COMP - 285

Analysis of Algorithms

Welcome to COMP 285

Lecture 26: Final Review I

Lecturer: Chris Lucas (cflucas@ncat.edu)

HW8 Due Thursday!

Due 12/01 @ 11:59PM ET

HW8 Due Thursday!

Latest due date 12/04 @ 11:59PM ET

T-1 week until the Final!

12/06 from 2:00pm-4:00pm

T-1 week until the Final!

One double sided 8.5x11 cheat sheet!

Practice Final Released before Thanksgiving!

(Solutions posted EoW)

My Email...

Make a private post on Piazza

Big Questions!

- What can I expect on the final?
- Can we review what we've learned?
- What other questions do you have?

Big Questions!

- What can I expect on the final?
- Can we review what we've learned?
- What other questions do you have?

What's Covered?

- Material from:
  - Lecture 0 to Lecture 27
  - Homework 1 to Homework 8
  - Quiz 0 to Quiz 10

Big-Oh/Asymptotics	Data strucutres
Divide & Conquer	Sorting Algorithms
Recursion	Trees/BSTs
Graphs	Greediness
Exhaustive Search & Backtracking	Dynamic Programming
Approximation Algorithms	Complexity Theory

What's the format?

- In-Class, In-Person (**120 minutes**)
    - Bring a pencil! No tech
- You can start at **max(2:00pm, time you walk in)**
- We all stop at 4:00pm
- **40-ish** questions, **100** points total

How to prepare for the final exam?

- Reviewing written+coding homeworks
  - You will be asked to write code!
- Reviewing lectures slides/recordings, “more resources” on course website.
- Reviewing each quiz/walkthrough video
- Reviewing the practice midterm/real midterm
- Final week of lectures!
- Practice final!

Big Questions!

- What can I expect on the midterm?
- Can we review what we've learned?
- What other questions do you have?

Asymptotic Analysis!

What is Big-O?

- Algorithms are judged by their **correctness** and **efficiency** (time efficiency and space efficiency).

- Big-O is **how we quantify efficiency**; it gives us a way to compare different algorithms to say which are better than others.

What is Big-O? (pt. 2)

- Big-O is a way to express the algorithm's efficiency in terms of the size of its input (which we often call "N").

- Big-O communicates an upper-bound on how many "operations" an algorithm will take.

What’s the runtime and space complexity?

```cpp
void printElements(const std::vector<int>& vec) {
 std::cout << "Printing..." << std::endl;
 for(int i = 0; i < 100; i++) {
   for(int j = 0; j < vec.size(); j++) {
     std::cout << i << " " << vec[j] << " ";
   }
 }
 std::cout << std::endl;
}
```

What's the runtime and space complexity?

```
void printElements(const std::vector<int>& vec) {
 std::cout << "Printing..." << std::endl;
 for(int i = 0; i < 100; i++) {
   for(int j = 0; j < vec.size(); j++) {
     std::cout << i << " " << vec[j] << " ";
   }
 }
 std::cout << std::endl;
}
```

O(N) runtime
O(1) space

What’s the runtime and space complexity?

```
int someFunction (vector<int> v) {
    vector<int> result;
    int index = min<int>(v.size(), 1);
    for (int i = 0; i < index; i++) {
        result.push_back(i);
    }
    return result.size();
}
```

What's the runtime and space complexity?

```cpp
int someFunction (vector<int> v) {
    vector<int> result;
    int index = min<int>(v.size(), 1);
    for (int i = 0; i < index; i++) {
        result.push_back(i);
    }
    return result.size();
}
```

O(1) runtime
O(1) space

What’s the runtime and space complexity?

```
int someFunction (vector<int> v) {
    vector<int> result;
    int index = max<int>(v.size(), 1);
    for (int i = 0; i < index; i++) {
        result.push_back(i);
    }
    return result.size();
}
```

What's the runtime and space complexity?

```
int someFunction (vector<int> v) {
    vector<int> result;
    int index = max<int>(v.size(), 1);
    for (int i = 0; i < index; i++) {
        result.push_back(i);
    }
    return result.size();
}
```

O(N) runtime
O(N) space

What's the runtime?

```
vector<int> someFunction1(vector<int> v) {
    vector<int> result;
    for (int i = 0; i < v.size(); i+=2) {
        result.push_back(i);
    }
    return result;
}

vector<int> someFunction2(vector<int> v) {
    vector<int> result;
    for (int i = 0; i < v.size(); i*=2) {
        result.push_back(i);
    }
    return result;
}
```

Runtime poll?

