INTRO TO DATA STRUCTURES

Fullstack Academy of Code

What is a Data Structure?

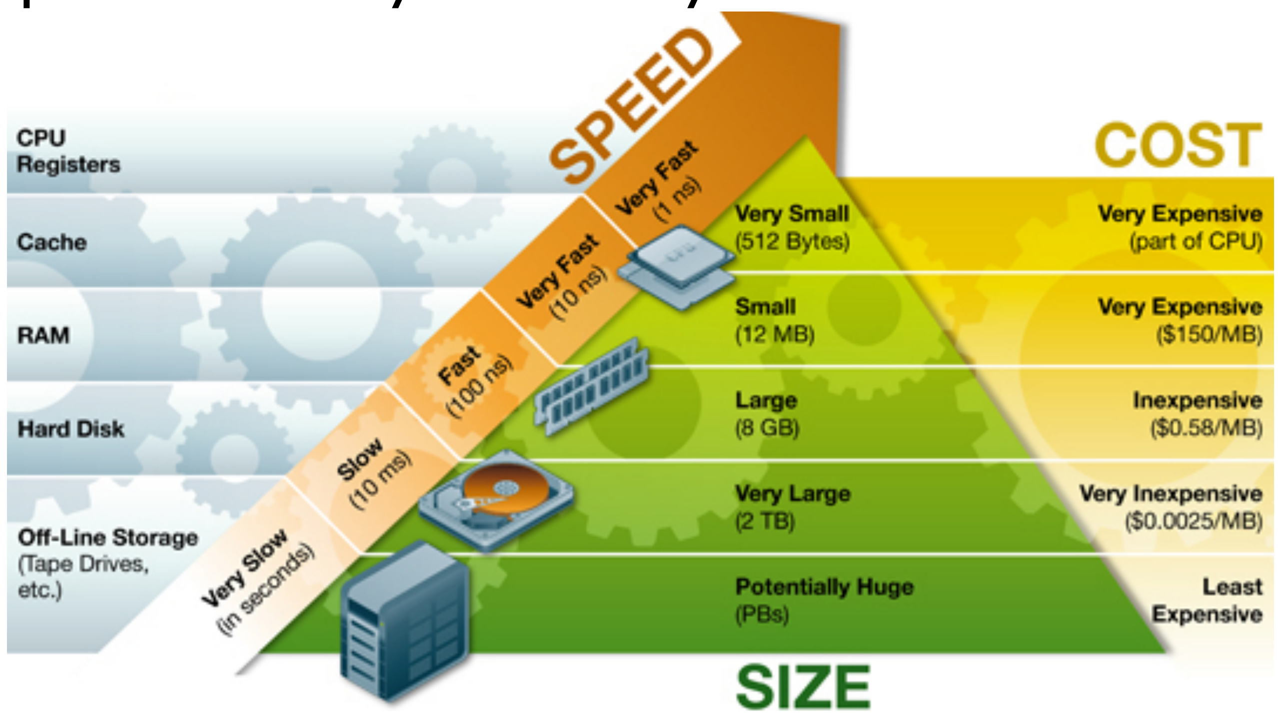
- An blueprint which defines how a data set is organized
- An abstract way to structure data in a way that makes adding more data and finding data much faster
  - Example: Library of Books. Library is the Data Structure and Books are the data.
- The simplest data structure is a **primitive variable**
  - Example: Boolean, Integer, Floating Point integer (decimal)
- Complex Data Structures
  - Array
  - Hash Table/Map
  - Stacks/Queues
  - Linked Lists
  - Trees

Some Background

Memory and Pointers oh my!

Computer Memory Organization

- Let's first understand memory a bit more
- Organized in blocks (like an excel sheet)
- Computer Memory Hierarchy



Memory and Pointers

- Memory is organized like a table
- Each field has an "address"
- A pointer is a variable that holds the address of a item in memory
- A phone book vs neighborhood
  - Phone book is a list of pointers to houses.
  - Neighborhood has the actual houses.

