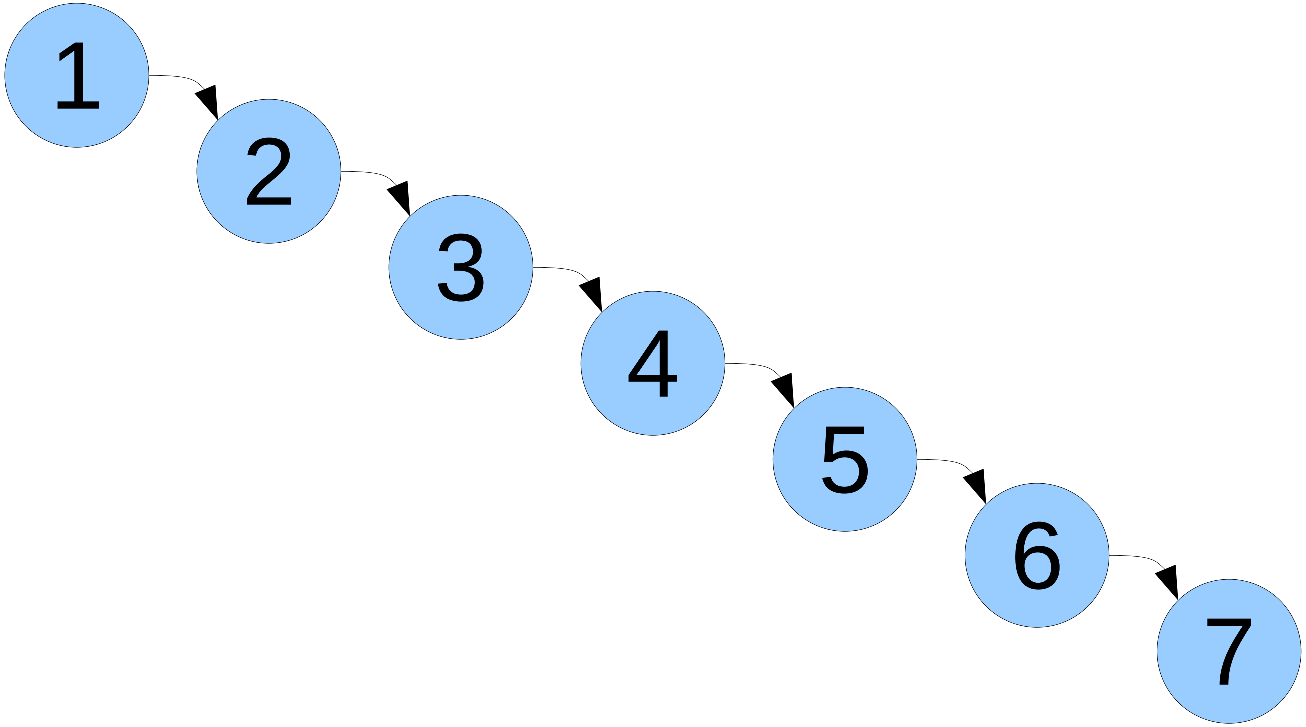
Efficiency Questions

- The time to add an element to a BST (or look up an element in a BST) depends on the height of the tree.
- The runtime is **O(*h*)**, where *h* is the height of the tree.



Tree Heights

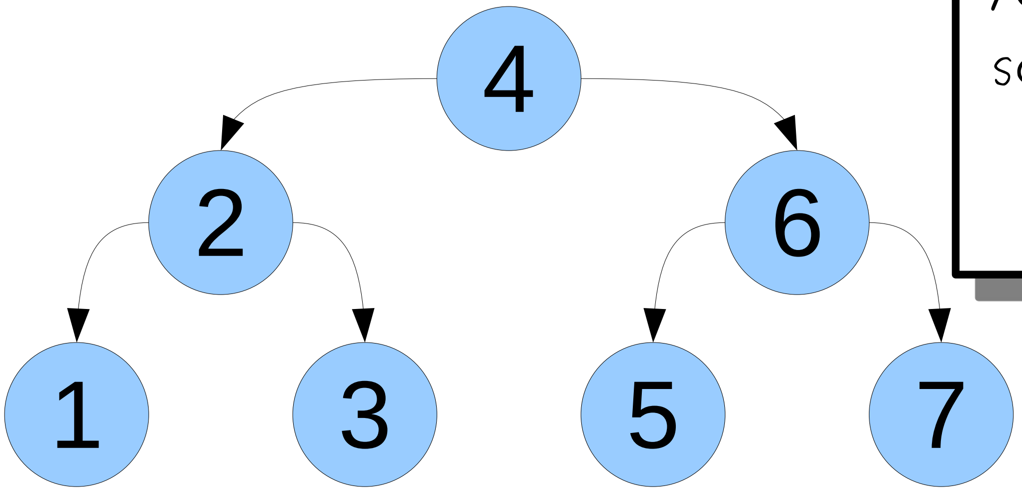
- What is the maximum and minimum possible height of a tree with *n* nodes?
- Maximum height: all nodes in a chain. Height is O(*n*).



Tree Heights

- What is the maximum and minimum possible height of a tree with *n* nodes?
- Maximum height: all nodes in a chain. Height is O(*n*).
- Minimum height: tree is as complete as possible. Height is O(log *n*).

You can only double something O(log *n*) times before it exceeds *n*.



Balanced Trees

- A binary search tree is called ***balanced*** if its height is O(log n), where n is the number of nodes in the tree.
- Balanced trees are extremely efficient:
  - Lookups take time O(log n).
  - Insertions take time O(log n).
  - Deletions take time O(log n).
- ***Question:*** How do you balance a tree?

Balanced Trees

- A ***self-balancing tree*** is a BST that reshapes itself on insertions and deletions to stay balanced.
- There are many strategies for doing this. They’re beautiful. They’re clever. And they’re beyond the scope of CS106B.
- Some suggested topics to read up on, if you’re curious:
  - Red/black trees (take CS161 or CS166!)
  - AVL trees (covered in the textbook.)
  - Splay trees (trees that reshape on lookups.)
  - Scapegoat trees (yes, that’s what they’re called.)
  - Treaps (half binary heap, half binary search tree!)

What if you do no balancing at all?

A Tale of Two Trees

- We have a thermometer that gives a temperature reading at 4PM each day. We insert the temperature readings into a BST each day, starting on January 1 and ending on December 31.
- There’s a marathon race. We insert the names of the athletes into a BST as they cross the finish line.

Which BST will be more balanced?
Which BST will be less balanced?