1. O(SF1) < O(SF2)
2. O(SF1) = O(SF2)
3. O(SF1) > O(SF2)

What's the runtime?

```
vector<int> someFunction1(vector<int> v) {
    vector<int> result;
    for (int i = 0; i < v.size(); i+=2) {
        result.push_back(i);
    }
    return result;
}

vector<int> someFunction2(vector<int> v) {
    vector<int> result;
    for (int i = 0; i < v.size(); i*=2) {
        result.push_back(i);
    }
    return result;
}
```

Runtime poll?

1. O(SF1) < O(SF2)
2. O(SF1) = O(SF2)
3. O(SF1) > O(SF2)

SF1 = O(N)
SF2 = O(logN)

What's the space complexity?

```
vector<int> someFunction1(vector<int> v) {
    vector<int> result;
    for (int i = 0; i < v.size(); i+=2) {
        result.push_back(i);
    }
    return result;
}

vector<int> someFunction2(vector<int> v) {
    vector<int> result;
    for (int i = 0; i < v.size(); i*=2) {
        result.push_back(i);
    }
    return result;
}
```

Space complexity poll?

1. O(SF1) < O(SF2)
2. O(SF1) = O(SF2)
3. O(SF1) > O(SF2)

What’s the space complexity?

```
vector<int> someFunction1(vector<int> v) {
    vector<int> result;
    for (int i = 0; i < v.size(); i+=2) {
        result.push_back(i);
    }
    return result;
}

vector<int> someFunction2(vector<int> v) {
    vector<int> result;
    for (int i = 0; i < v.size(); i*=2) {
        result.push_back(i);
    }
    return result;
}
```

Space complexity poll?

1. O(SF1) < O(SF2)
2. O(SF1) = O(SF2)
3. O(SF1) > O(SF2)

SF1 = O(N)
SF2 = O(logN)

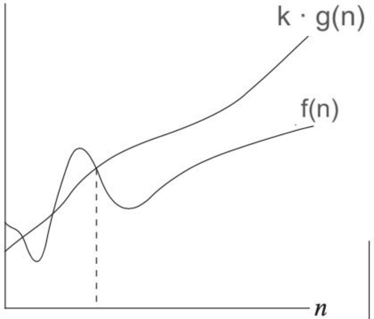
Recap

Upper-bound | f = O(g) is similar to f ≤ g
"f grows no faster than g"

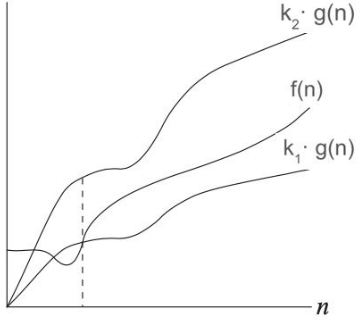
Tight-bound | f = Θ(g) is similar to f = g
"f grows as fast as g"

Lower-bound | f = Ω(g) is similar to f ≥ g
"f grows no slower than g"

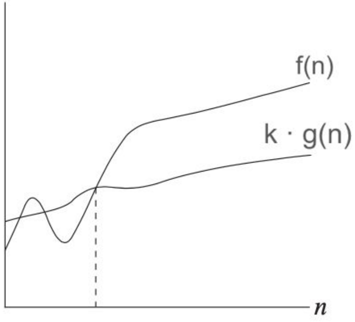
If both f = O(g) and f = Ω(g), then f = Θ(g)
If f = O(g), then g = Ω(f)



$f(n) = O(g(n))$



$f(n) = \Theta(g(n))$



$f(n) = \Omega(g(n))$

Asymptotic Analysis Practice

1. $O(n/100 + \log(n) + 200)$ can be simplified to $O(n)$. True or False?

2. $2x + x^2/2 = \Theta(x^2 + 2x + x\log(x))$. True or False?

3. $x + 20 = \Omega(999)$. True or False?

Asymptotic Analysis Practice

1. $O(n/100 + \log(n) + 200)$ can be simplified to $O(n)$. True or False?
   True

2. $2x + x^2/2 = \Theta(x^2 + 2x + x\log(x))$. True or False?
   True

3. $x + 20 = \Omega(999)$. True or False?
   True

Sorting!

Sorting

When comparing different sorting algorithms, these are some of the properties we care about:

- Best-case/worst-case/average-case time complexity
- In-place: can we use only O(1) additional space?
- Adaptive: does it run faster if the array is partially sorted?
- Stable: will elements of the same value stay ordered relative to each other?

