CSE 373

OCTOBER 9TH – AMORTIZED ANALYSIS

TODAY

- Master Theorem
- Amortized Analysis
- Binary Search Trees

REVIEW

- **Algorithm Analysis**
  - Asymptotic behavior

REVIEW

- **Algorithm Analysis**
  - Asymptotic behavior
- **Loops and iterations**

REVIEW

- **Algorithm Analysis**
  - Asymptotic behavior
- **Loops and iterations**
- **Recursive functions**

REVIEW

- **Algorithm Analysis**
  - Asymptotic behavior
- **Loops and iterations**
- **Recursive functions**
  - Recurrence relations

ANALYSIS

- **On Friday, we showed the formal recurrence approach**
  - Break into recursive, non-recursive
  - Compute non-recursive computation time
  - Produce the recurrence
  - Roll out the recurrence and produce the closed form
  - Upper-bound the closed form with bigO notation

ANALYSIS

- **While this process is important, we can save some steps if all we care about is the upper bound**
  - bigO notation eliminates the need for constants
  - Lots of our messing around with c_0 and c_1 doesn't come through to the solution

ANALYSIS

- **Master theorem**
  - These recurrences all follow a similar pattern

ANALYSIS

- **Master theorem**
  - These recurrences all follow a similar pattern
  - Therefore, if you can produce a recurrence, there is actually a procedural way to produce solutions

ANALYSIS

- **Master theorem**
  - These recurrences all follow a similar pattern
  - Therefore, if you can produce a recurrence, there is actually a procedural way to produce solutions
  - If $T(n) = a*T(n/b)+n^c$ for $n > n_0$ and if the base case is a constant

ANALYSIS

- **Master theorem**
  - These recurrences all follow a similar pattern
  - Therefore, if you can produce a recurrence, there is actually a procedural way to produce solutions
  - If $T(n) = a*T(n/b)+n^c$ for $n > n_0$ and if the base case is a constant
    - Case 1: $\log_b(a) < c$: $T(n) = O(n^c)$
    - Case 2: $\log_b(a) = c$: $T(n) = O(n^c \lg n)$
    - Case 3: $\log_b(a) > c$: $T(n) = O(n^{\log a})$

ANALYSIS

- Recurrences come up all the time

ANALYSIS

- **Recurrences come up all the time**
  - Analyze methods and iterative approaches through the normal methods
  - Recursive functions use a recurrence
  - Possible to get to bigO solution quickly
  - Usually for worst-case analysis

ANALYSIS

- **Final analysis type**

ANALYSIS

- **Final analysis type**
  - Worst-case

ANALYSIS

- **Final analysis type**
  - Worst-case
    - Consider adding to an unsorted array

ANALYSIS

- **Final analysis type**
  - Worst-case
    - Consider adding to an unsorted array
    - Resizing is the costly O(n) operation

ANALYSIS

- **Final analysis type**
  - Worst-case
    - Consider adding to an unsorted array
    - Resizing is the costly O(n) operation
    - This occurs in predictable ways

ANALYSIS

- **Final analysis type**
  - Worst-case
    - Consider adding to an unsorted array
    - Resizing is the costly O(n) operation
    - This occurs in predictable ways
    - Do these types of operations really slow down the function?

AMORTIZED ANALYSIS

- Adding to unsorted array

AMORTIZED ANALYSIS

- **Adding to unsorted array**
  - How long does it take to add *n* elements into the array?

AMORTIZED ANALYSIS

- **Adding to unsorted array**
  - How long does it take to add *n* elements into the array?
  - Let's say the array is full with *n* elements and we add *n* more

AMORTIZED ANALYSIS

- **Adding to unsorted array**
  - How long does it take to add *n* elements into the array?
  - Let's say the array is full with *n* elements and we add *n* more
  - It takes n-1*O(1) + 1*O(n) = O(n)

AMORTIZED ANALYSIS

- **Adding to unsorted array**
  - How long does it take to add *n* elements into the array?
  - Let's say the array is full with *n* elements and we add *n* more
  - It takes n-1*O(1) + 1*O(n) = O(n)
  - Amortized over the whole set of operations, each one is only O(1) time

AMORTIZED ANALYSIS

- **Adding to unsorted array**
  - How long does it take to add *n* elements into the array?
  - Let's say the array is full with *n* elements and we add *n* more
  - It takes n-1*O(1) + 1*O(n) = O(n)
  - Amortized over the whole set of operations, each one is only O(1) time
  - What does this depend on?