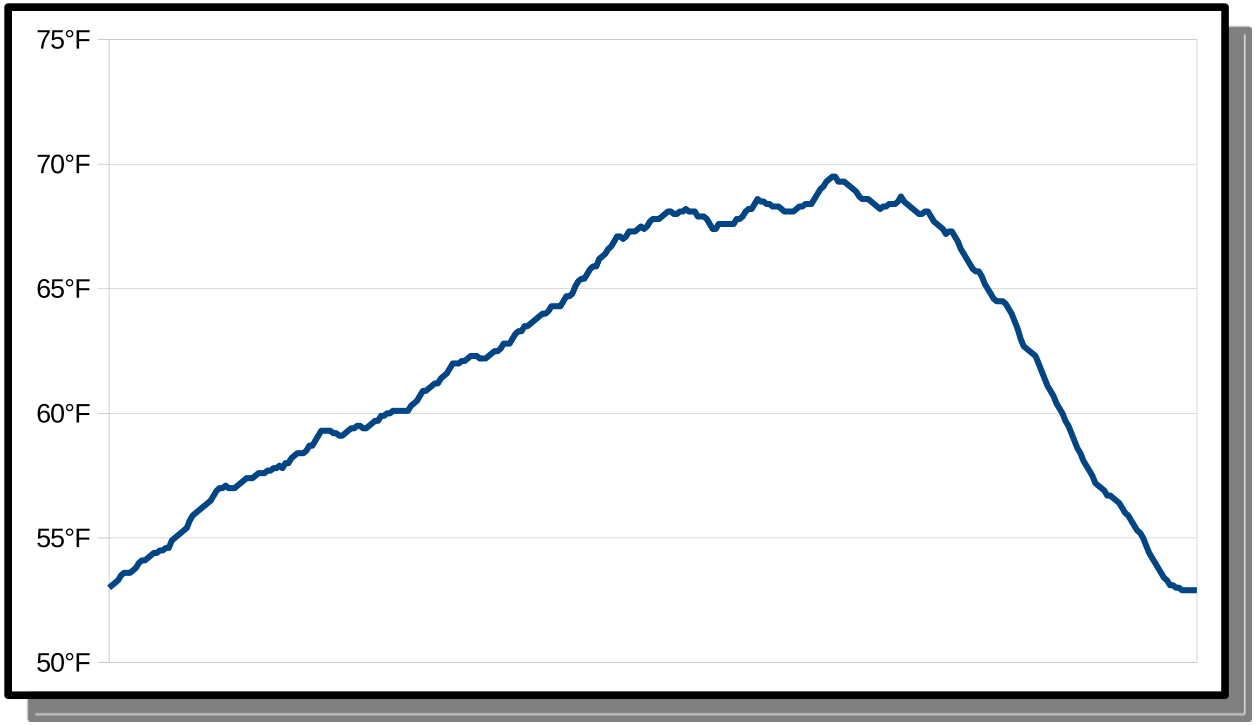
Formulate a hypothesis!

A Tale of Two Trees

- We have a thermometer that gives a temperature reading at 4PM each day. We insert the temperature readings into a BST each day, starting on January 1 and ending on December 31.
- There’s a marathon race. We insert the names of the athletes into a BST as they cross the finish line.

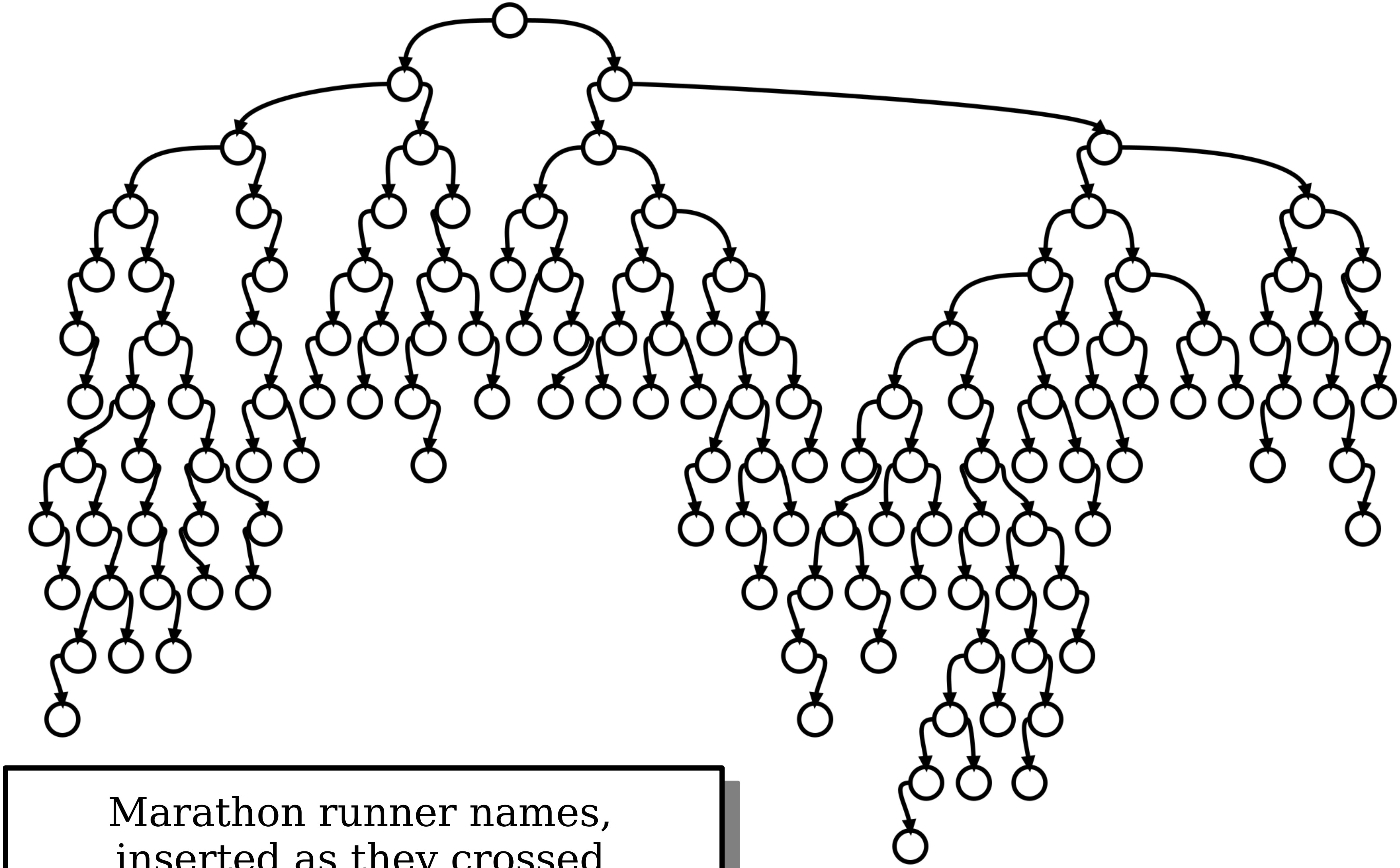
Which BST will be more balanced?
Which BST will be less balanced?

Discuss with your neighbors!



Temperature readings, inserted daily at 4PM, from January 1 to December 31.