Selection Sort Pseudocode

```
algorithm selectionSort
  Input: vector<int> vec of size N
  Output: vector<int> with sorted elements

  for index i = 0, 1, 2, ..., N-2
    min_index = i
    for j = i+1, i+2, ..., N-1
      if vec[j] < vec[min_index]
        min_index = j
    temp = vec[i]
    vec[i] = vec[min_index]
    vec[min_index] = temp
```

What's the tight upper-bound on the:

- Best-case runtime? $O(n^2)$
- Worst-case runtime? $O(n^2)$
- Average-case runtime? $O(n^2)$
- Worst-case space complexity? O(1)

Is this adaptive? No

https://visualgo.net/en/sorting

Insertion Sort In-Place Pseudocode

```
algorithm insertionSort
  Input: vector<int> vec of size N
  Output: vector<int> with sorted elements

  for index i = 1, 2, ..., N-1
    next = vec[i]
    j = i-1
    while j >= 0 and vec[j] > next
      vec[j+1] = vec[j]
      j = j-1
    vec[j+1] = next
```

What's the tight upper-bound on the:

- Best-case runtime? O(n)
- Worst-case runtime? $O(n^2)$
- Average-case runtime? $O(n^2)$
- Worst-case space complexity? O(1

Is this adaptive? Yes

https://visualgo.net/en/sorting

QuickSort: Pseudocode

```
algorithm quickSort
  Input: vector<int> vec of size N
  Output: vector<int> with sorted elements

  if N < 2
    return vec
  pivot = vec[N-1]
  left = new empty vec
  right = new empty vec
  for index i = 0, 1, 2, ... N-2
    if vec[i] <= pivot
      left.push_back(vec[i])
    else
      right.push_back(vec[i])
  return quickSort(left) + [pivot] + quickSort(right)
```

What's the tight upper-bound on the:

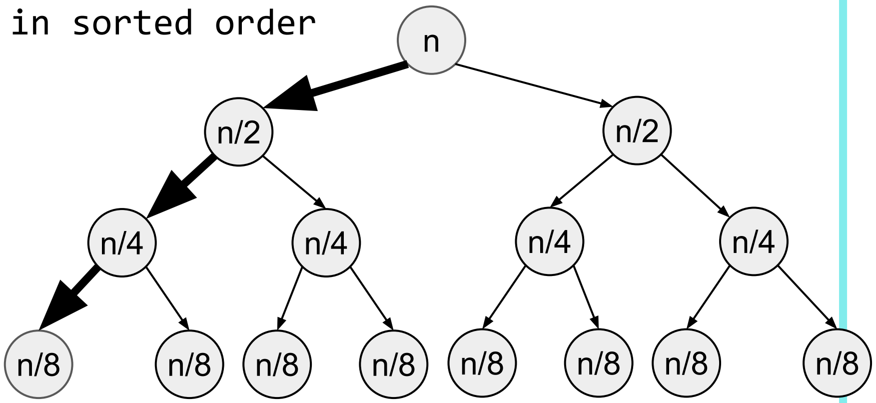
- Best-case runtime? O(nlogn)
- Worst-case runtime? $O(n^2)$
- Average-case runtime? O(nlogn)
- Worst-case space complexity?$O(n^2)$

Is it adaptive? NO

MergeSort Pseudocode

```
algorithm mergeSort
  Input: vector of ints vec of size N
  Output: vec such that its elements are in sorted order

  if N <= 1
    return vec
  midpoint = floor(N/2)
  left = mergeSort(vec[0 to midpoint])
  right = mergeSort(vec[midpoint to N])
  return merge(left, right)
```



- Best-Case Runtime? O(n log(n))
- Average-Case Runtime? O(n log(n))
- Worst-Case Runtime? O(n log(n))
- Worst-Case Space complexity? O(n)

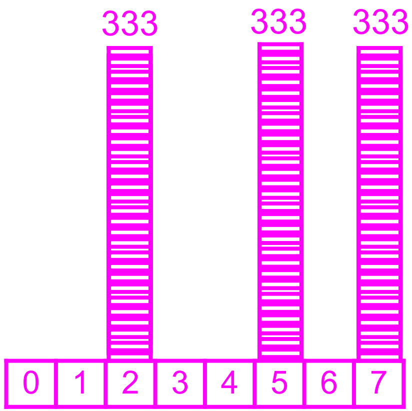
n total work happening at each level, with log(n) levels.