AMORTIZED ANALYSIS

- **Adding to unsorted array**
  - How long does it take to add *n* elements into the array?
  - Let's say the array is full with *n* elements and we add *n* more
  - It takes n-1*O(1) + 1*O(n) = O(n)
  - Amortized over the whole set of operations, each one is only O(1) time
  - What does this depend on?
    - **Doubling the array**

AMORTIZED ANALYSIS

- **Adding to unsorted array**
  - What if we only add some constant number to the array?
  - Let's resize and add 10,000 elements every time
  - How long does it take to add *n* elements?
  - n-n/10,000*O(1) + n/10,000*O(n)

AMORTIZED ANALYSIS

- **Adding to unsorted array**
  - What if we only add some constant number to the array?
  - Let's resize and add 10,000 elements every time
  - How long does it take to add *n* elements?
  - n-n/10,000*O(1) + n/10,000*O(n) = $O(n^2)$
  - This is for any constant, regardless of how large

AMORTIZED ANALYSIS

- Amortization the average runtime over repeated calls to the same function

AMORTIZED ANALYSIS

- **Amortization the average runtime over repeated calls to the same function**
  - If the worst case happens in predictable ways (i.e. every *n* inserts), then the costly operation doesn't increase the total asymptotic runtime of multiple operations

AMORTIZED ANALYSIS

- **Amortization the average runtime over repeated calls to the same function**
  - If the worst case happens in predictable ways (i.e. every *n* inserts), then the costly operation doesn't increase the total asymptotic runtime of multiple operations
  - Over *n* operations, remember to divide the total runtime by *n*

DICTIONARIES

- **Back to the dictionary problem**
  - Can we apply these analytical tools to some simple implementations?

IMPLEMENTATIONS

- **Simple implementations**

IMPLEMENTATIONS

- **Simple implementations**

	`insert`	`find`	`delete`
Unsorted linked-list	*O(n)**	*O(n)*	*O(n)*
Unsorted array	*O(n)**	*O(n)*	*O(n)*
Sorted linked list	*O(n)*	*O(n)*	*O(n)*
Sorted array	*O(n)*	*O(log n)*	*O(n)*

* Because we need to check for duplicates

IMPLEMENTATIONS

- Other implementations?

IMPLEMENTATIONS

- **Other implementations?**
  - Binary Search Tree (BST)

IMPLEMENTATIONS

- **Other implementations?**
  - Binary Search Tree (BST)
  - Sort based on keys (which have to be comparable)

IMPLEMENTATIONS

- **Other implementations?**
  - Binary Search Tree (BST)
  - Sort based on keys (which have to be comparable)
  - How do we implement this?

BINARY SEARCH TREE

- Review

BINARY SEARCH TREE

- **Review**
  - What is a binary search tree?

BINARY SEARCH TREE

- **Review**
  - What is a binary search tree?
    - A rooted tree, where each node has at most two children

BINARY SEARCH TREE

- **Review**
  - What is a binary search tree?
    - A rooted tree, where each node has at most two children
    - All elements less than the root are in the left subtree and all elements larger than the root are in the right subtree

BINARY SEARCH TREE

- **Review**
  - What is a binary search tree?
    - A rooted tree, where each node has at most two children
    - All elements less than the root are in the left subtree and all elements larger than the root are in the right subtree
    - All, subtrees must also be binary search trees

BINARY SEARCH TREE

- **Review**
  - What is a binary search tree?
    - A rooted tree, where each node has at most two children
    - All elements less than the root are in the left subtree and all elements larger than the root are in the right subtree
    - All, subtrees must also be binary search trees
  - With this property, all binary search trees have sorted in-order traversals

IMPLEMENTATIONS

- **Other implementations?**
  - Binary Search Tree (BST)
  - Sort based on keys (which have to be comparable)
  - How do we implement this?
  - What changes need to be made?

IMPLEMENTATIONS

- **BST Node:**
  - Before:

IMPLEMENTATIONS

- **BST Node:**
  - Before:
    - Node left
    - Node right
    - Value data

IMPLEMENTATIONS

- **BST Node:**
  - Before:
    - Node left
    - Node right
    - Value data
  - Now?

IMPLEMENTATIONS

- **BST Node:**
  - Before:
    - Node left
    - Node right
    - Value data
  - Now?
    - Node left
    - Node right

IMPLEMENTATIONS

- **BST Node:**
  - Before:
    - Node left
    - Node right
    - Value data
  - Now?
    - Node left
    - Node right
    - Key k
    - Value v

IMPLEMENTATIONS

- **BST Changes:**
  - Insert() and find() remain similar

IMPLEMENTATIONS

- **BST Changes:**
  - Insert() and find() remain similar
  - Key is the primary comparison

IMPLEMENTATIONS

- **BST Changes:**
  - Insert() and find() remain similar
  - Key is the primary comparison
  - Value is attached to the key

IMPLEMENTATIONS

- **BST Changes:**
  - Insert() and find() remain similar
  - Key is the primary comparison
  - Value is attached to the key
  - **Dictionary fact: All values have an associated key**

IMPLEMENTATIONS

- **BST Changes:**
  - Insert() and find() remain similar
  - Key is the primary comparison
  - Value is attached to the key
  - **Dictionary fact: All values have an associated key**
  - **All keys are unique, i.e. each key only has one value**

IMPLEMENTATIONS

- **BST Analysis:**
  - What is our time for the three functions?