Pointers in C++

- In C/C++ there is a explicit distinction between a "pointer" and an actual variable that holds value.

```cpp
#include <iostream>
using namespace std;

int main ()
{
   int  var = 20;   // actual variable declaration.
   int  *ip;        // pointer variable

   ip = &var;       // store address of var in pointer variable

   cout << "Value of var variable: ";
   cout << var << endl;

   // print the address stored in ip pointer variable
   cout << "Address stored in ip variable: ";
   cout << ip << endl;

   // access the value at the address available in pointer
   cout << "Value of *ip variable: ";
   cout << *ip << endl;

   return 0;
}
```

Data Structures

Primitive Variables

- Booleans
  - Have values: True or False
- Integer
  - Whole numbers (positive and negative)
- Floating Point
  - A Decimal Number Approximation
  - Represented as Binary fractions
  - Try 0.1*0.2 in the Chrome Web Tools

Arrays

- Arrays
  - A fixed range of Memory Addresses that allocated for use by an array
  - Size of array must be declared at the start
  - Why are Arrays indexed from Zero?
- Dynamic Arrays
  - JavaScript/Ruby use Dynamic Arrays
  - Grow as needed

Linked Lists

- A sequence of data nodes, each containing arbitrary data fields and a pointers ("links") to next node
- Elements of Linked Lists
  - Pointer to HEAD node
  - Each node has a value and a NEXT pointer
  - If the NEXT pointer is null, the list has ended

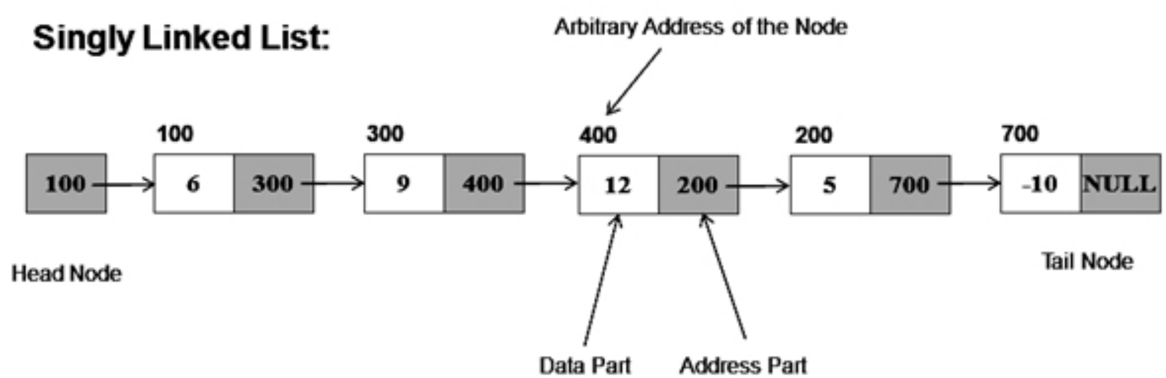
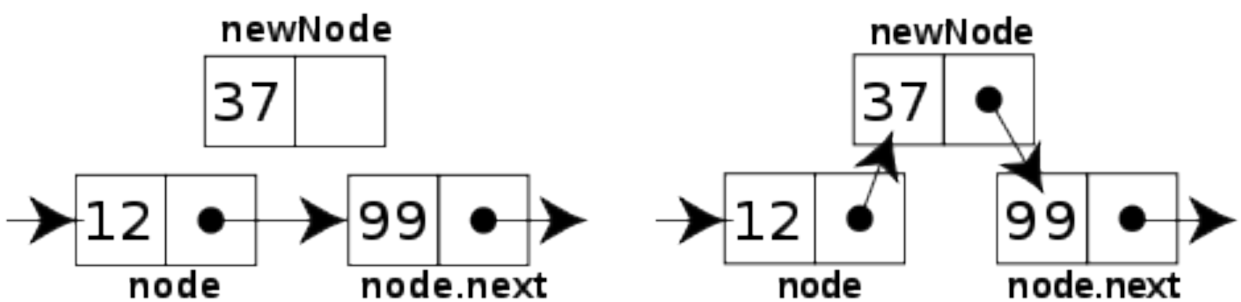