Counting Sort Pseudocode

input = {7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, … , 7, 7}
count = {0, 0, 0, 0, 0, 0, 0, 999}

input = {7, 7, 7, 7, … , 2, 2, 2, 2, … 5, 5, 5, 5}
count = {0, 0, 333, 0, 0, 333, 0, 333}

```
algorithm countingSort
  Input: vector<int> vec and an integer k where every int in vec
is between 0 and k
  Output: vector<int> in sorted order
// count each element using buckets
for (int i = 0; i < vec.size(); i++) {
    elem = input[i];
    buckets[elem] += 1;
}
for (int i = 1; i < k; i++) {
    count[i] += count[i-1];
}
for (int i = input.size()-1; i >= 0; i--) {
    elem = input[i];
    count[i] -= 1;
    output[count[i]] = elem;
}
```



n is vec.size() and k is value of k.

- Best-Case Runtime? O(n + k)
- Average-Case Runtime? O(n + k)
- Worst-Case Runtime? O(n + k)
- Worst-Case Space complexity? O(n + k)

Sorting Practice

10. Which array of the following will CountingSort take the most number of steps on? Select **ONE.**
    a. [1, 2, 3, 4, 5, 6]
    b. [5, 43, 3, 11, 6, 9]
    c. [3, 1, 34, 3, 4, 81]
    d. [4, 4754, 4, 24, 1, 33]

11. For each of the below, explain in 1 - 2 sentences what they mean with respect to sorting.
    - Adaptive
    - Stability
    - In-Place

12. Given an array is already sorted, which sort will take the least time? Select **ONE.**
    a. Insertion Sort
    b. Quick Sort
    c. Merge Sort
    d. Selection Sort

Sorting Practice

10. Which array of the following will CountingSort take the most number of steps on? Select **ONE.**
    a. [1, 2, 3, 4, 5, 6]
    b. [5, 43, 3, 11, 6, 9]
    c. [3, 1, 34, 3, 4, 81]
    d. [4, 4754, 4, 24, 1, 33]

11. For each of the below, explain in 1 - 2 sentences what they mean with respect to sorting.
    - Adaptive

    If a sorting algorithm is adaptive, it will run more efficiently if the array is more sorted.

    - Stable

    If a sorting algorithm is stable, elements of the same value will stay ordered relative to each other in the output. For example {1, 4, 1*, 2} → {1, 1*, 2, 4} would be a stable sort, because the star 1 is to the right of the non-starred 1 in both the input and output.

    - In-Place

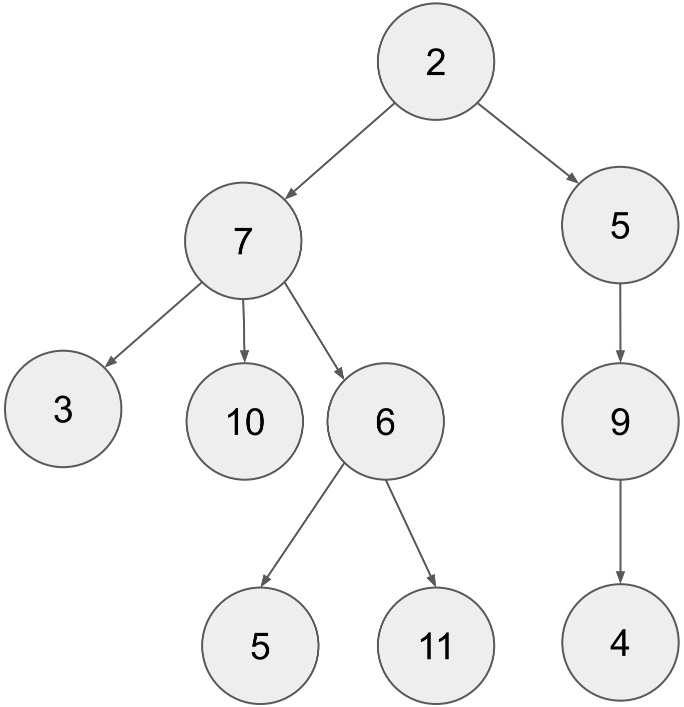
    If a sorting algorithm is in-place, we only use O(1) additional space.

12. Given an array is already sorted, which sort will take the least time? Select **ONE.**
    a. Insertion Sort
    b. Quick Sort
    c. Merge Sort
    d. Selection Sort

Trees!