IMPLEMENTATIONS

- **BST Analysis:**
  - What is our time for the three functions?
    - Insert()? Delete()? Find()?
    - *Consider best and worst-case.*
    - *What are the inputs for best and worst-case?*

IMPLEMENTATIONS

- **BST Analysis:**
  - Insert():

IMPLEMENTATIONS

- **BST Analysis:**
  - Insert():
    - Worst case:

IMPLEMENTATIONS

- **BST Analysis:**
  - Insert():
    - Worst case: O(n)

IMPLEMENTATIONS

- **BST Analysis:**
  - Insert():
    - Worst case: O(n). *What is this worst case?*

IMPLEMENTATIONS

- **BST Analysis:**
  - Insert():
    - Worst case: O(n)
    - Best case:

IMPLEMENTATIONS

- **BST Analysis:**
  - Insert():
    - Worst case: O(n)
    - Best case: O(1)
    - What is the general case here?
      - What does the runtime for a particular insert depend on?
      - Height of the tree

HEIGHT REVIEW

- **Height**

HEIGHT REVIEW

- **Height**
  - In this class, we set the height of an empty tree to be equal to -1

HEIGHT REVIEW

- **Height**
  - In this class, we set the height of an empty tree to be equal to -1
  - This makes the height of a single node 0

HEIGHT REVIEW

- **Height**
  - In this class, we set the height of an empty tree to be equal to -1
  - This makes the height of a single node 0
  - How do you calculate the height of a large tree?

HEIGHT REVIEW

- **Height**
  - In this class, we set the height of an empty tree to be equal to -1
  - This makes the height of a single node 0
  - How do you calculate the height of a large tree?
    - Height = 1 + max(height(left),height(right))

IMPLEMENTATIONS

- **BST Analysis:**
  - Find():

IMPLEMENTATIONS

- **BST Analysis:**
  - Find():
    - Worst-case:

IMPLEMENTATIONS

- **BST Analysis:**
  - Find():
    - Worst-case: O(n)

IMPLEMENTATIONS

- **BST Analysis:**
  - Find():
    - Worst-case: O(n)
    - What is this case?

IMPLEMENTATIONS

- **BST Analysis:**
  - Find():
    - Worst-case: O(n)
    - What is this case? *When the tree is linear*

IMPLEMENTATIONS

- **BST Analysis:**
  - Find():
    - Worst-case: O(n)
    - What is this case? *When the tree is linear*
    - Best-case: O(1)

IMPLEMENTATIONS

- **BST Analysis:**
  - Find():
    - Worst-case: O(n)
    - What is this case? *When the tree is linear*
    - Best-case: O(1) *When the item is the root*

IMPLEMENTATIONS

- **BST Analysis:**
  - Find():
    - Worst-case: O(n)
    - What is this case? *When the tree is linear*
    - Best-case: O(1) *When the item is the root*
    - Generally, however: O(log n) when the tree is balanced

IMPLEMENTATIONS

- **BST Analysis:**
  - Delete():

IMPLEMENTATIONS

- **BST Analysis:**
  - Delete():
    - What are some strategies for deleting?

IMPLEMENTATIONS

- **BST Analysis:**
  - Delete():
    - What are some strategies for deleting?
    - Are there any cases where deleting is easy?

IMPLEMENTATIONS

- **BST Analysis:**
  - Delete():
    - What are some strategies for deleting?
    - Are there any cases where deleting is easy?
    - Case 0: The element is not in the data structure

IMPLEMENTATIONS

- **BST Analysis:**
  - Delete():
    - What are some strategies for deleting?
    - Are there any cases where deleting is easy?
    - Case 0: The element is not in the data structure
      - Don’t change the data, possibly throw an exception

IMPLEMENTATIONS

- **BST Analysis:**
  - Delete():
    - What are some strategies for deleting?
    - Are there any cases where deleting is easy?
    - Case 0: The element is not in the data structure
      - Don’t change the data, possibly throw an exception
    - Case 1: The key is a leaf in the tree

IMPLEMENTATIONS

- **BST Analysis:**
  - Delete():
    - What are some strategies for deleting?
    - Are there any cases where deleting is easy?
    - Case 0: The element is not in the data structure
      - Don’t change the data, possibly throw an exception
    - Case 1: The key is a leaf in the tree
      - Remove the pointer to that node

IMPLEMENTATIONS

- **BST Analysis:**
  - Delete():
    - What are some strategies for deleting?
    - Are there any cases where deleting is easy?
    - Case 0: The element is not in the data structure
      - Don’t change the data, possibly throw an exception
    - Case 1: The key is a leaf in the tree
      - Remove the pointer to that node
    - Case 2: The node has one child

IMPLEMENTATIONS

- **BST Analysis:**
  - Delete():
    - What are some strategies for deleting?
    - Are there any cases where deleting is easy?
    - Case 0: The element is not in the data structure
      - Don’t change the data, possibly throw an exception
    - Case 1: The key is a leaf in the tree
      - Remove the pointer to that node
    - Case 2: The node has one child
      - Replace that node with its child