Fig: Singular Linked List

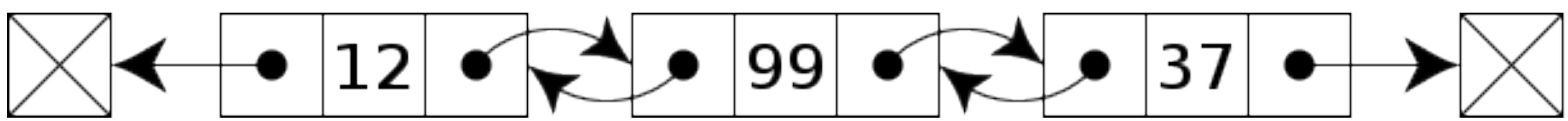
Working with Linked Lists

- Finding a node
  - Traverse the whole list, starting with HEAD
  - O(n) time
- Adding a node



Doubly Linked Lists

- Doubly linked lists are lists that have pointers to the next node AND the previous node

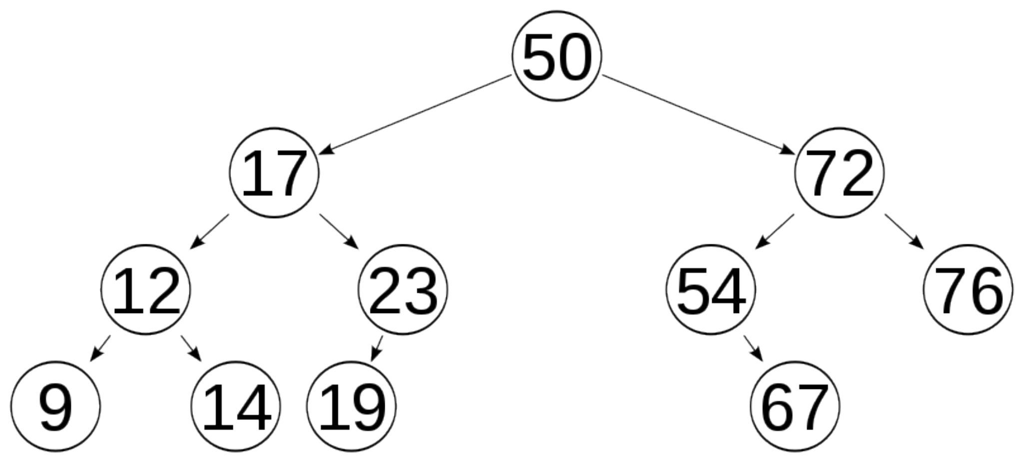


Trees and Hashes

- In Linked Lists and Arrays, searching is SLOW
- What if searching data was a frequent operation in your application?
- What if you wanted to store data in a sorted way?