(Data source: NOAA: SFO readings from Jan 1 – Dec 31 2010)

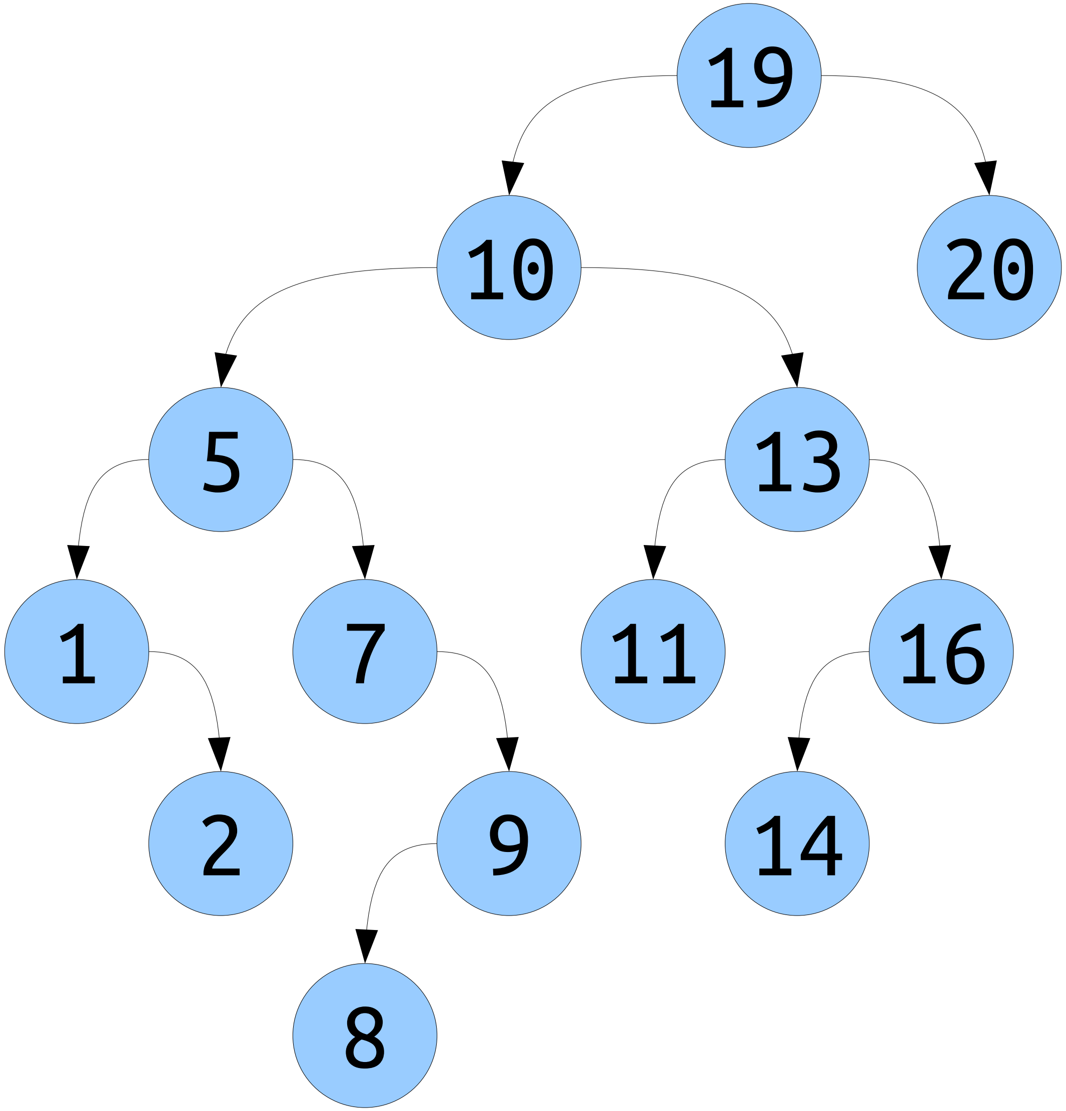


Marathon runner names,
inserted as they crossed
the finish line.

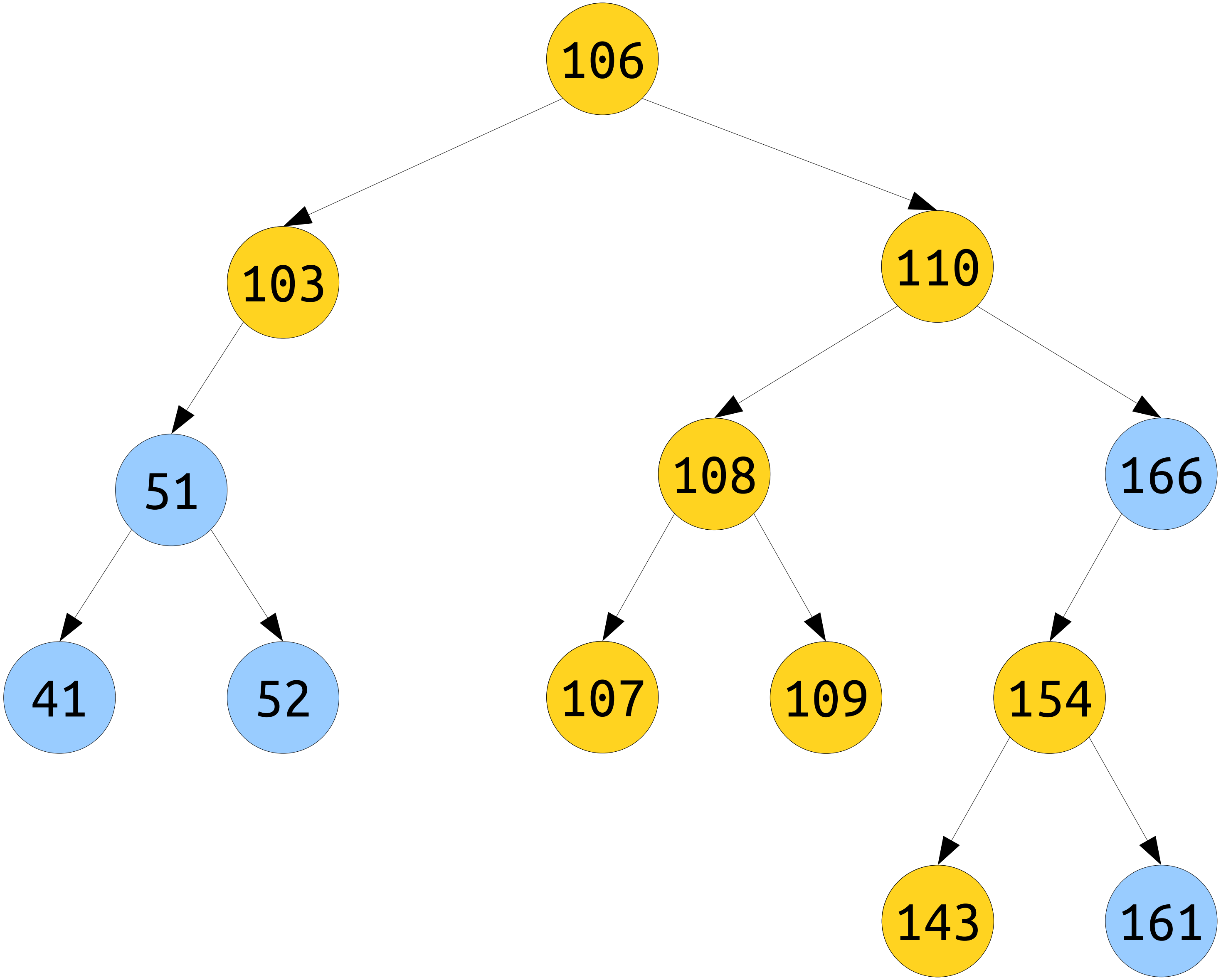
(Data source: https://www.olympic.org/rio-2016/athletics/marathon-women)

