Unit 2: Data Structures

...

Data Types & Composite Objects

Motivations

Why is it important to know about the **JavaScript type system**?

- The type system dictates how we are allowed to store information (i.e. program state) and how we are allowed to pass this information around parts of our programs

Why is it important to know about **data structures**?

- Structuring our information/program state according to thoughtful patterns increases program efficiency
- Using common data structures makes code more readable, understandable, and maintainable

Roadmap

In this lecture we'll cover the following topics:

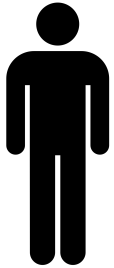
1. **Intro to Type Systems**
    - *Types*
    - *Static vs. Dynamic*
    - *Weak vs. Strong*
2. **Features of JavaScript's Type System**
    - *Dynamic and Weak*
    - *Primitive and Composite Data Types*
3. **Using Data Structures in JavaScript**

JavaScript Type System

Real World Example: What is this thing?

Typing: *assigning a data type is **called typing**. It gives **meaning** to our data. What is it and what can it do!*

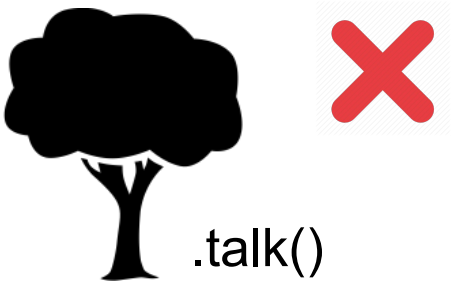
Type: Person
Abilities: Talk



Type: Apple Tree
Abilities: Grow Apples

Real World Example: What is this thing?

Type Errors: *When you try to use a thing in a context where it doesn't make sense (it doesn't work).*



Dynamic Typing

JavaScript is a **dynamically typed** language. Variables **Do NOT** have a type. The values they hold have a type. These get assigned at runtime.

```
let hello = 2;

hello = “world”
```

This speeds the JavaScript development cycle, but is less efficient than statically typed languages at runtime

Weakly Typed

JavaScript is **weakly typed.** What happens if we call add(1, “cat”) ?

```
function add(a, b) {
  return a + b;
}
```

= “1cat”

Type Systems: Strong/Weak scale and Static/Dynamic Scale

- *Static/Dynamic typing* is about **when** type information is acquired (either at compile-time or at runtime). More concretely, statically typed languages require knowledge of what data type each variable holds.

- *Strong/Weak typing* is about **how strictly** types are distinguished (e.g. whether the language tries to do implicit conversion from strings to numbers). This conversion is often referred to as ‘type coercion’.

Primitive Data Types

Primitive Types

JavaScript is composed of 7 primitive data types and the Object data type. Primitives are **immutable values** without properties:

- Number
- String
- Boolean
- BigInt
- Undefined
- Null
- Symbol

Primitive Types and Immutability

All primitive data types are **immutable**, which means we cannot change/alter any portion of them.

```
1    let originalString = "abcd";
2    originalString[2] === "c"; // true or false?
3    originalString[2] = "2"; // ???
4    console.log(originalString); // what do we get?
5
```

Primitive Types are Passed by Value

```
1  function change(passedNum) {
2      passedNum += 1;
3  }
4
5  let originalNum = 1;
6  change(originalNum);
7  console.log('originalNum is:', originalNum);
8
```

Composite Data Types

Composite (Compound) Data Types

A ***composite (compound)*** data type is **any data type which can be constructed using its programming language's primitive data types and other composite types.** The act of constructing a composite type is known as composition.

Composite data types are **user defined**, use **built in data-types** and composes them **to build all new data types**. (linked lists, stacks, etc)

JavaScript composite types include **Arrays** and **Objects**

Object

Objects are a javascript data type composed of key/value pairs. We can change/update the key/value pairs in an object. Objects, therefore, are **mutable!**

```
const myObj = {
    my_key: 123;
};

myObj.my_key = 321;

console.log(myObj.my_key);
```

JavaScript Arrays

Arrays are built-in objects organized by **integer-keyed** properties and a **length** property.

