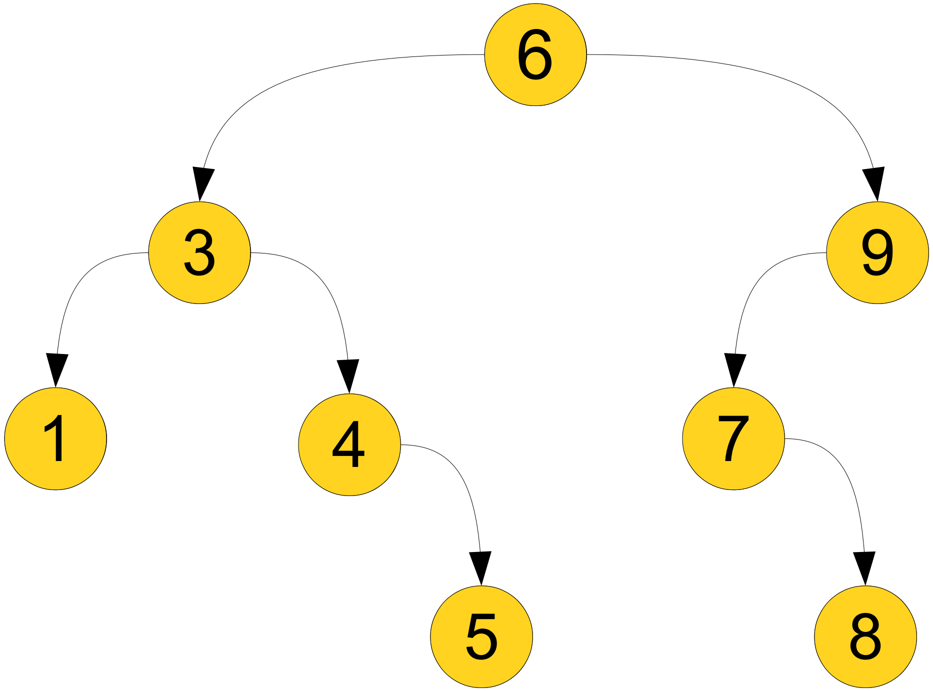
Binary Search Trees

Part Two

Recap from Last Time

Binary Search Trees

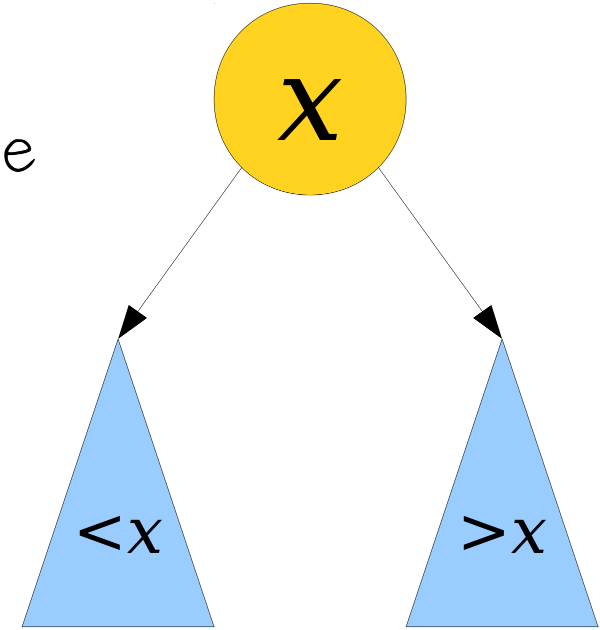
- A ***binary search tree*** (or ***BST***) is a data structure often used to implement maps and sets.
- The tree consists of a number of ***nodes***, each of which stores a value and has zero, one, or two ***children***.
- ***Key structural property:*** All values in a node's left subtree are ***smaller*** 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…