Trees

- A Tree is a **hierarchical** data structure that has a value and children. Each child is also a Tree, making this data structure **recursive** in nature.
- Don't confuse general N-ary Trees with Binary Trees (a special kind of tree where each node has at most two children) or Binary Search Trees (a special kind of binary tree where left subtree is less and right subtree is greater).



Binary tree terminology

Each node has two children.

The left child of $\boxed{3}$ is $\boxed{2}$

The right child of $\boxed{3}$ is $\boxed{4}$

The parent of $\boxed{3}$ is $\boxed{5}$

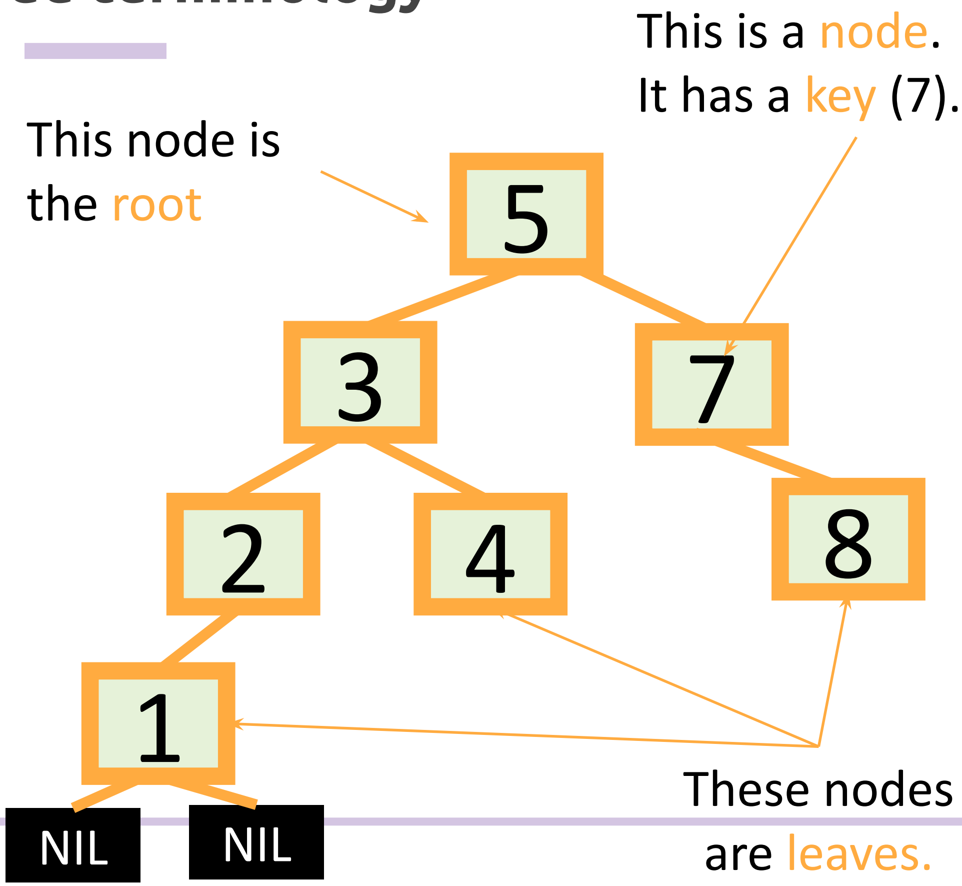
$\boxed{2}$ is a descendant of $\boxed{5}$

Each node has a pointer to its left child, right child, and parent.

Both children of $\boxed{1}$ NIL.

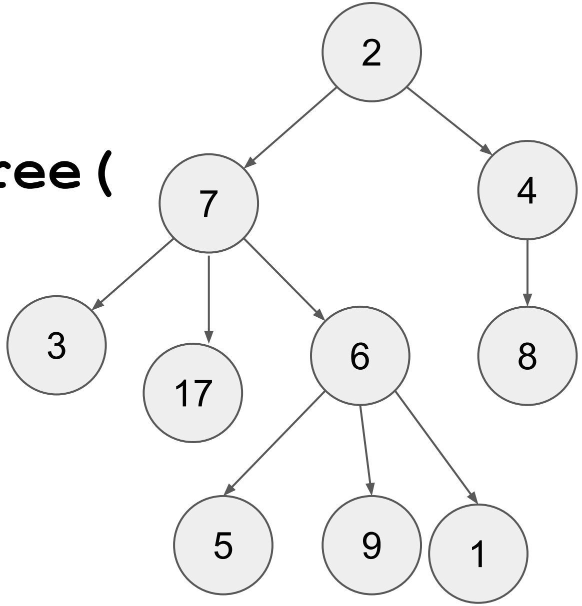
(I won't usually draw them).