- Great for working with ordered lists, but require some extra work for things like searching, adding and removing.
- Arrays are actually objects in JS (they're also dynamic!)

```
const myArr = [0,1,2,3,4,5,6]; // length = 7
```

Arrays: Under The Hood

```
Array prototype names:
[
  'at',          'concat',      'constructor',
  'copyWithin',  'entries',     'every',
  'fill',        'filter',      'find',
  'findIndex',   'findLast',    'findLastIndex',
  'flat',        'flatMap',     'forEach',
  'includes',    'indexOf',     'join',
  'keys',        'lastIndexOf', 'length',
  'map',         'pop',         'push',
  'reduce',      'reduceRight', 'reverse',
  'shift',       'slice',       'some',
  'sort',        'splice',      'toLocaleString',
  'toString',    'unshift',     'values'
]
```

Composite Data Types are Passed by Reference

Composite Types are Passed by Reference, meaning they represent the same place in memory when assigned to existing variables

```
1    function change(passedObj) {
2        passedObj.myNum += 1;
3    }
4
5    const originalObj = { myNum: 1 };
6    change(originalObj);
7    console.log(originalObj);
8
```

Primitive Data Types: passed by value
Composite Data Types: passed by reference

Primitive data types are **copied**. Object data types are **referenced**.

```
let a = 4;
let b = a;
b = 5;

// a = ?
```

```
const a = {num:4};
const b = a;
b.num = 5;

// a.num = ?
```

New(ish) JavaScript Features

What is ES6?

The latest *major* release for JavaScript (ECMAScript 2015) was released in Fall of 2015. It's the 6th version of ECMAScript, and is often referred to as ES6.

The prior standard, ES5, was released in 2009. At this time, it's good to know both ES5 and ES6.

ES7 and ES8 were released in 2016 and 2017, respectively. While they contain useful utilities, such as async/await and string padding, they contain much smaller feature sets than ES6.

Keyed Collections

ES6 introduces **Map** and **Set**

A **Set** represents a unique set of values.

Set has methods like add, has, delete, foreach, et al

A **Map** associates a value with a collection.

Map has methods like get, set, has, foreach, et al

Map

```
const trafficLights = new Map();
trafficLights.set('green', 'go');
trafficLights.set('yellow', 'be careful');
trafficLights.set('red', 'stop');
console.log(trafficLights.has('green'));
console.log(trafficLights.get('green'));
for ([color, meaning] of trafficLights.entries()) {
  console.log(color + ' means ' + meaning);
}
```

Map vs Object - What and When?

Using Data Structures

Objects vs. Array

JavaScript **objects** and **arrays** are both examples of data structures. Specifically, they create **composite data structures** out of smaller **components**.

Array or Object? : How should we store all of the stock prices of Apple at 10 second intervals across the whole day?

Array or Object? : “How should we store the frequency of each character in this sentence?”

Searching Arrays

How to find an object by name in an array:

1. Direct lookup by index (**O(1) constant time**).
2. Loop through to find a match (**O(n) linear time**).

```
let users = [{'will': 3},{'dan': 4},{'jared': 5}];
users[1].dan; // value for dan is 4
```

Searching Objects

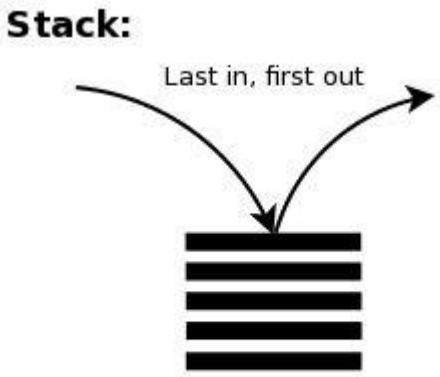
Since you're referencing properties by key, the search is a direct lookup (**O(1) – constant time**).

```
var users = {will: 3, dan: 4, jared: 5};
users['dan']; // 4
```

Stacks & Queues

Stacks: Last In, First Out

- A stack is a list of elements that are accessible only from one end of the list, which is called the *top.*
- Stacks are efficient because data can be added or removed only from the top (constant time) making them fast and easy to implement
- We can use an array or an object to implement our stack. Primary behavior includes:
  - Pushing - adding to the top of the add
  - Popping - removing from the top of the stack