⊘

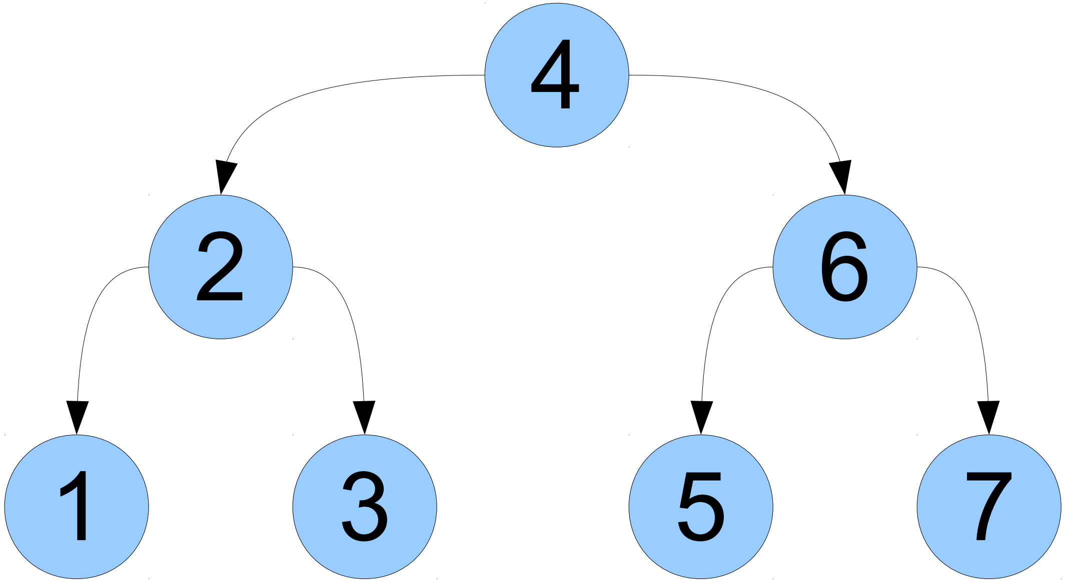


… a single node,
whose left subtree
is a BST of
smaller values …

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

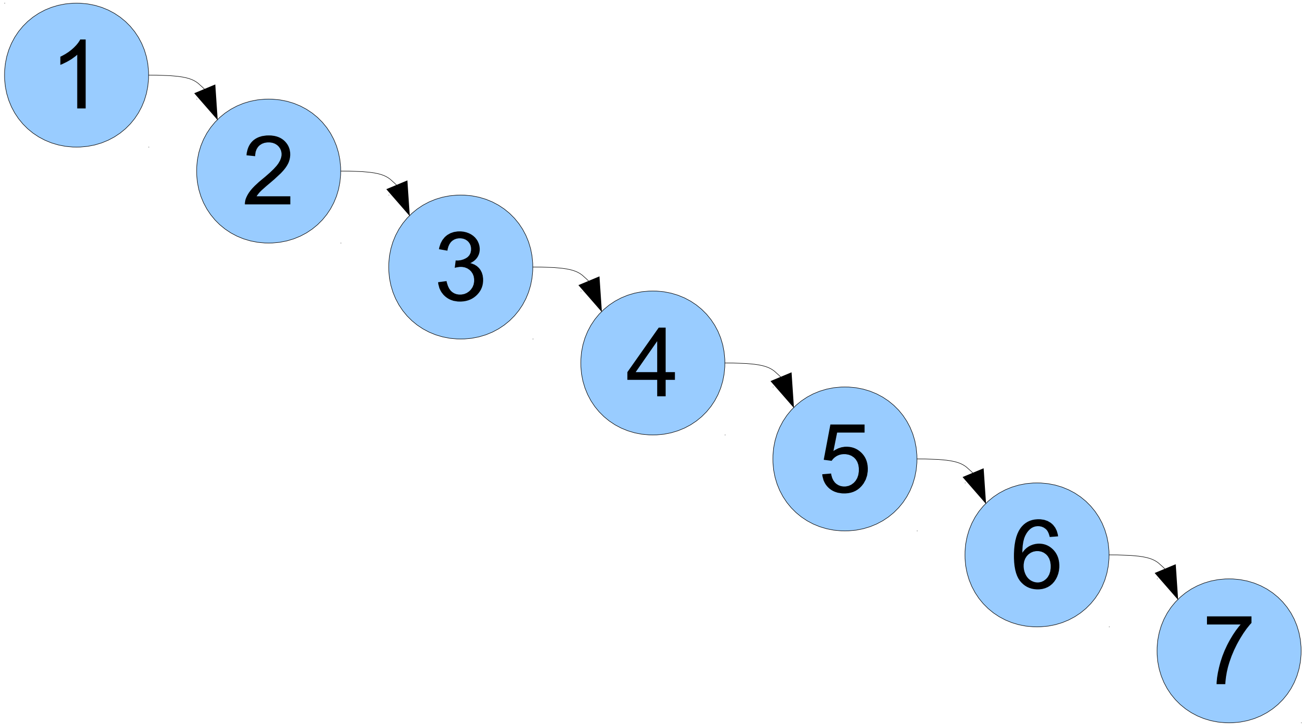
Tree Terminology

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



Tree Terminology

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

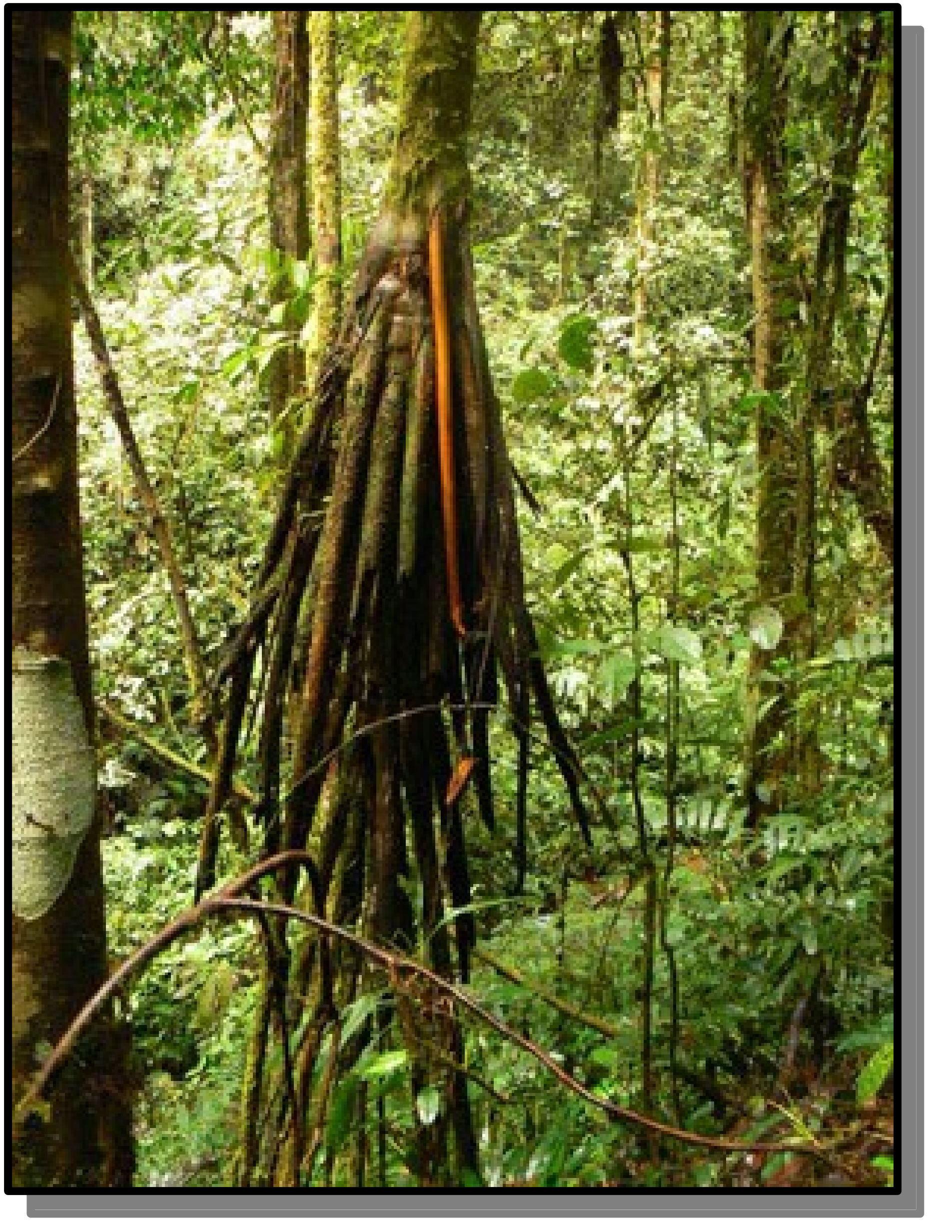


Efficiency Questions

