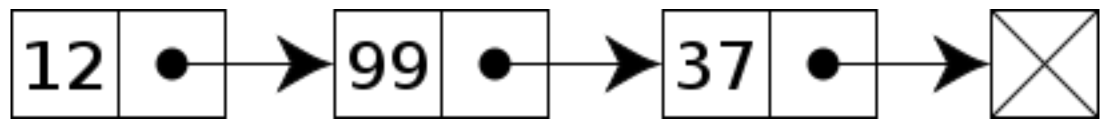
Week 4 Lecture 12

Theory

What’s in this lecture?

- Binary Search Trees

Recap: List



- A singly-linked list consists of list nodes with value and next pointers
- In general, the expected time to find an element in a list of size N is O(N)
- If we structure the pointers differently, can we do better?
- Motivation: Binary Search, or the "Guess a Number Game"

Guess a number...

- ... Between 1 and 100
- How many guesses does it take for me to guess, if you always answer "higher," "lower," or "you got it" truthfully?
- We can use a similar technique to search a sorted array...

Find() in a Sorted Array

0	1	2	3	4	5	6	7
-5	-2	1	10	14	17	23	37

- What's the fastest way to find "1"?
- What's the fastest way to detect that "18" is not present?

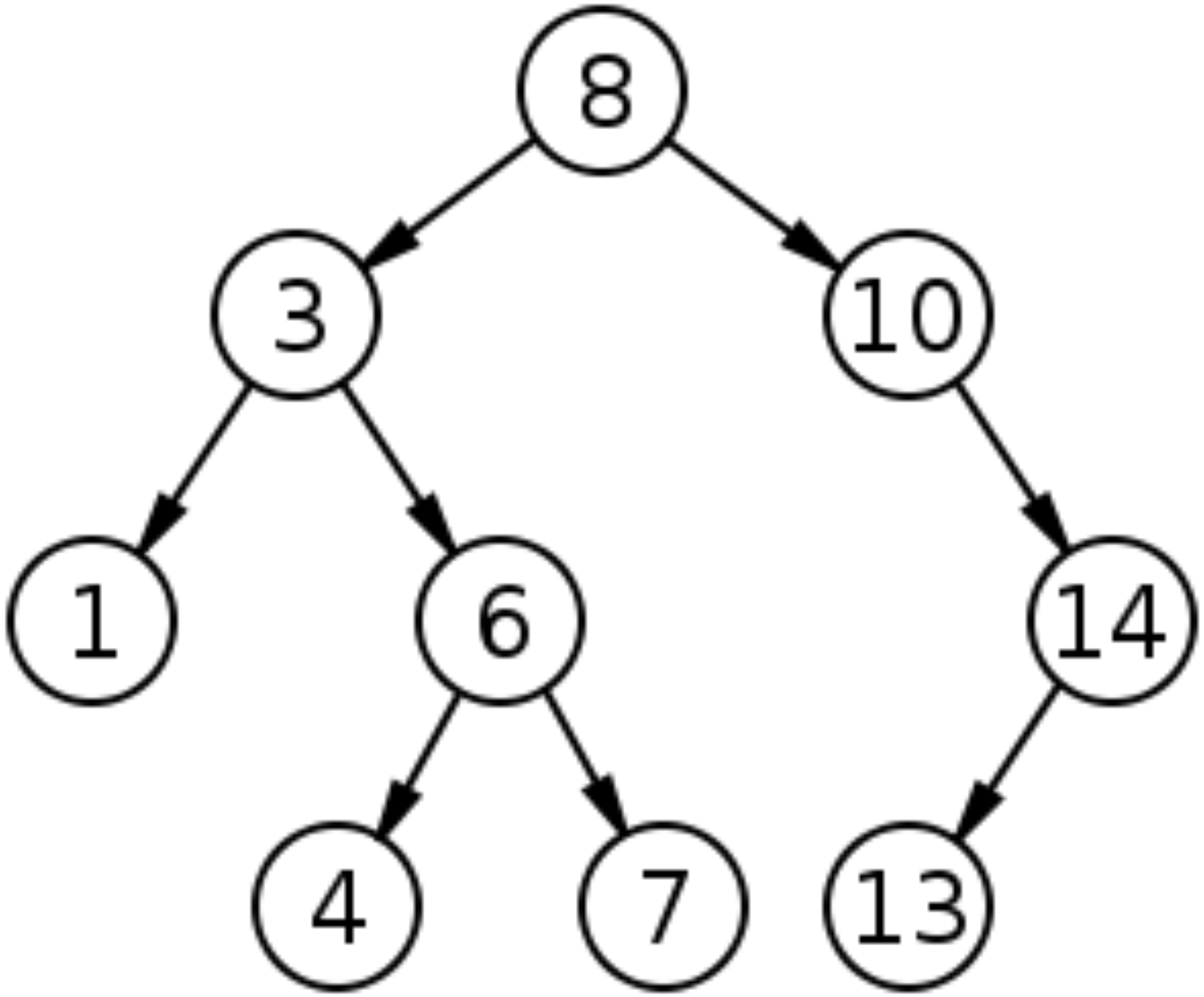
Find() in a Sorted Array

```
function binary_search(value, sorted_arr) {
  var lo = 0;
  var hi = n - 1;
  while (lo <= hi) {
    var mid = parseInt((lo + hi) / 2);
    var cur = sorted_arr[mid];
    if (cur == value) {
      return mid;
    } else if (cur < value) {
      lo = mid + 1;
    } else if (cur > value) {
      hi = mid - 1;
    }
  }
  return -1;
}
```