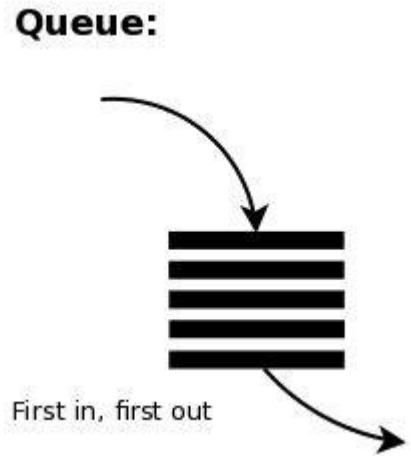


Queues: First In, First Out

- Queues are similar to stacks in that you can't access random elements in it.
  - Enqueue: add an item to back of the queue
  - Dequeue: Remove item off the front of queue

- With array, enqueue (adding) is constant time and dequeue (removing) is linear

- With object(s), enqueue and dequeue are constant

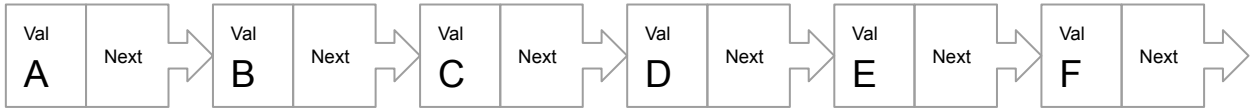


- push() back of queue
- shift() front of queue

Linked Lists

Linked List - Introduction

- A linked list is a collection of objects (called *nodes*), each of which has a property that references (or *points to*) the next object in the list.

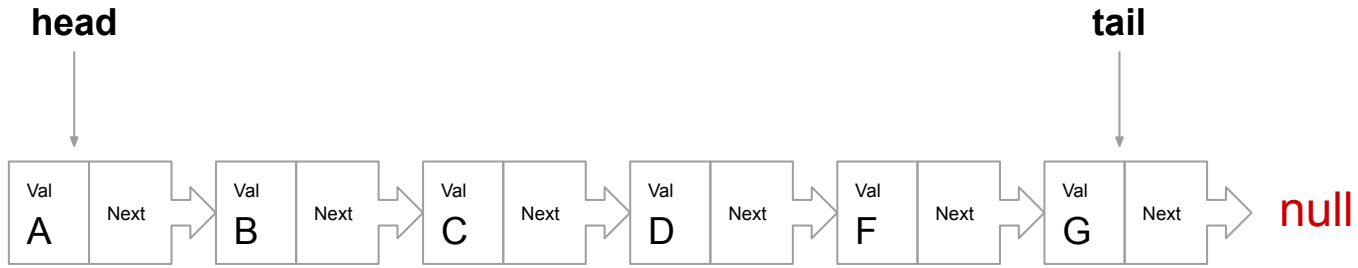


- We must keep track of the first node (or head) in order to access and traverse our list.
- It's very useful to keep track of the last node (or tail) to give us a shortcut to add new nodes to end of our list
- But one of the things that is most useful about linked lists is that adding or removing nodes from the middle of the list is very efficient compared to arrays