- In a *balanced* BST, the cost of doing an insertion or lookup is O(log n).
- Although we didn't cover this, the cost of a deletion is also O(log n) (play around with this in section!)
- The runtimes of these operations depend on the height of the BST, which we're going to assume is O(log n) going forward.

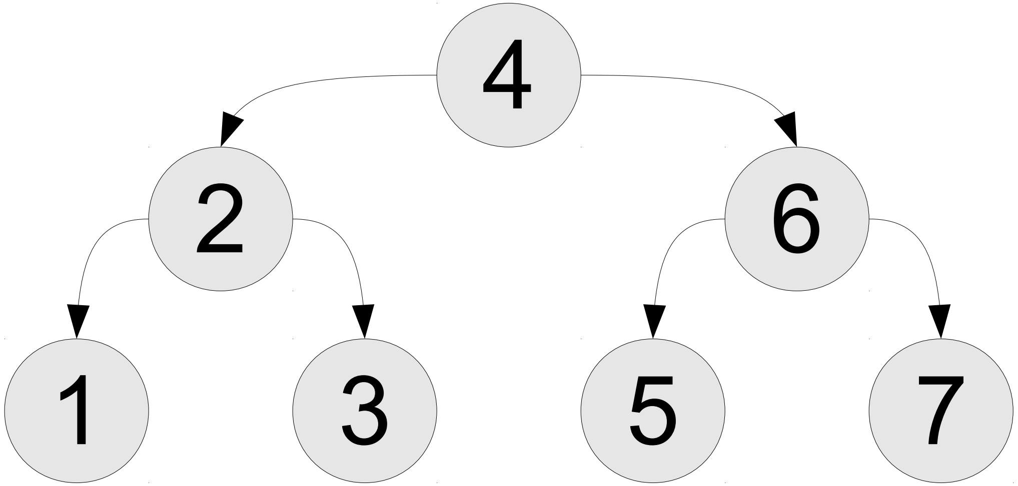
New Stuff!

Walking Trees