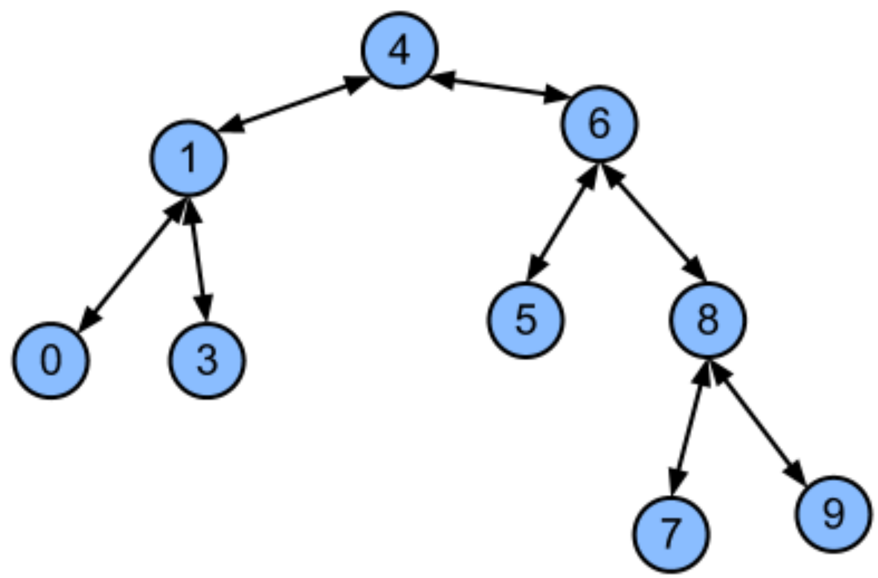
Binary Search Tree

- Like a linked list, where each node has two linked nodes.
- The left node is smaller and right is larger or equal



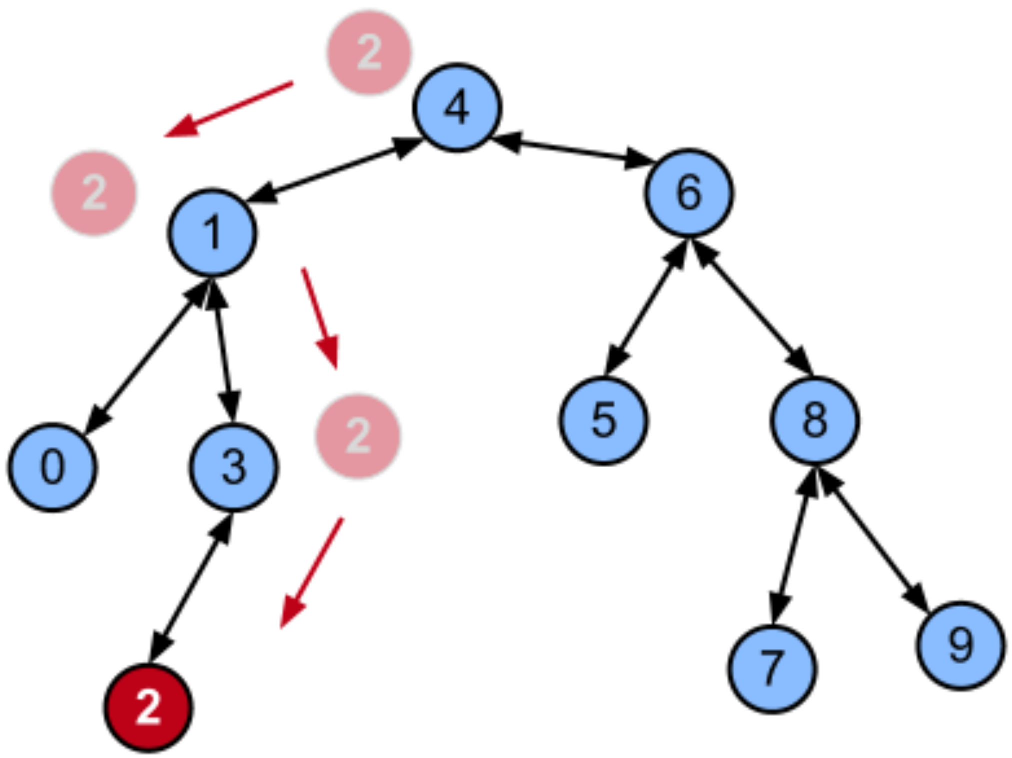
BST: Inserts

- The first node inserted is the root
- Subsequent inserted nodes are added by traversing down the BST
- Example. Insert a 2 in this:



BST: Insert Example

- Insert a **2**:



BST: Insert

- Insert order matters
- Let's build a BST from scratch together
- Data to insert:
  - 9, 3, 10, 6, 7, 4, 13, 14, 1
- What is the depth of this tree?

BST: Time Complexity

- How does BST compare to Linked Lists or Arrays in Time Complexity?

	Average	Worst Case
Space	O(n)	O(n)
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)

Hash Tables

- BSTs still take log n time to look up. What if we wanted super fast look up? Enter Hash Tables.
- Remember "Associative Arrays"?
  - In JavaScript, these are called "Objects"
  - In Ruby, these are called "Hashes"
- How are they implemented?