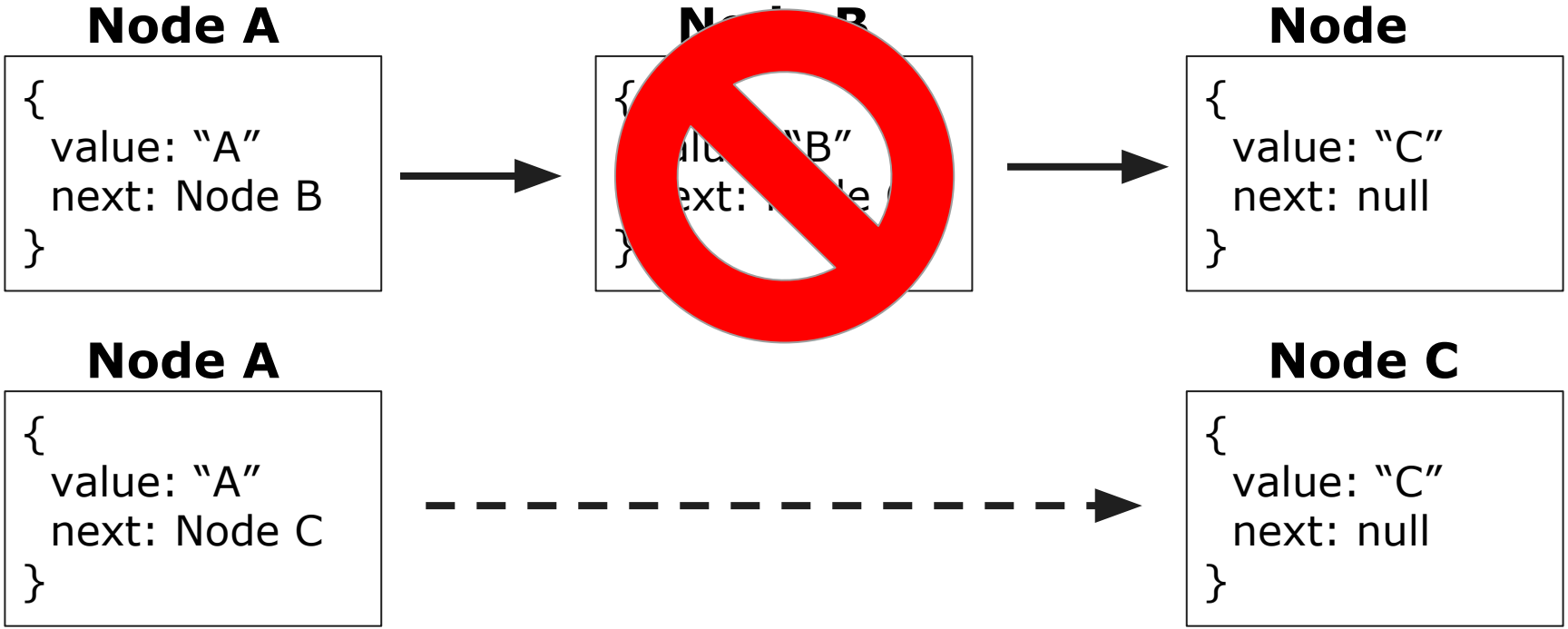
Linked List - Entry Points: Head & Tail

- In a singly linked list, each node has a single pointer (next). The last node points to null to indicate it's the last node in the list.

- A linked list conventionally has one or two entry points. These entry points are called the head and tail pointers.
  - The head and tail pointer are our only entry points into the list
  - With a singly linked list, a tail pointer is not required. However, it does make insertion of new nodes more efficient O(1) vs. O(n) insertion