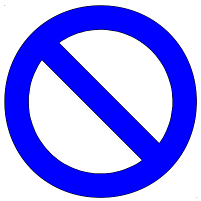
Printing a Tree

- BSTs store their elements in sorted order.
- By visiting the nodes of a BST in the right order, we'll get back the nodes in sorted order!
  - (This is also why iterating over a `Map` or `Set` gives you the keys/elements in sorted order!)

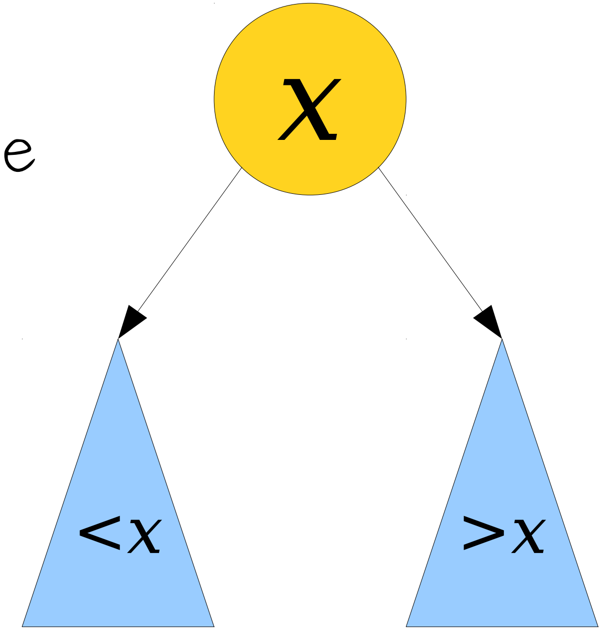


A Binary Search Tree Is Either…

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.