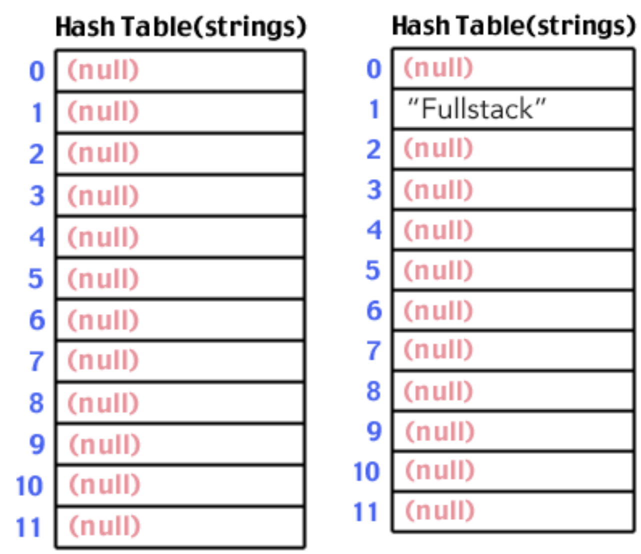
Hash Tables

- A new way to store a list of items
- Made of two parts:
  - An array to hold data
  - A hashing function
    - Looks at input and return what array cell to store data in
- Sample Hashing Function:
    - Add up all ASCII Character values of a String, and Mod it by the array size

```
function hash_me(value, array_size) {
  var asciiSum = 0;
  for(var i=0; i<value.length; i++) {
    asciiSum += value.charCodeAt(i); //each letter has a unique code
  }
  return asciiSum % array_size;
}
```

Hash Table Sample

- Say we make a new hash table of size 12
- We want to store something at the key “Fullstack”
- We run hashing function
  - hash_me(“Fullstack”,12) => returns 1
- So, store something at “Fullstack” at in index 1



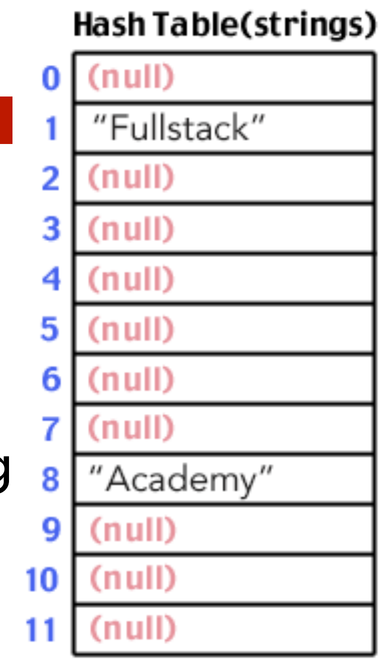
Hash Table Sample Con't

- Now let's add something at a 2nd key: "Academy"
- "Academy" hashes to 8, so we now have

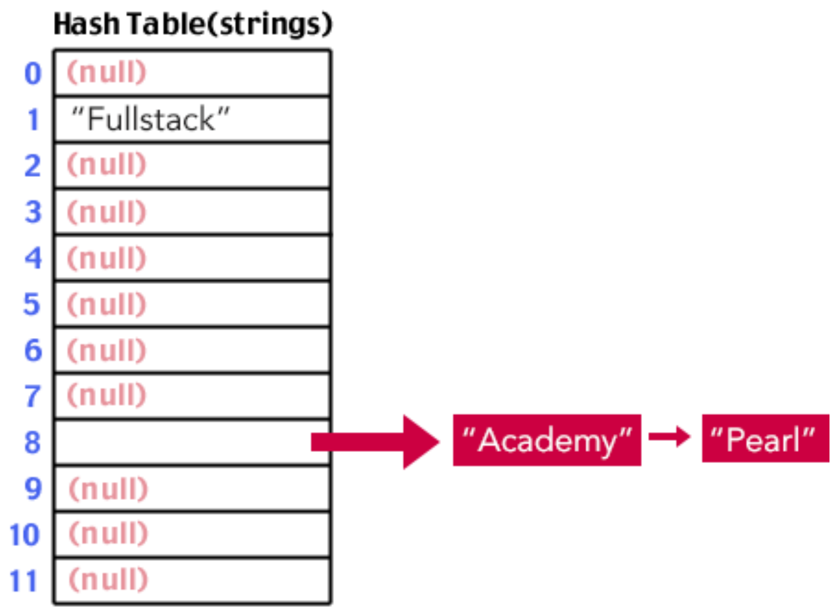
Hash Table(strings)