Balanced Trees

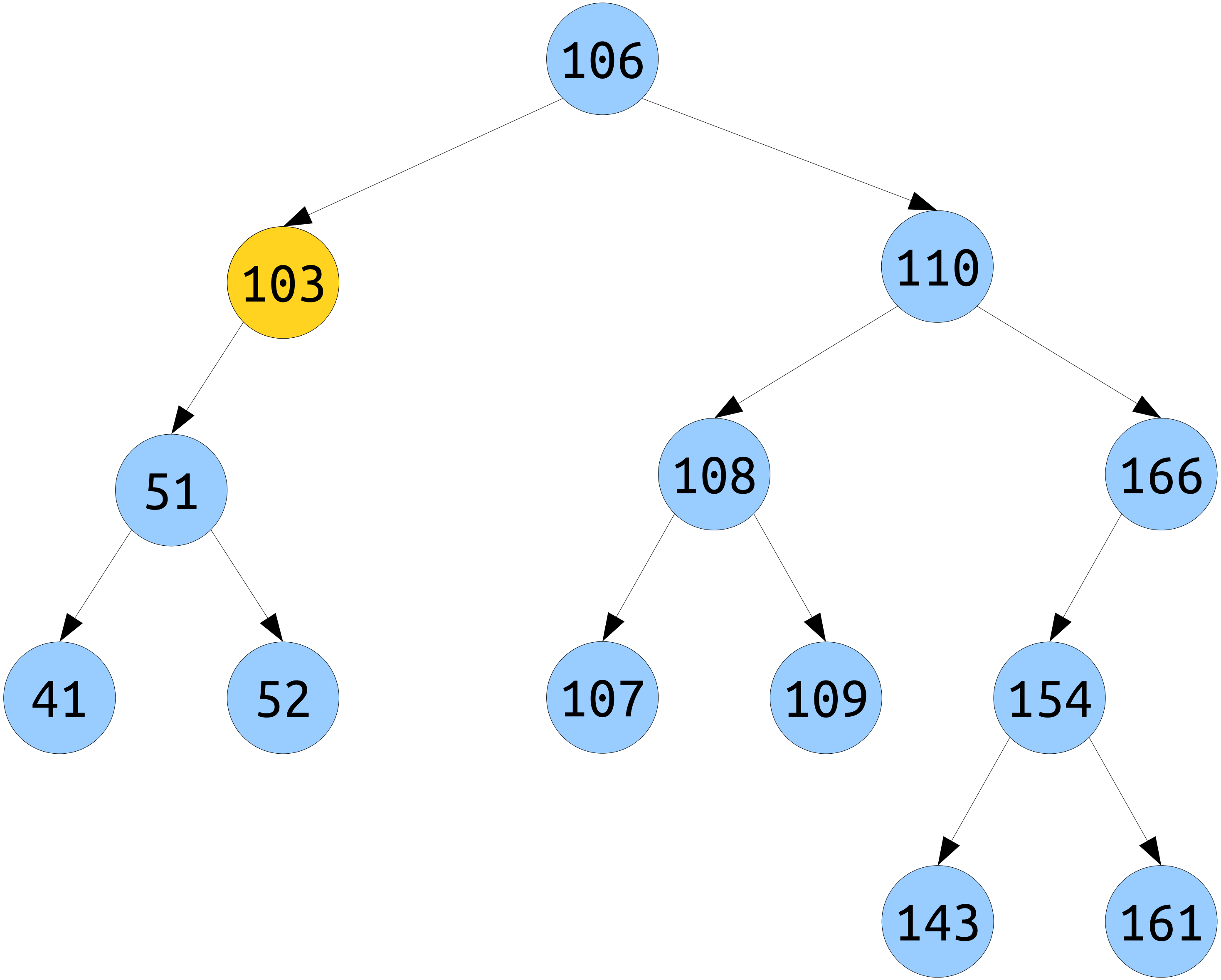
- ***Theorem:*** If you start with an empty tree and add in random values, then, with high probability, the tree is balanced.
- ***Proof:*** Take CS161!
- ***Takeaway:*** If you're adding elements to a BST and their values are actually random, then your tree is likely to be balanced.