Inorder Traversals

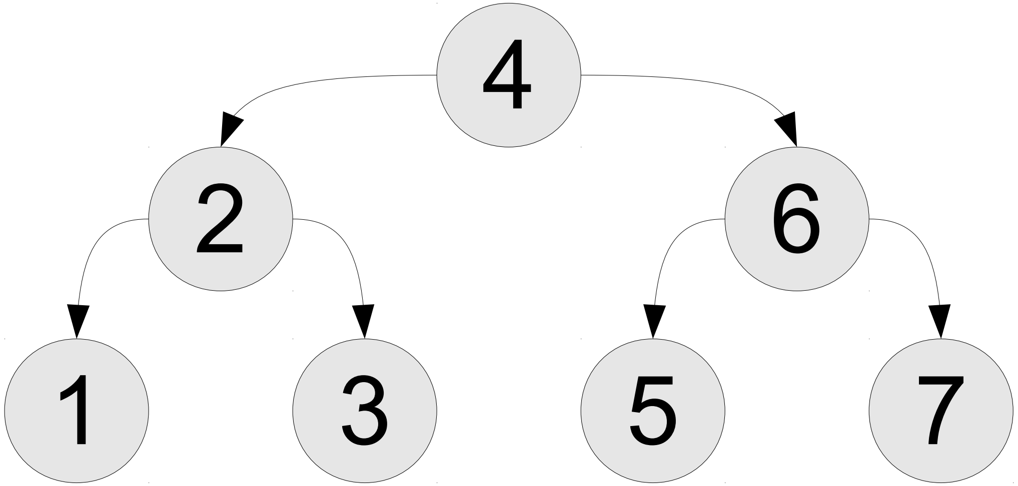
- The particular recursive pattern we just saw is called an ***inorder traversal*** of a binary tree.
- Specifically:
    - Recursively visit all the nodes in the left subtree.
    - Visit the node itself.
    - Recursively visit all the nodes in the right subtree.

Getting Rid of Trees

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

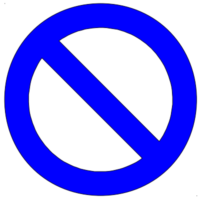
Freeing a Tree

- 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.

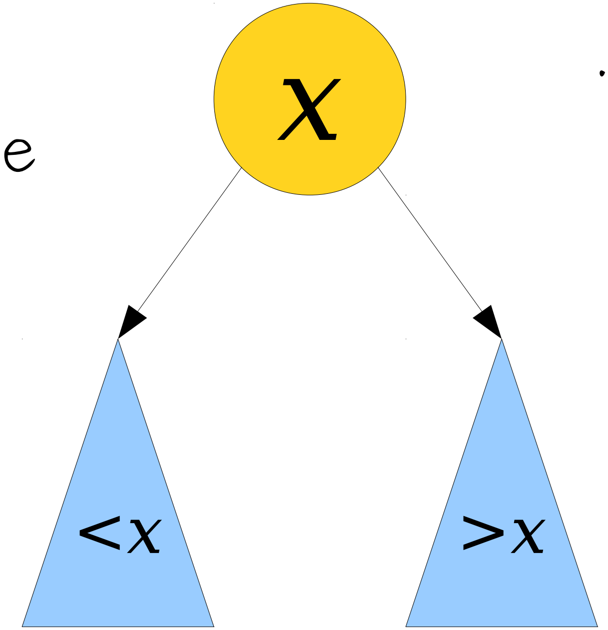


A Binary Search Tree Is Either…

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.

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 left subtree.
  - Recursively visit all the nodes in the right subtree.
  - Visit the node itself.

Time-Out for Announcements!