IMPLEMENTATIONS

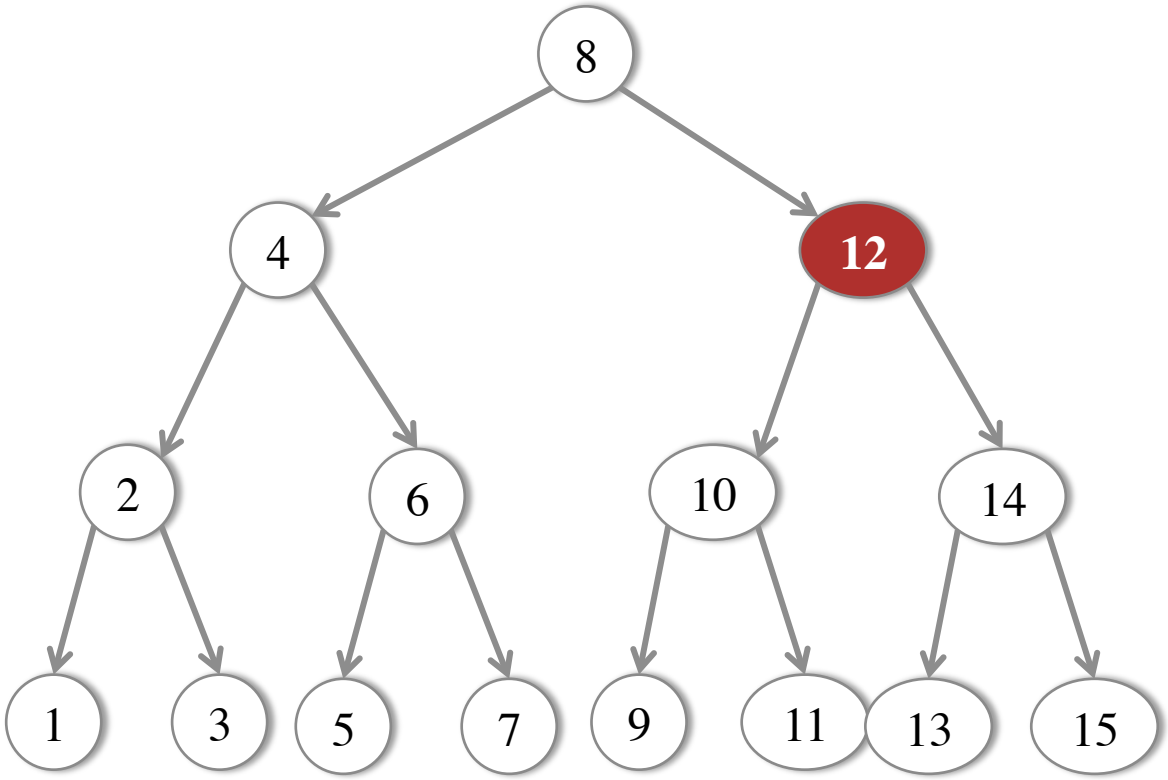
- **BST Analysis:**
  - Delete():
    - What are some strategies for deleting?
    - Are there any cases where deleting is easy?
    - Case 0: The element is not in the data structure
      - Don’t change the data, possibly throw an exception
    - Case 1: The key is a leaf in the tree
      - Remove the pointer to that node
    - Case 2: The node has one child
      - Replace that node with its child
    - Case 3: The node has two children

IMPLEMENTATIONS

- **BST Analysis:**
  - Delete():
    - What are some strategies for deleting?
    - Are there any cases where deleting is easy?
    - Case 0: The element is not in the data structure
      - Don’t change the data, possibly throw an exception
    - Case 1: The key is a leaf in the tree
      - Remove the pointer to that node
    - Case 2: The node has one child
      - Replace that node with its child
    - Case 3: The node has two children
      - *What are some possible strategies?*