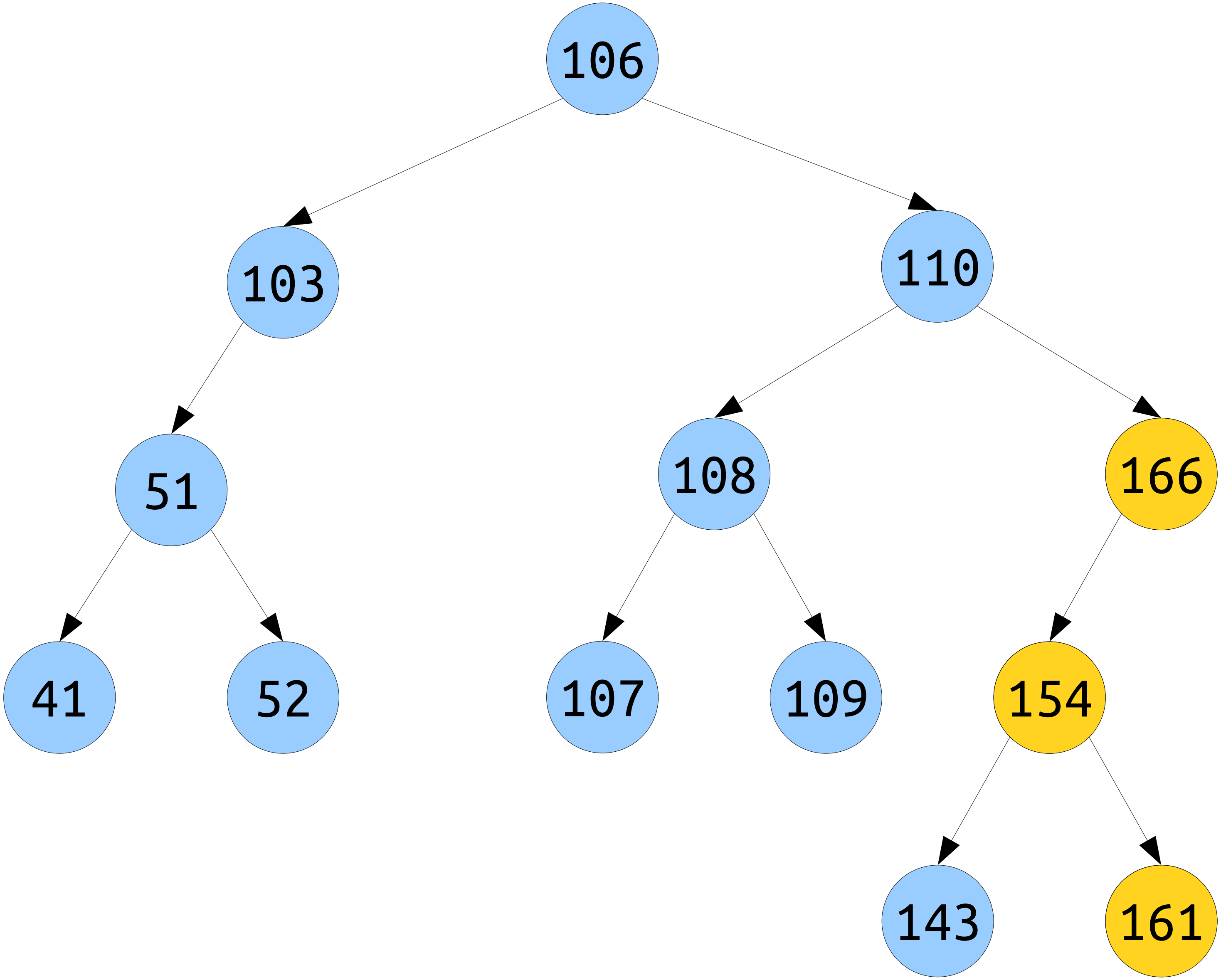
Range Searches



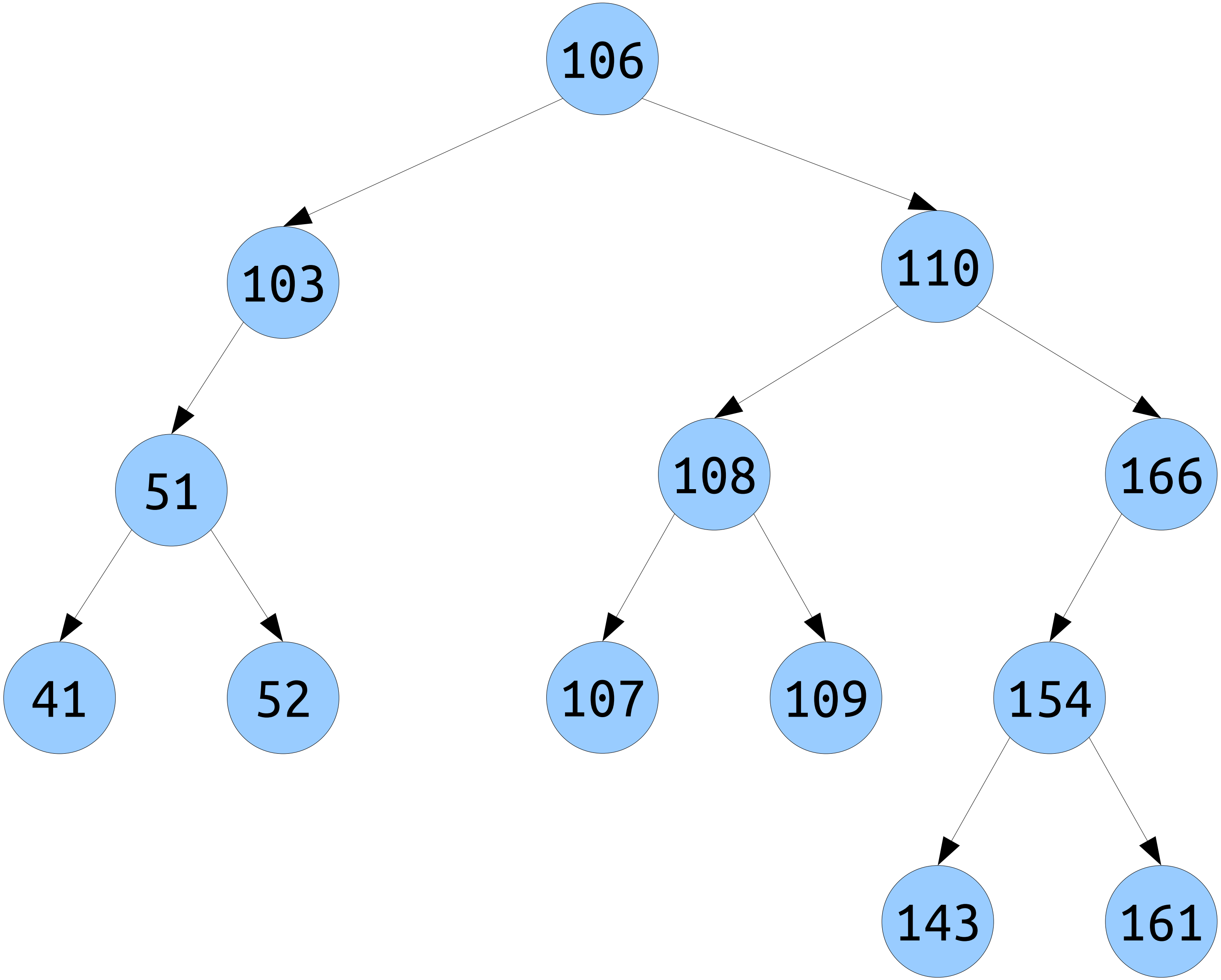
Find all elements in this tree in the range **[103, 154]**.



Find all elements in this tree in the range **[99, 105]**.



Find all elements in this tree in the range **[150, 170]**.



Find all elements in this tree in the range **[137, 138]**.

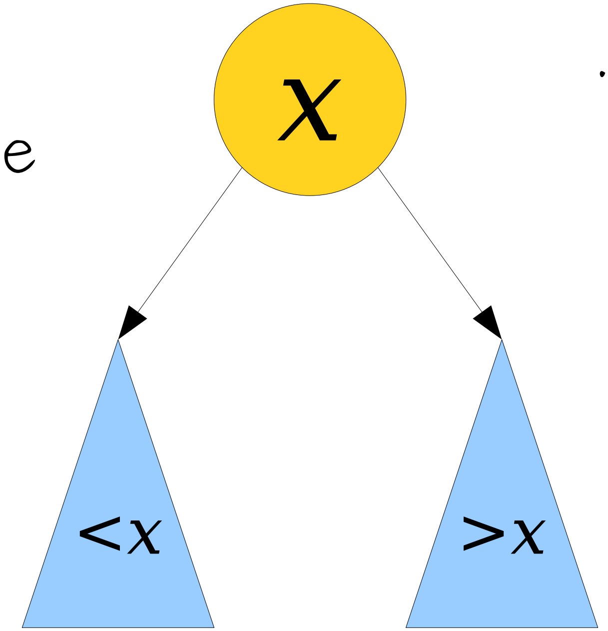
Range Searches

- We can use BSTs to do ***range searches***, in which we find all values in the BST within some range.
- For example:
  - If the values in the BST are dates, we can find all events that occurred within some time window.
  - If the values in the BST are number of diagnostic scans ordered, we can find all doctors who order a disproportionate number of scans.

A Binary Search Tree Is Either…

an empty tree,
represented by
`nullptr`, or…

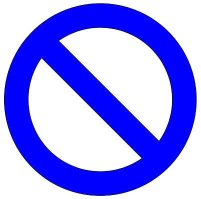
… a single node,
whose left subtree
is a BST of
smaller values …