Assignment 5

- Assignment 5 is due this Friday at the start of class.
- ***Recommendation:*** Aim to complete the first three implementations by the end of tonight. Finish the binary heap by Wednesday.
- Questions? Ask your SL, stop by the LaIR, visit office hours, or ask on Piazza!

CS+SOCIAL GOOD PRESENTS

WINTER **CAREERS PANEL** & **DISCUSSION**

WHERE: **Kehillah Hall, 2nd Floor Hillel**

WHEN: **February 28th, 6:00-7:30pm**

Interested in using your CS skills for social impact? Come hear from people working on tech for social good - learn about career paths and ways to get involved! Dinner will be served at the event.

Moderated by Cynthia Lee

Featuring:

Lean In Exygy Lioness Udemy Facebook Google

Back to CS106B!

Has this ever happened to you?

What’s Going On?

- Internally, the `Map` and `Set` types are implemented using binary search trees.
- BSTs assume there’s a way to compare elements against one another using the relational operators.
- But you can’t compare two `struct`s using the less-than operator!
- “There’s got to be a better way!”

Defining Comparisons

- Most programming languages provide some mechanism to let you define how to compare two objects.
- C has comparison functions, Java has the `Comparator` interface, Python has `__cmp__`, etc.
- In C++, we can use a technique called ***operator overloading*** to tell it how to compare objects using the `<` operator.

This function is named "operator<"

```
bool operator< (const Doctor& lhs, const Doctor& rhs) {
    /*      …      */
}
```

Its arguments correspond to the left-hand and right-hand operands to the < operator.

```
Doctor zhivago = /*      …      */
Doctor acula   = /*      …      */

if (zhivago < acula) {
    /*      …      */
}
```

```
bool operator< (const Doctor& lhs, const Doctor& rhs) {
    /*      …      */
}
```

```
Doctor zhivago = /*     …     */
Doctor acula   = /*     …     */

if (zhivago < acula) {
    /*      …      */
}
```

C++ treats this as

`operator< (zhivago, acula)`

Overloading Less-Than

- To store custom types in `Map`s or `Set`s in C++, overload the less-than operator by defining a function like this one:

```
bool operator< (const Type& lhs, const Type& rhs);
```

- This function must obey four rules:
  - It is ***consistent:*** writing $x < y$ always returns the same result given x and y.
  - It is ***irreflexive:*** $x < x$ is always false.
  - It is ***transitive:*** If $x < y$ and $y < z$, then $x < z$.
  - It has ***transitivity of incomparability:*** If neither $x < y$ nor $y < x$ are true, then x and y behave indistinguishably.
- (These rules mean that < is a ***strict weak order***; take CS103 for details!)

Overloading Less-Than

A standard technique for implementing the less-than operator is to use a ***lexicographical comparison***, which looks like this:

```
bool operator< (const Type& lhs, const Type& rhs) {
    if (lhs.field1 != rhs.field1) {
        return lhs.field1 < rhs.field1;
    } else if (lhs.field2 != rhs.field2) {
        return lhs.field2 < rhs.field2;
    } else if (lhs.field3 != rhs.field3) {
        return lhs.field3 < rhs.field3;
    } … {
      …
    } else {
        return lhs.fieldN < rhs.fieldN;
    }
}
```

One Last Cool Trick, If We Have Time

Filtering Trees

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.