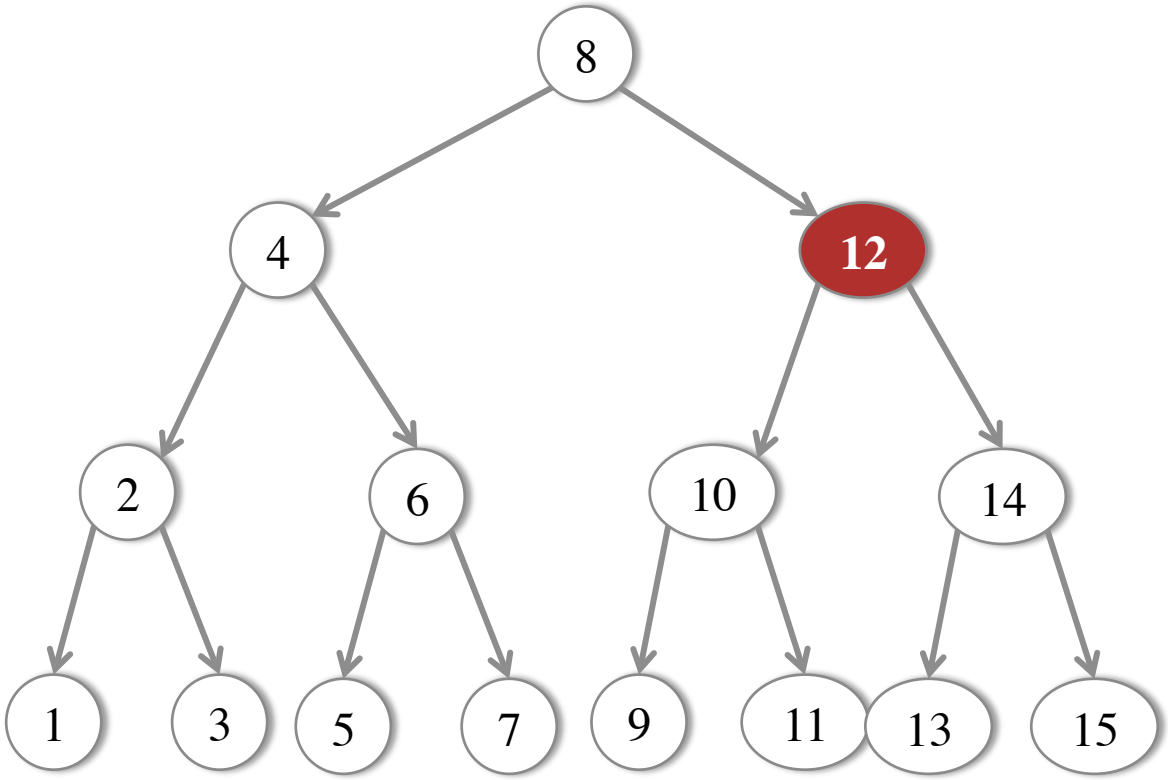
IMPLEMENTATIONS

- **Deleting nodes with 2 children**
  - *How do we delete 12?*



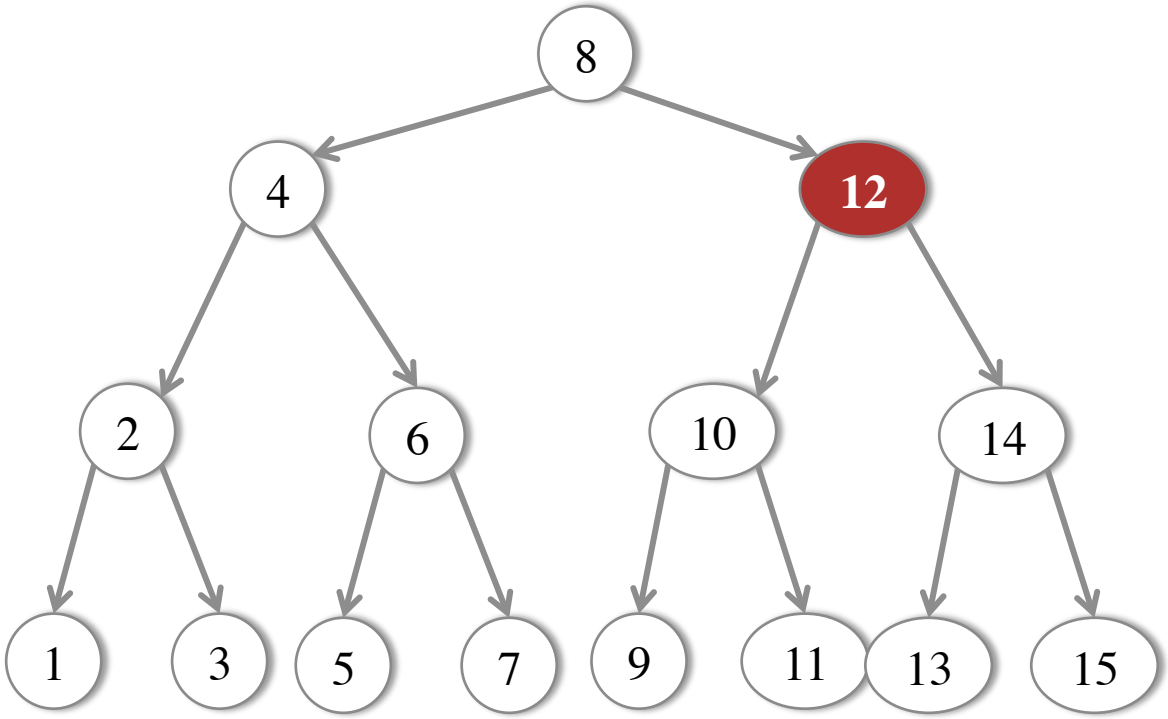
IMPLEMENTATIONS

- **Deleting nodes with 2 children**
  - *How do we delete 12?*
  - *Can we replace 12 with one of it's children?*



IMPLEMENTATIONS

- **Deleting nodes with 2 children**
  - *How do we delete 12?*
  - *Can we replace 12 with one of it's children?*
  - *Need to find candidate to replace 12*