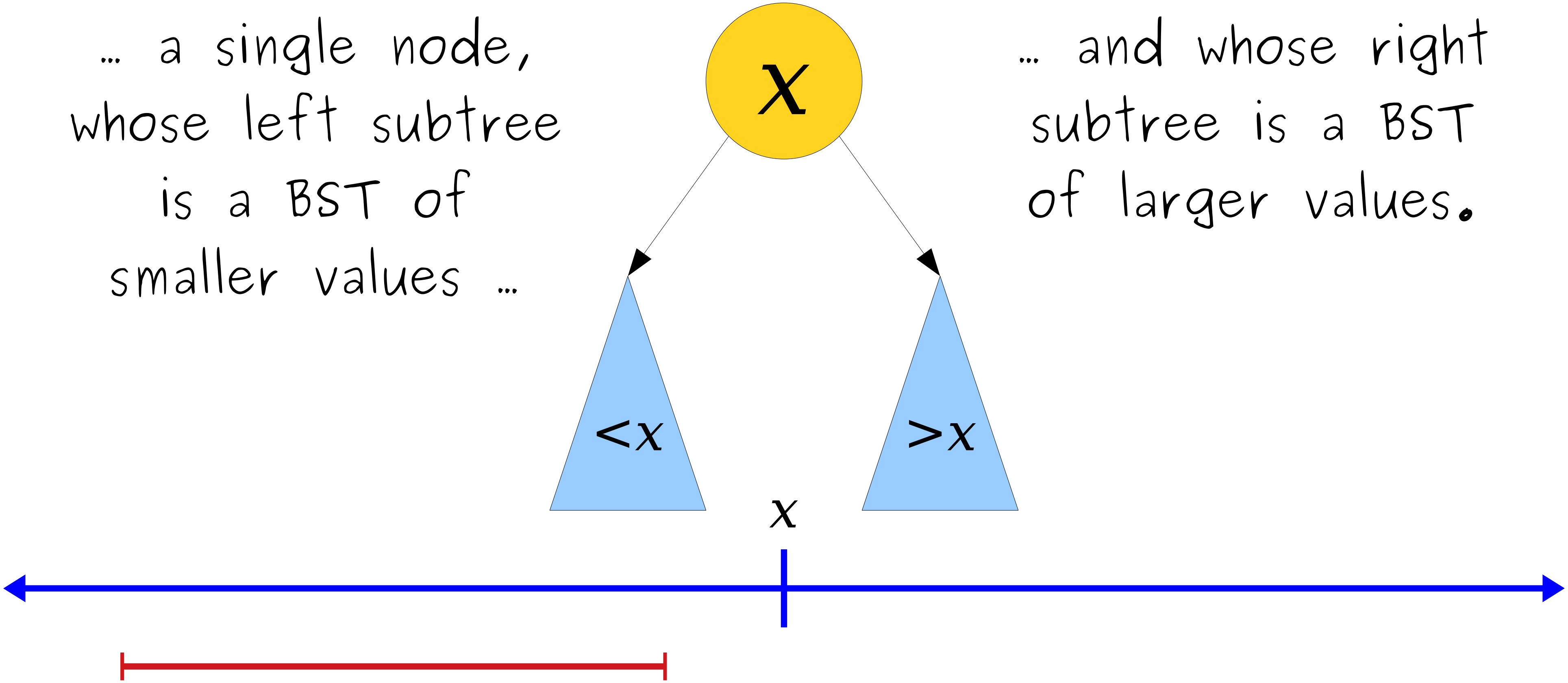


… and whose right
subtree is a BST
of larger values.

A Binary Search Tree Is Either…

an empty tree,
represented by
`nullptr`, or…





… a single node,
whose left subtree
is a BST of
smaller values …

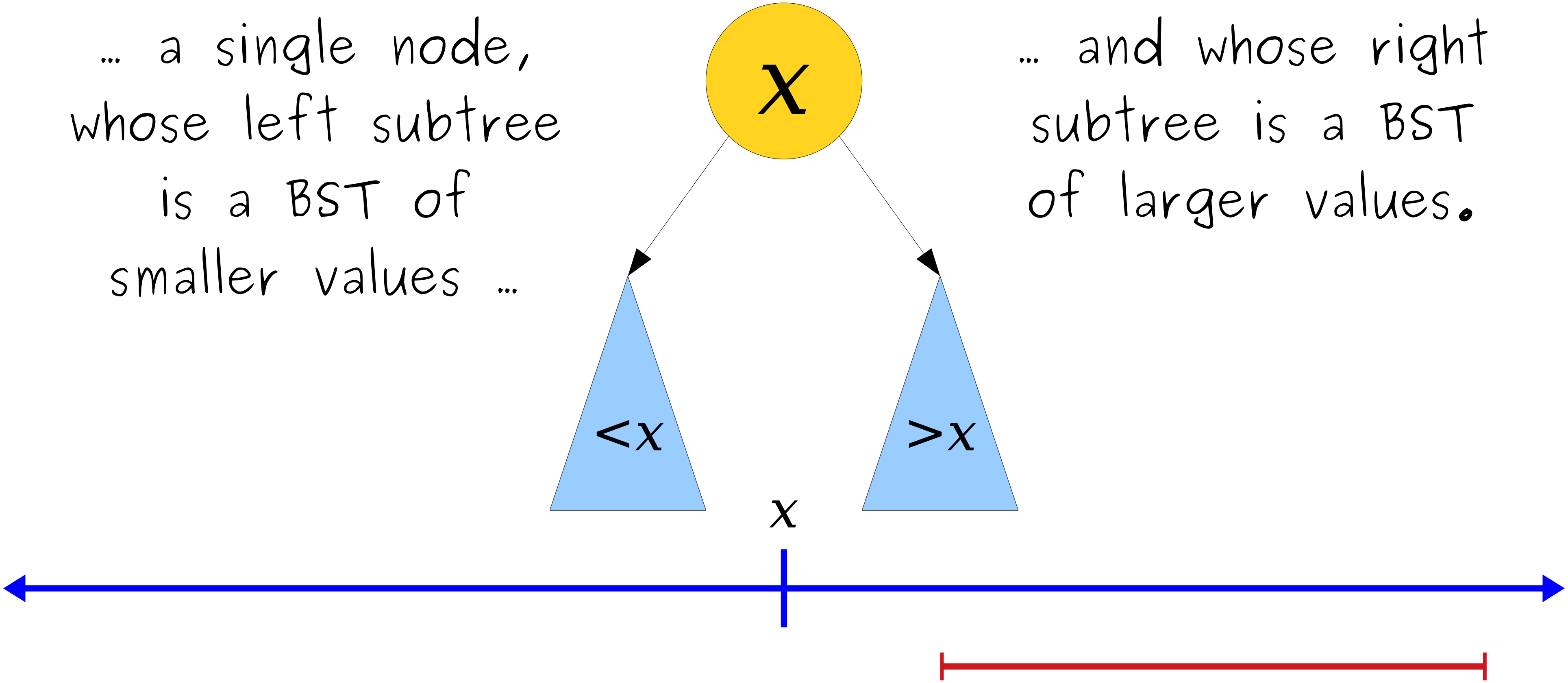
… and whose right
subtree is a BST
of larger values.

A Binary Search Tree Is Either…

an empty tree,
represented by
`nullptr`, or…

… a single node,
whose left subtree
is a BST of
smaller values …

… and whose right
subtree is a BST
of larger values.