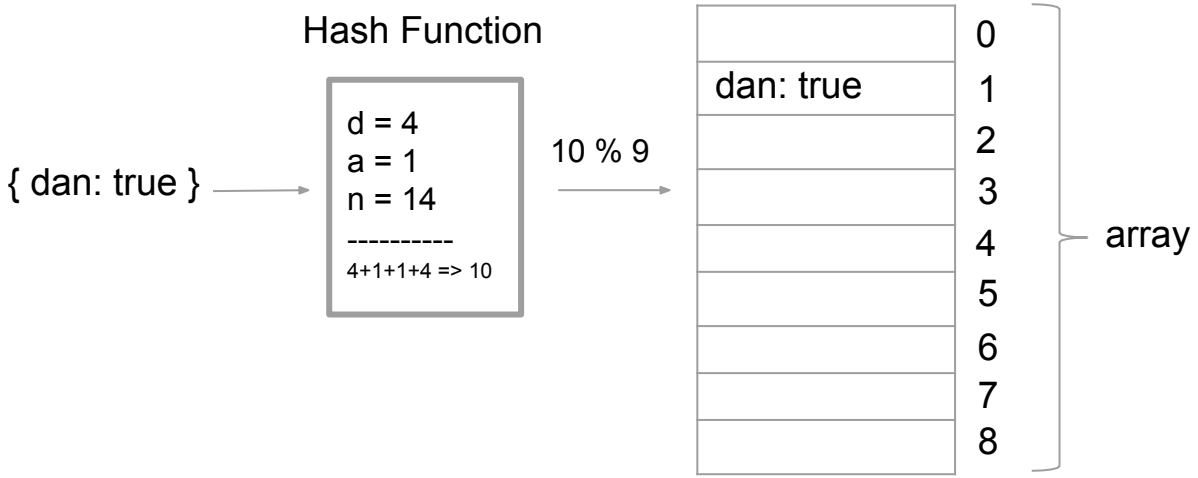
Removing a Node



Hash Tables

Objects & Hash Tables

- A **hash table** is a data structure (array) that is used to store information (data).
- Data is inputted into an array at a specific location in the form of key value pairs. This location can be referred to as a "**bucket**"
- **Hashing** - we use a process called hashing to convert some sort of input into an integer. This integer then becomes to index of the array where our key value will be stored



What should a Hash Function do?

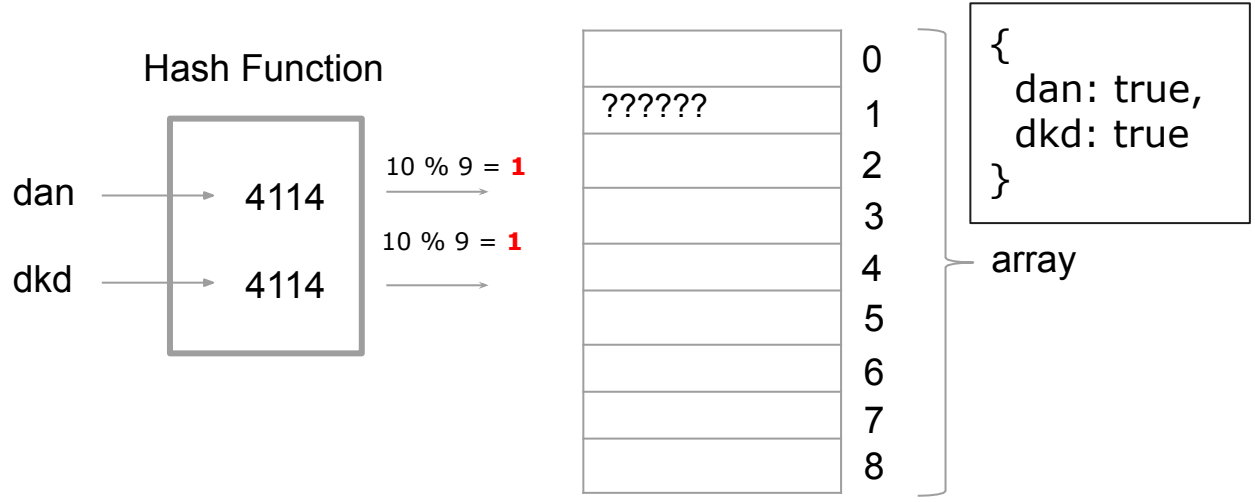
- Map to unique numbers as much as possible with minimal collision.
  - *A hash function is no good if you always return 1 for any keyword, right!*
- Consistently map a name to the same index. Every time you put in ‘dan’ you should get the same index number back