IMPLEMENTATIONS

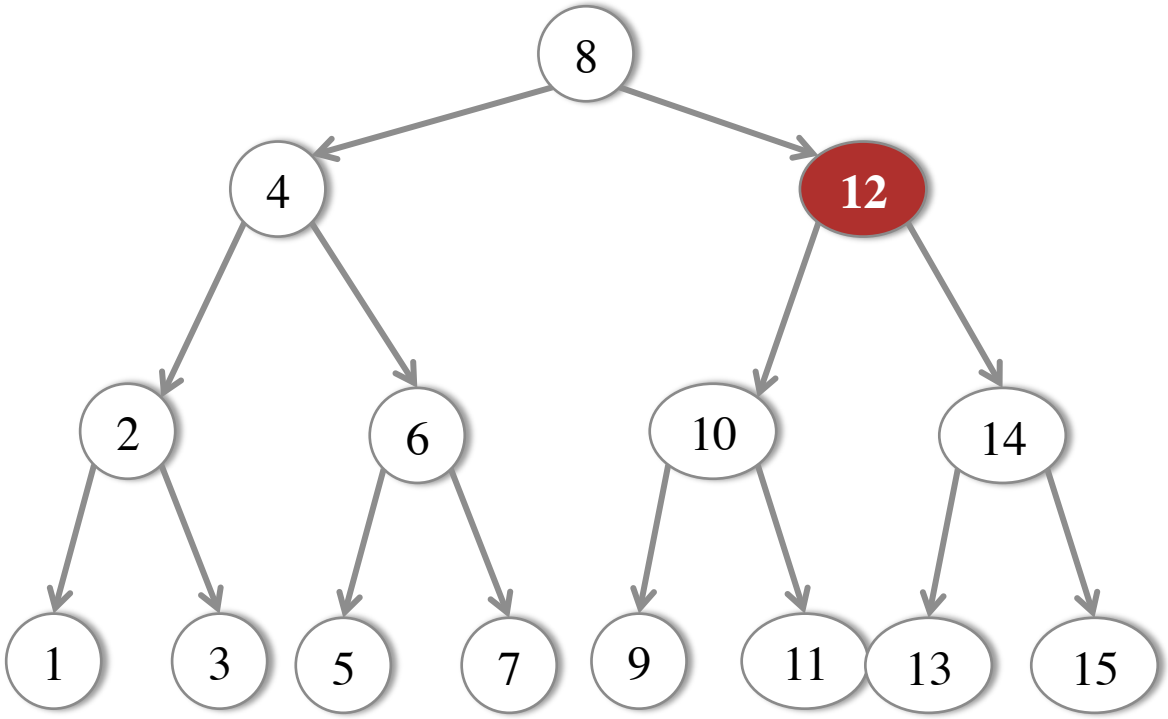
- **Deleting nodes with 2 children**
  - If a node has 2 children, then we can “delete” it by over writing the node with a different <key, value> pair

IMPLEMENTATIONS

- **Deleting nodes with 2 children**
  - If a node has 2 children, then we can “delete” it by over writing the node with a different <key, value> pair
  - In order to avoid changing the shape and doing too much work, it must be either the predecessor (the element just before it in sorted order) or the successor (the element just after it in sorted order)