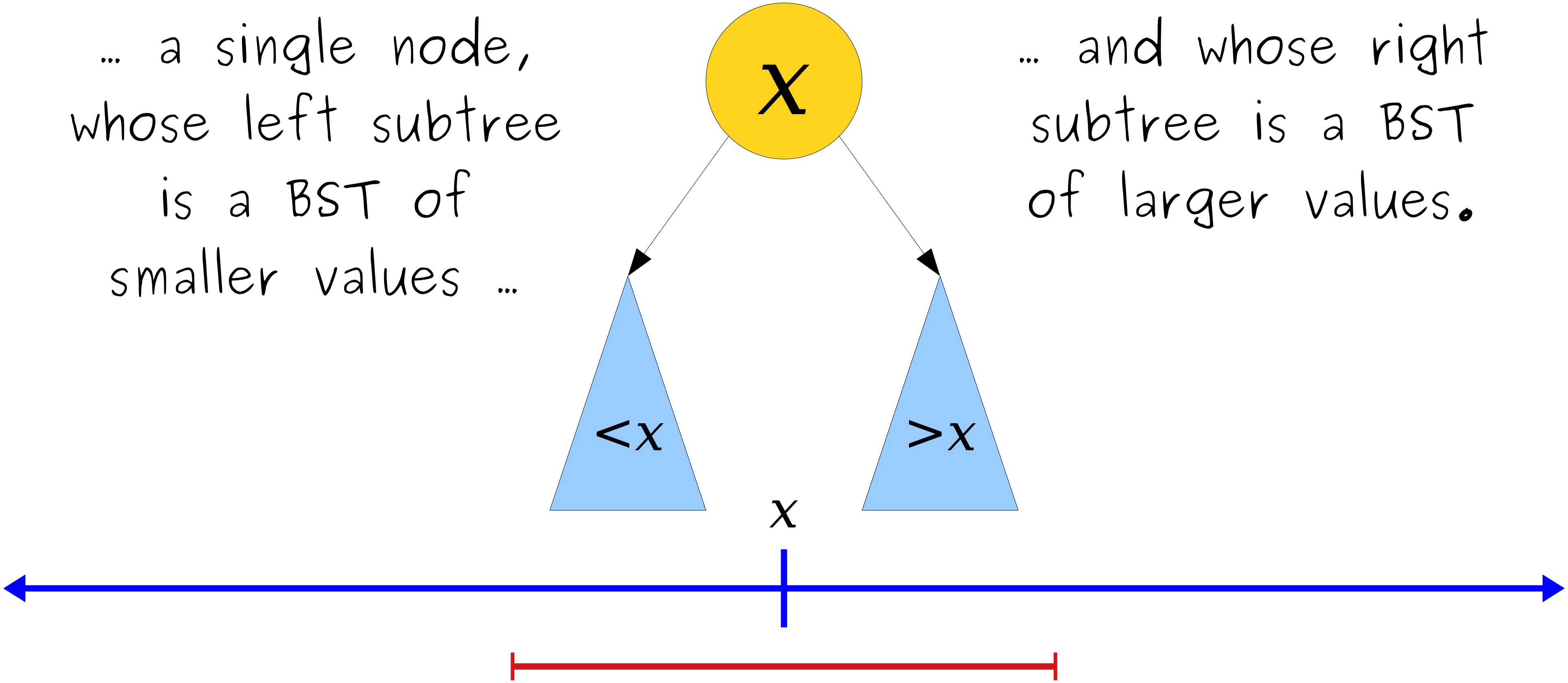


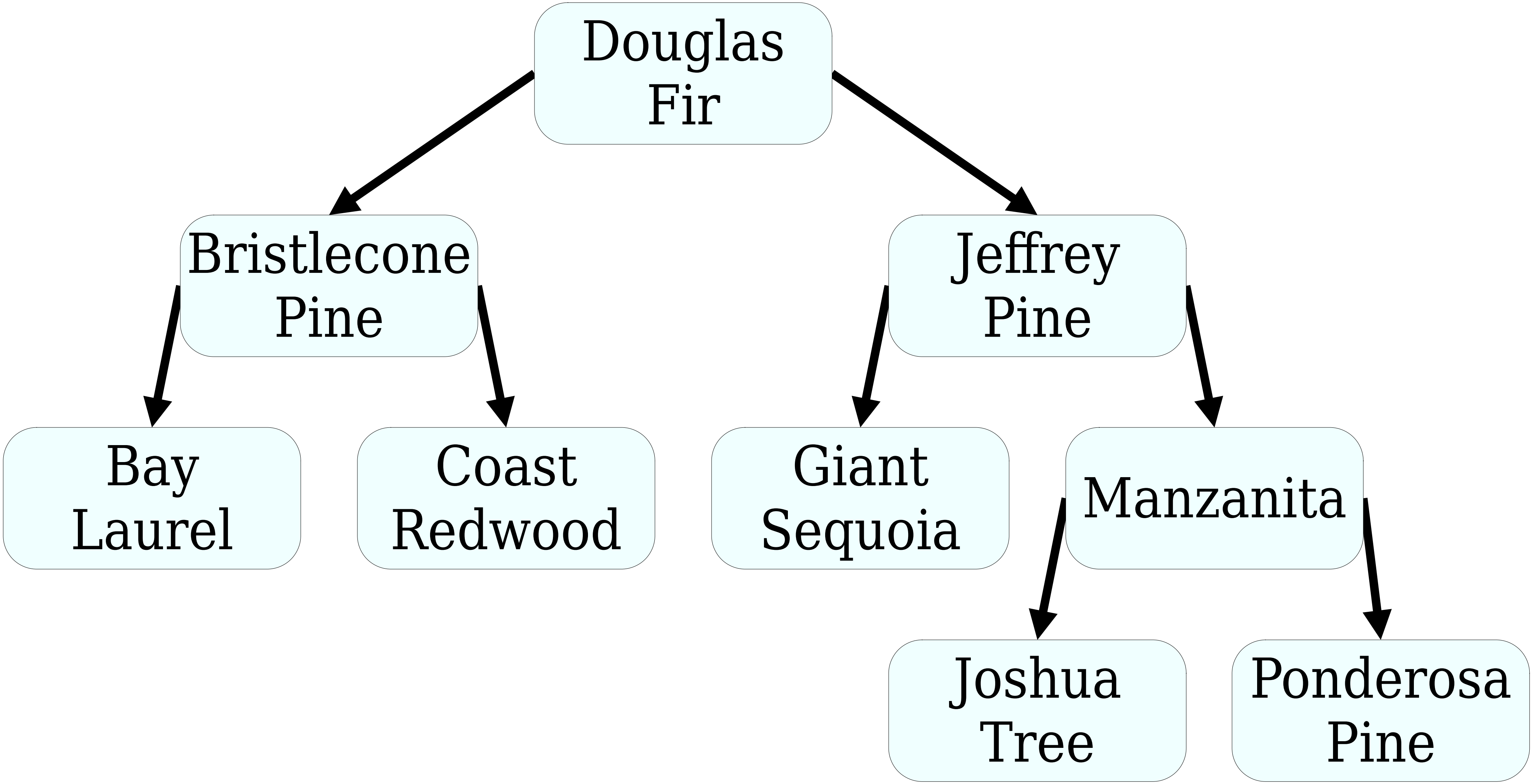
A Binary Search Tree Is Either…

an empty tree,
represented by
`nullptr`, or…

… a single node,
whose left subtree
is a BST of
smaller values …

… and whose right
subtree is a BST
of larger values.





Douglas Fir

Bristlecone Pine

Jeffrey Pine

Bay Laurel

Coast Redwood

Giant Sequoia

Manzanita

Joshua Tree

Ponderosa Pine

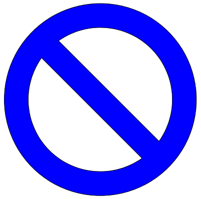
Range Searches

- A hybrid between an inorder traversal and a regular BST lookup!
- The idea:
  - If the node is in the range being searched, add it to the result.
  - Recursively explore each subtree that could potentially overlap with the range.
- ***Fun fact:*** The runtime of a range search is O(h + z), where h is the height of the tree and z is the number of items in the range. Come chat with me after class if you're curious why this is!