Hash Table Collision

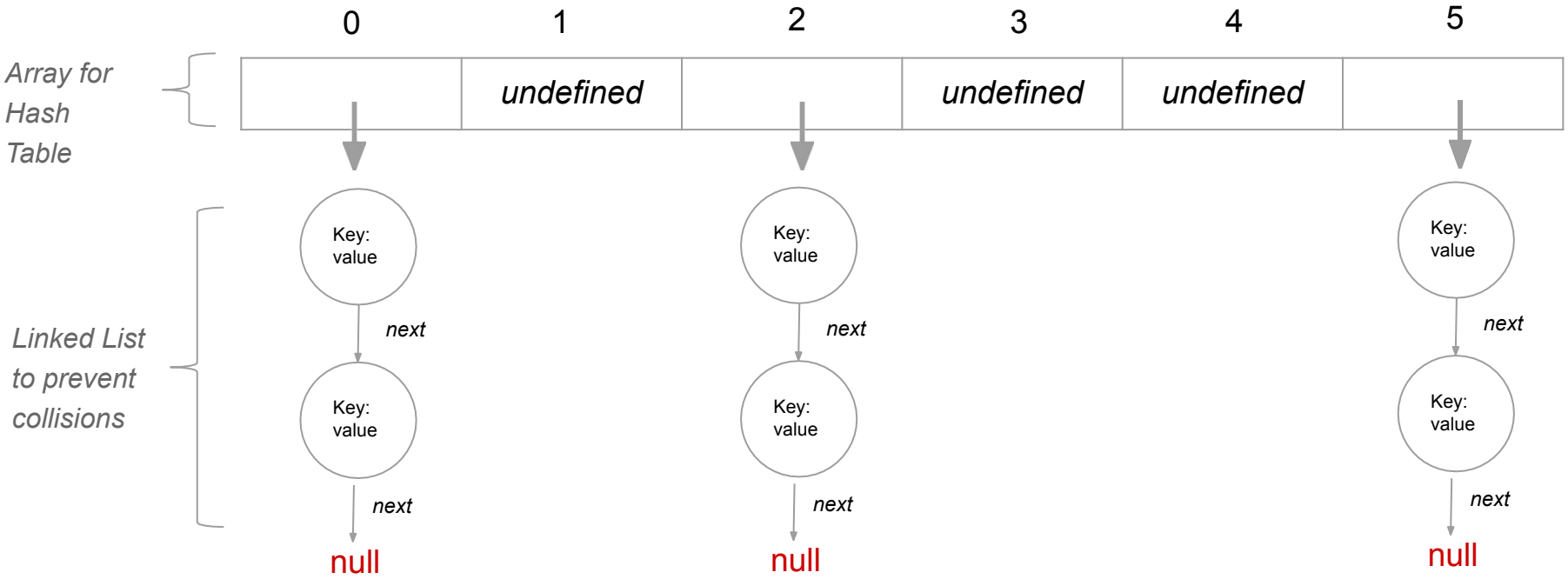
The problem with finding memory addresses with a hash function is the possibility of **collision**. *How can you solve this?*

hashtable.add(dan, true)

hashtable.add(dkd, true)



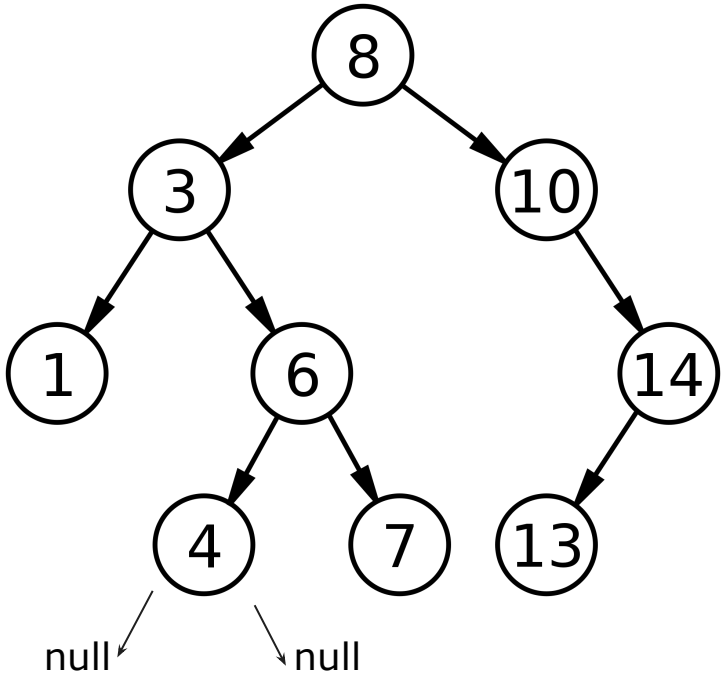
Hash Table Collision Solution? => Linked Lists!