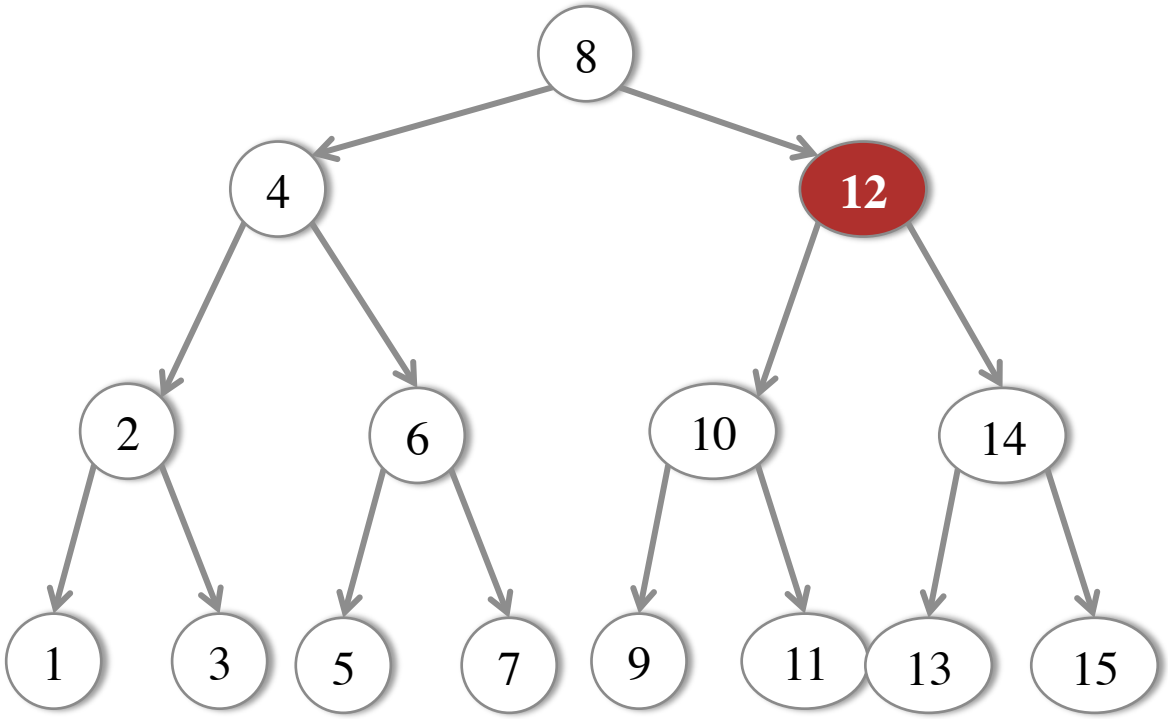
IMPLEMENTATIONS

- **Deleting nodes with 2 children**
  - *What are the predecessor and successor of 12?*



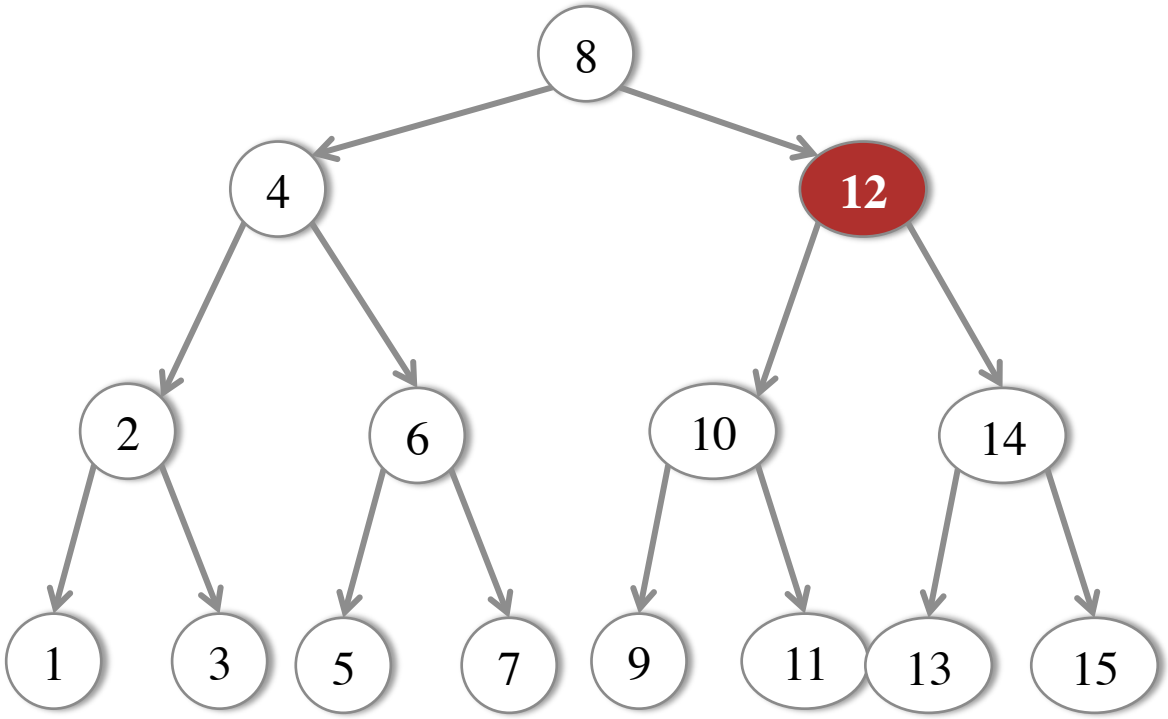
IMPLEMENTATIONS

- **Deleting nodes with 2 children**
  - *What are the predecessor and successor of 12?*
  - *What is unique about these elements?*