0	(null)
1	"Fullstack"
2	(null)
3	(null)
4	(null)
5	(null)
6	(null)
7	(null)
8	"Academy"
9	(null)
10	(null)
11	(null)

Hash Table Sample Con't

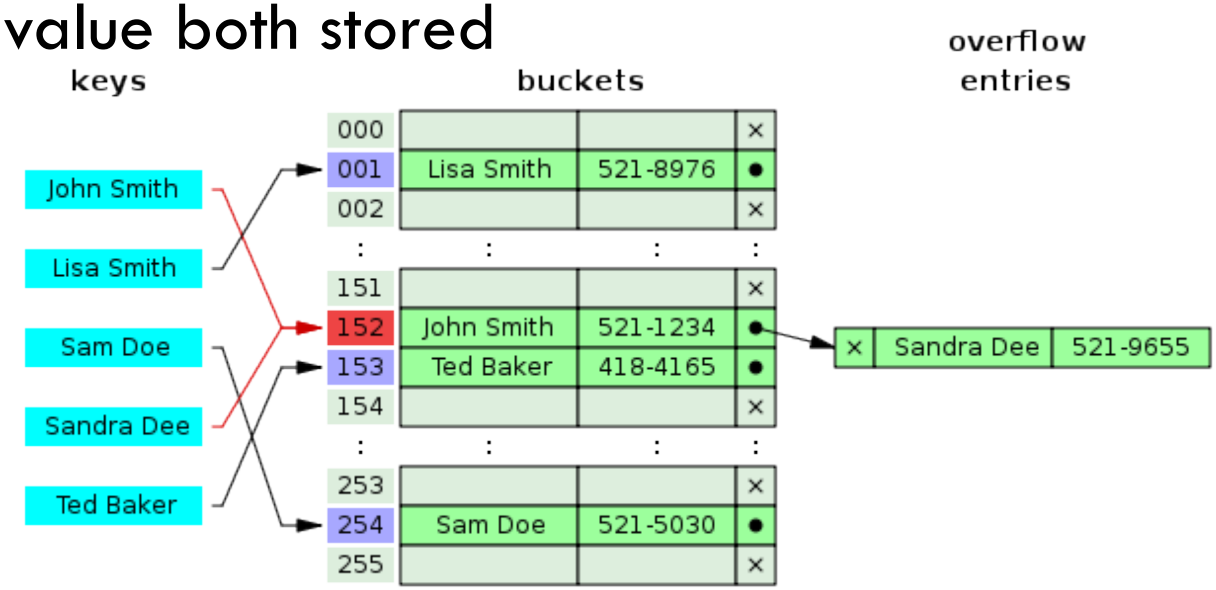


- Now, let's say we want to add "Pearl"
- "Pearl" also hashes to 8, so now what?
- **Collisions**
  - Most Hash Tables handle collisions by adding a linked list at each cell



Hash Tables with Key/Value

- This is great, but it's not useful yet. You mean we need to know the value to find the value in the array?
- In most Hash Table implementations, the hashed value is actually a key.
- The key hashes to a place in memory that has a key and value both stored



Demo

Hash Table vs BST

- Binary Search Trees (Most Trees in general)
  - Sorted keys
  - Memory Efficient
  - Worst case is not too bad
- Hash Tables
  - O(1) look-up time
  - O(1) insert time

Let's write some code!