Binary Search Trees

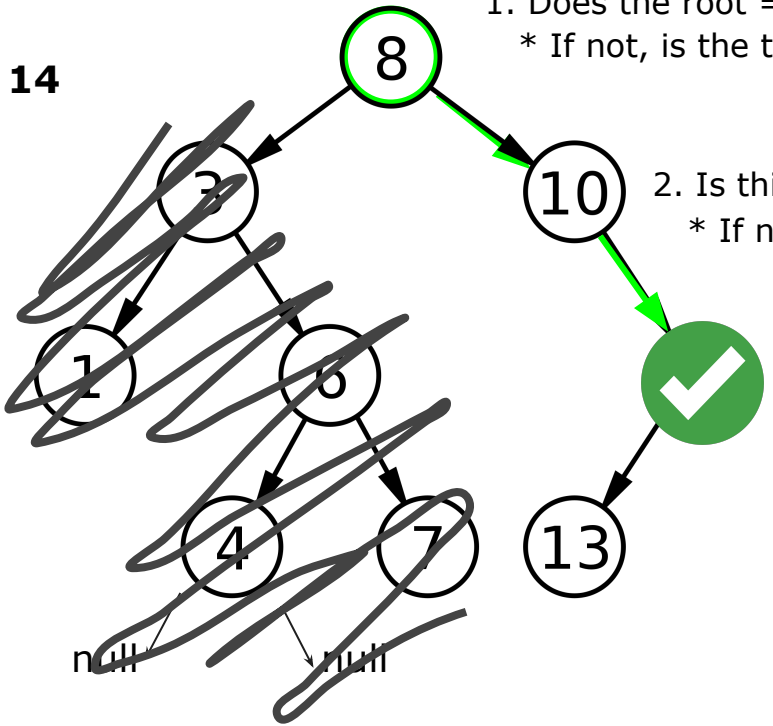
Binary Search Trees (BST)

- Binary Search Trees are designed to make data lookup very fast and efficient
- Like linked lists, they are just collections of objects that point to each other
- A Binary Search Tree allows you to store nodes with lesser values on the left side and greater values on the right side of a parent node
    - The top most node is called the ***root***, and the bottom most nodes for each parent, which point to null (no values), are called ***leaf nodes***.
    - A BST always has a **left** node and a **right** node (binary, two options).



BST Search

Find if target value of 14 exists in our BST…



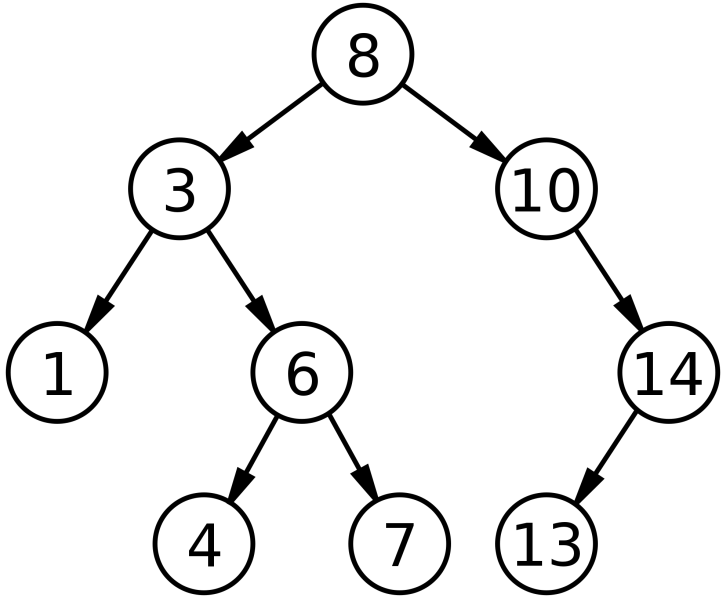
1. Does the root = target value?
   * If not, is the target > than the root?

2. Is this node the value we're looking for?
   * If not, is the target > than the root?

3. Is this node the value we're looking for?

Binary Search Tree Traversal

- They can be traversed with different methods, predicated on when the value of the current node is provided or displayed
  - Depth first
    - Pre Order - emit as you get to each node
    - Post Order - emit the last time you visit the node
    - In Order - display in sorted order by going down the left side first
  - Breadth-first
    - Traverse in level order where we visit each node on a level before going to a lower level



Core Takeaways

- Javascript is dynamically typed and weakly typed
- Primitive data types are passed by value
- Composite data types are passed by reference
- Many types of data we run into in the wild have a natural structure (think stock prices, or the call stack). By storing this data in appropriate data structures, our code is more understandable and frequently more efficient