The height of this tree is 3. (Max number of edges from the root to a leaf).



findHeightOfNAryTree

Write an algorithm that takes in a tree, and returns the height of the tree.

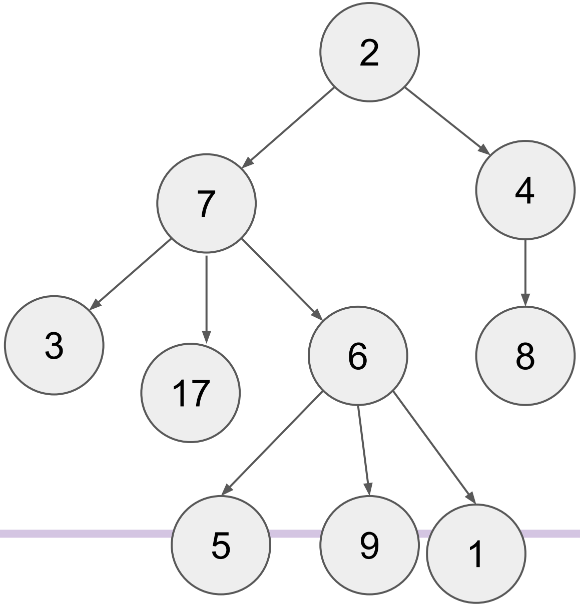
findHeightOfTree(

) outputs 3

Let’s code it!!!

findHeightOfNAryTree

Write an algorithm that takes in an n-ary tree, and returns the height of the tree.



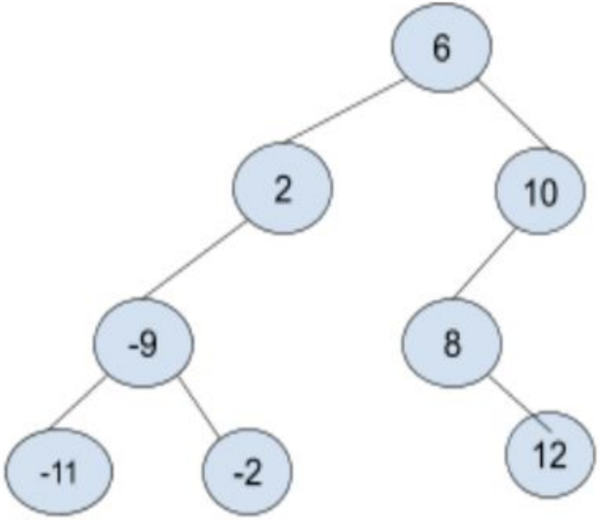

```
int findHeightOfTree(TreeNode<int> *root) {

}
```

Trees Practice

17. Is the tree on the right a Binary Search Tree? Explain.

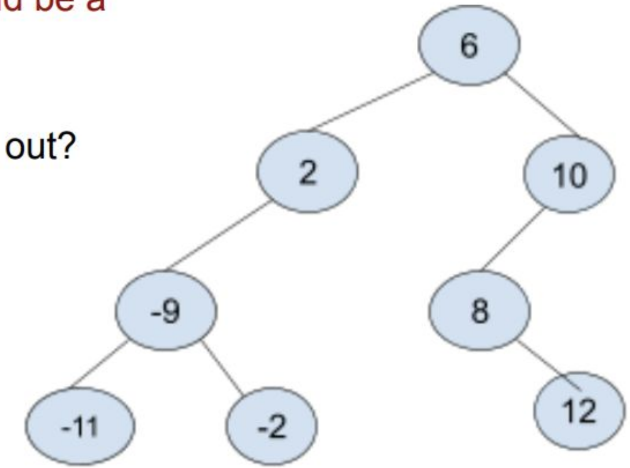
18. What would an post-order traversal of this tree print out?



Trees Practice

17. Is the tree on the right a Binary Search Tree? Explain.
    No. The left subtree of 10 contains 12, which is greater than it. If the 12 were a 9, for example, then it would be a Binary Search Tree.

18. What would a post-order traversal of this tree print out?
    -11, -6, -9, -2, 12, 8, 10, 6



Big Questions!

- Can we review what we've learned?
- What other questions do you have?

COMP - 285

Analysis of Algorithms

Welcome to COMP 285

Lecture 26: Final Review I

Lecturer: Chris Lucas (cflucas@ncat.edu)