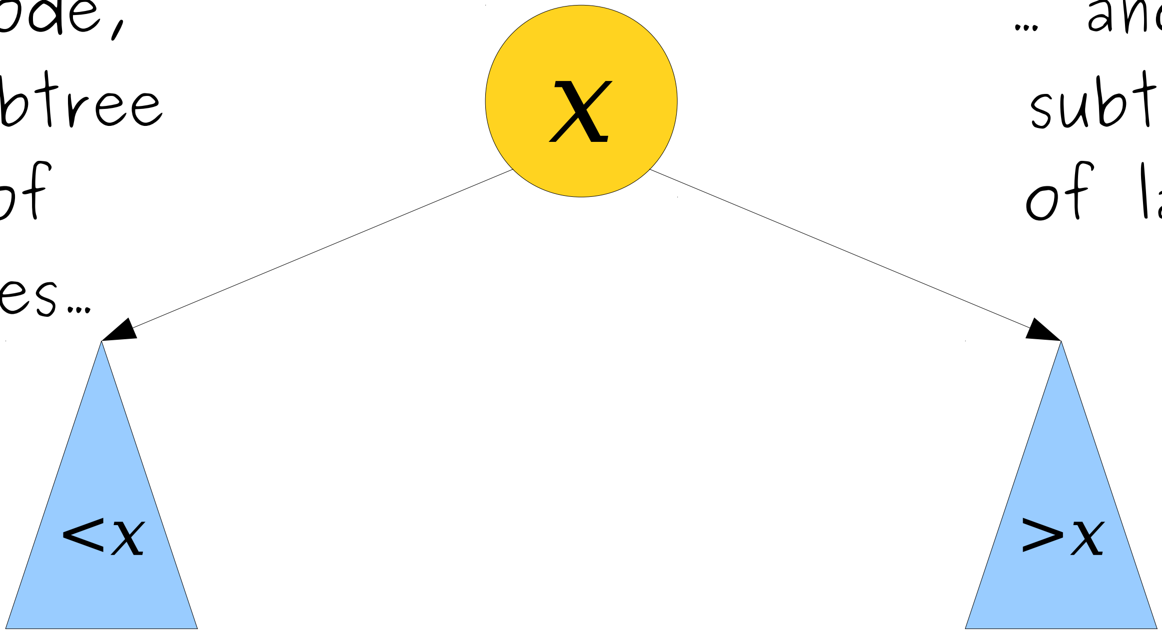
A Binary Search Tree Is Either…

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.



Range Searches

- The cost of a range search in a balanced BST is

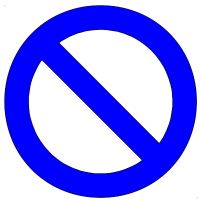
  $$\mathbf{O(\log \boldsymbol{n} + \boldsymbol{z})},$$

  where z is the number of matches reported.
- In a general BST, it's $O(h + z)$.
- Curious about where that analysis comes from? Come talk to me after class!

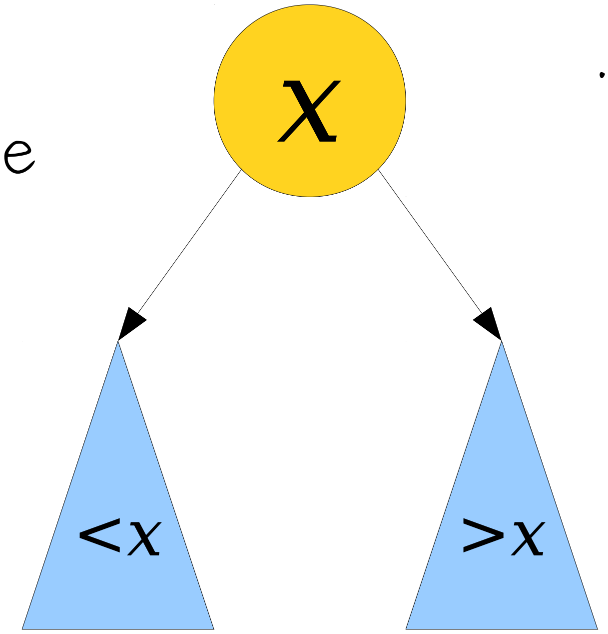
To Summarize:

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 …

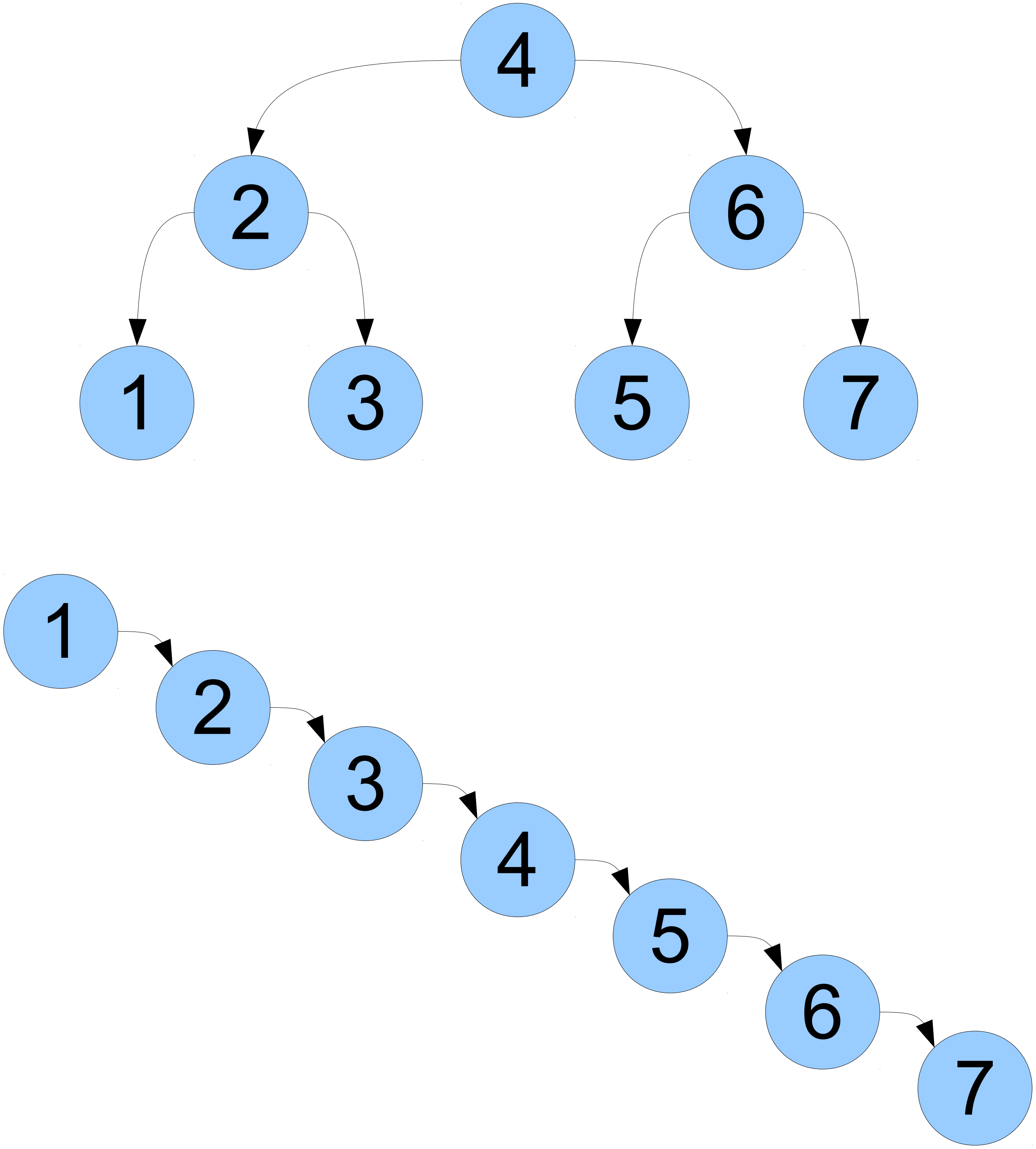


… and whose right
subtree is a BST
of larger values.

```cpp
struct Node {
    int 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 printTree(Node* root) {
    if (root == nullptr) return;

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

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

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

```
bool operator< (const Type& lhs, const Type& rhs) {
    if (lhs.field1 != rhs.field1) {
        return lhs.field1 < rhs.field1;
    } else if (lhs.field2 != rhs.field2) {
        return lhs.field2 < rhs.field2;
    } else if (lhs.field3 != rhs.field3) {
        return lhs.field3 < rhs.field3;
    } … {
      …
    } else {
        return lhs.fieldN < rhs.fieldN;
    }
}
```

Next Time

- ***Beyond Data Structures***
  - Why are these ideas useful outside of the realm of sets and maps?
- ***Huffman Encoding***
  - A powerful data compression algorithm.