How fast is this?

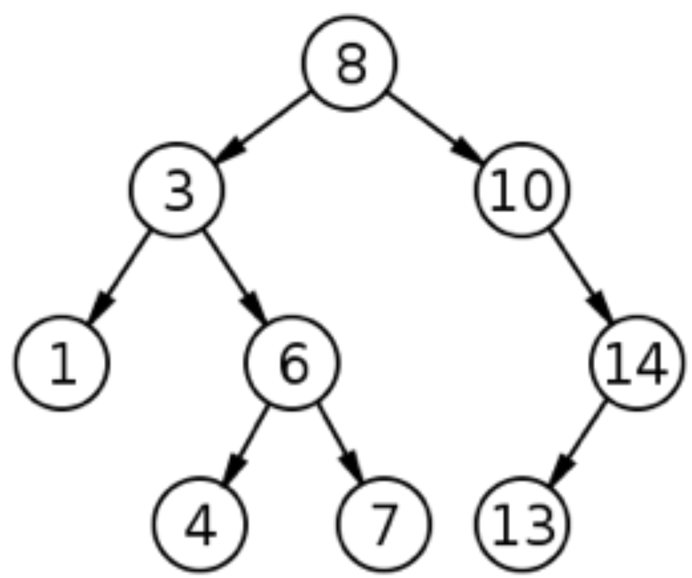
- Binary search of a sorted array of size N takes at most lg(N) checks to find (or not find) an item
- At each step, the distance between lo and hi shrinks by 1/2; this can only be done lg(N) times before lo == hi

Binary Search Tree



Binary Search Tree

- Binary Search Tree is made up of Tree Nodes that each have a value, left, and right pointers that point to children nodes (or null)
- In a balanced binary search tree, the left and right child nodes each have the same number of children
- The entry point for the tree is the root node (here, with value 8)



Make Tree Node

```
function mk_treenode(value, left, right) {
  var node = new Object();
  node[“value”] = value;
  node[“left”] = left;
  node[“right”] = right;

  return node;
}
```

Probe Value in Tree

```
function probe(value, node) {
  var cur = node["value"];
  if (cur < value && node["right"] != null) {
    return probe(value, node["right"]);
  } else if (cur > value && node["left"] != null) {
    return probe(value, node["left"]);
  }
  return node;
}
```

Tree Insert()

```
function insert(value, node) {
  var best = probe(value, node);
  var cur = best["value"];

  if (cur > value) {
    best["left"] = mk_treenode(value, null, null);
  } else if (cur < value) {
    best["right"] = mk_treenode(value, null, null);
  }
}
```

Tree Contains()

```
function contains(value, node) {
  var found = probe(value, node);
  if ((found != null) && (found["value"] == value)) {
    return true;
  }
  return false;
}
```

Tree Notes

- Balanced Binary Search Trees offer lg(N) running times for find, insert, and delete operations
- In practice, keeping trees balanced is hard
- There is a class of data structures called self-balancing trees that solves the balancing problem efficiently

Exercises

- Read Intro to Algorithms, 3rd Edition, Ch 12
- Modify the JavaScript code so that we handle duplicate values in the tree
- Implement delete() for a binary search tree
- Write a function that implements pre-, post- and in-order traversals of the tree (iterating over all tree nodes)