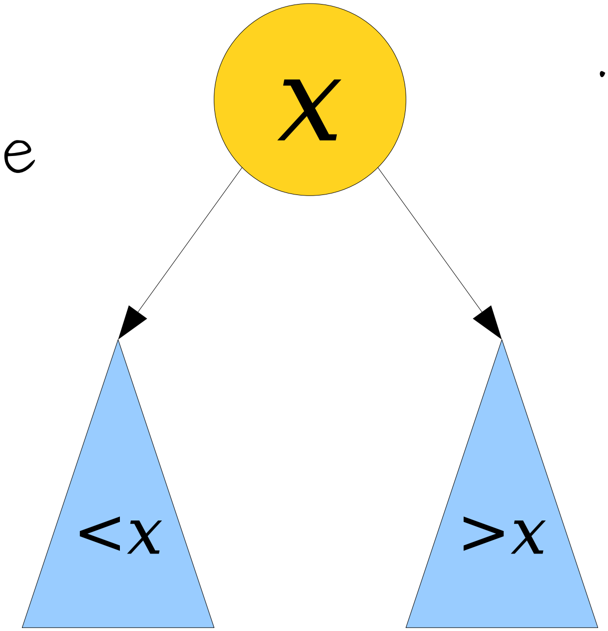
To Summarize:

A Binary Search Tree Is Either…

an empty tree,
represented by
`nullptr`, or…



… a single node,
whose left subtree
is a BST of
smaller values …

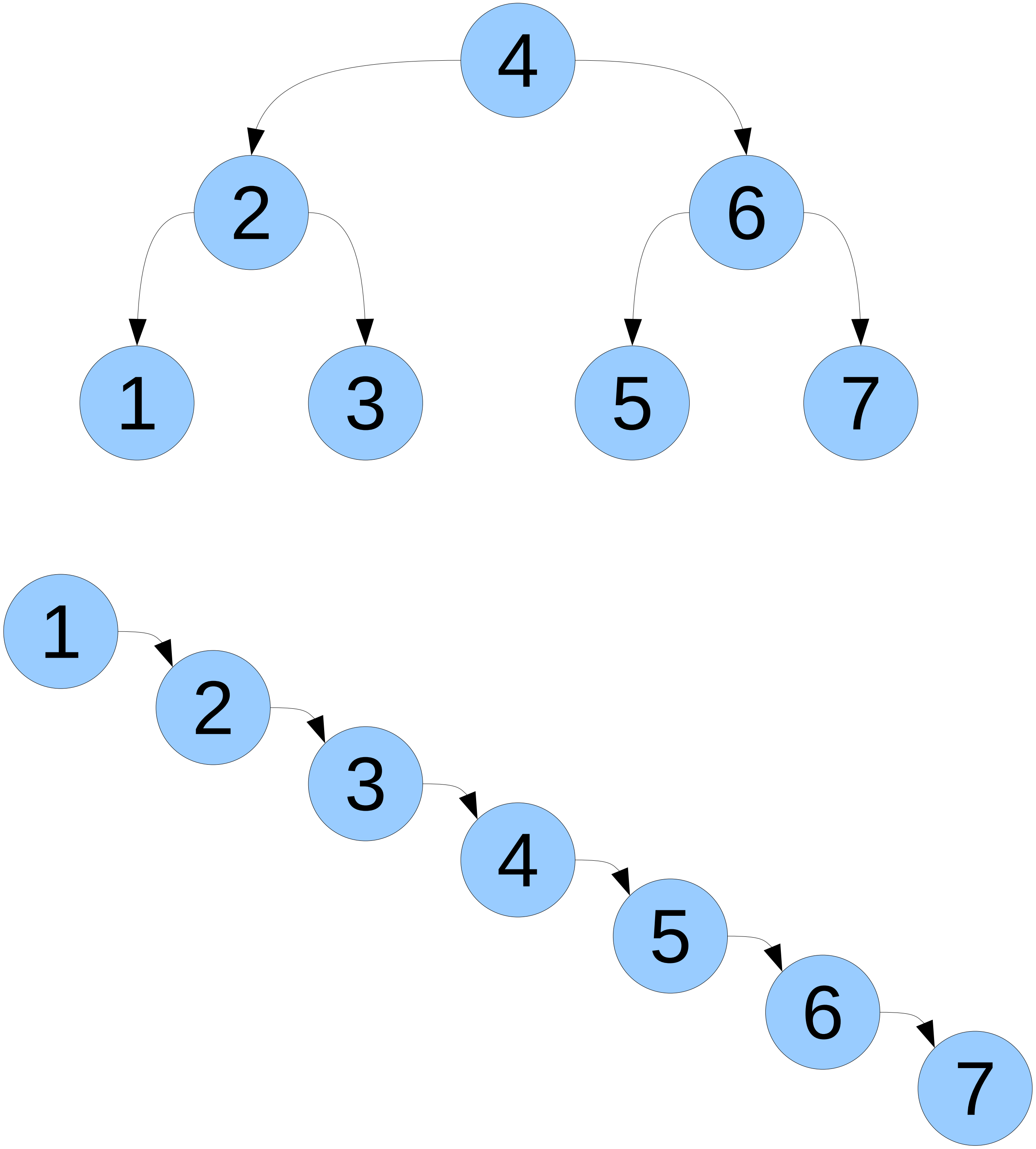


… and whose right
subtree is a BST
of larger values.

```
struct Node {
    Type value;
    Node* left;  // Smaller values
    Node* right; // Bigger values
};
```

```
bool contains(Node* root, const string& key) {
    if (root == nullptr) return false;
    else if (key == root->value) return true;
    else if (key <  root->value) return contains(root->left,  key);
    else return contains(root->right, key);
}

void insert(Node*& root, const string& key) {
    if (root == nullptr) {
        root = new Node;
        node->value = key;
        node->left = node->right = nullptr;
    } else if (key < root->value) {
        insert(root->left, key);
    } else if (key > root->value) {
        insert(root->right, key);
    } else {
        // Already here!
    }
}
```


4
2
6
1
3
5
7
1
2
3
4
5
6
7

```
void printContentsOf(Node* root) {
    if (root == nullptr) return;

    printContentsOf(root->left);
    cout << root->value << endl;
    printContentsOf(root->right);
}

void deleteTree(Node* root) {
    if (root == nullptr) return;

    deleteTree(root->left);
    deleteTree(root->right);
    delete root;
}
```

```
void printInRange(Node* tree, const string& low, const string& high) {
    if (tree == nullptr) return;

    if (high < tree->value) {
        printInRange(tree->left, low, high);
    } else if (low > tree->value) {
        printInRange(tree->right, low, high);
    } else {
        printInRange(tree->left, low, high);
        cout << tree->value << endl;
        printInRange(tree->right, low, high);
    }
}
```

Your Action Items

- ***Read Chapter 16.1 – 16.2.***
  - All about BSTs!
- ***Work on Assignment 8.***
  - Hopefully you’ve escaped your mazes by now! Aim to finish Splicing and Dicing by the Friday deadline.
  - Need help? Have questions? Come talk to us in LaIR or during office hours.

Next Time

- ***Other Binary Trees***
  - BSTs are wonderful, but other tree structures with similar shapes exist.
- ***Huffman Coding***
  - Practical data compression – with trees!