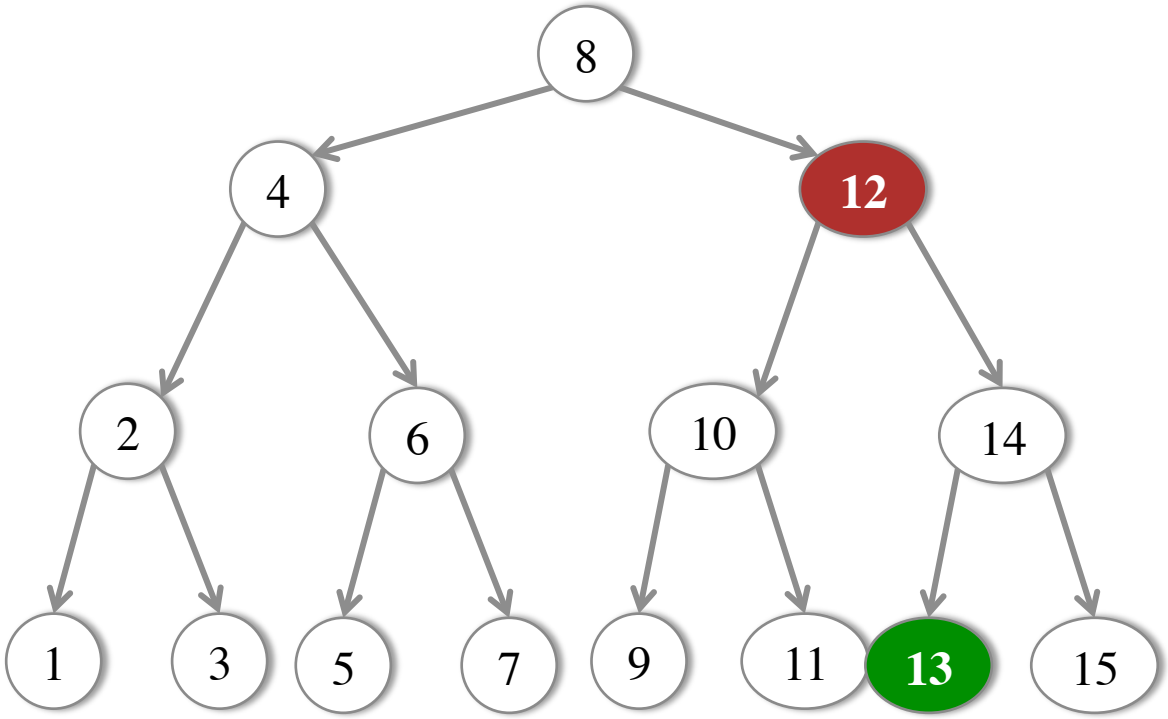
IMPLEMENTATIONS

- **Deleting nodes with 2 children**
  - *What are the predecessor and successor of 12?*
  - *What is unique about these elements?*
    - ***They have at most one child! Easy deletion***



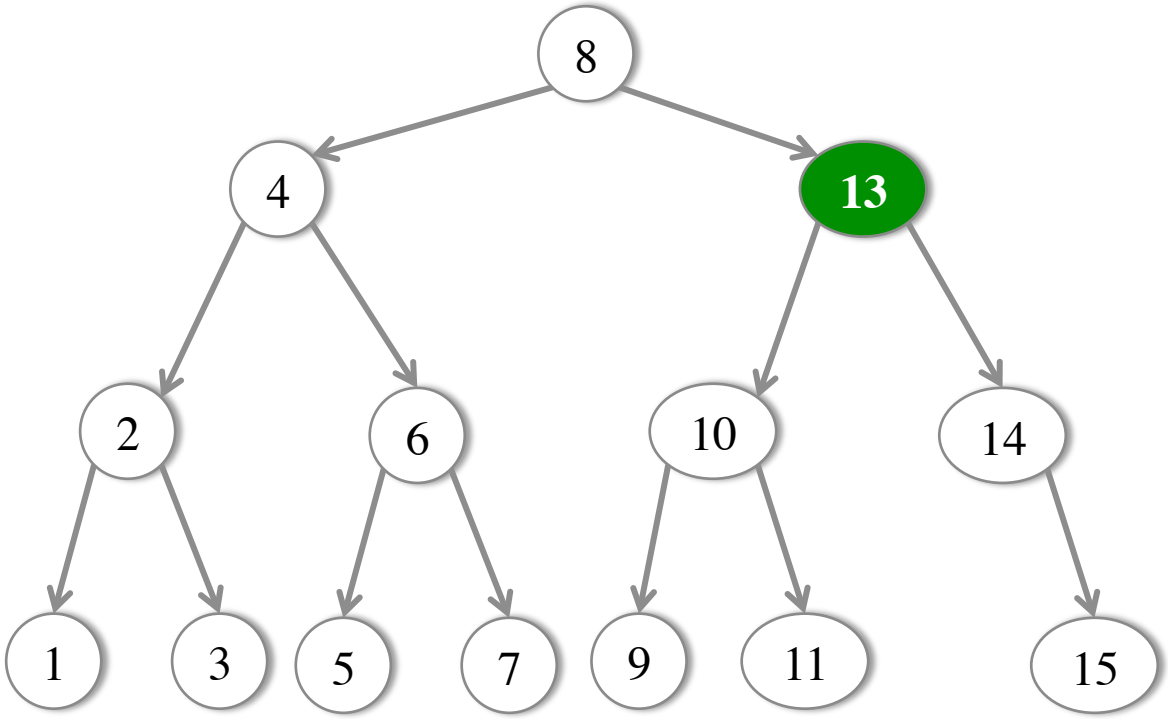
IMPLEMENTATIONS

- **Deleting nodes with 2 children**
  - *What are the predecessor and successor of 12?*
  - *What is unique about these elements?*
    - ***They have at most one child! Easy deletion***



IMPLEMENTATIONS

- **Deleting nodes with 2 children**
  - *What are the predecessor and successor of 12?*
  - *What is unique about these elements?*
    - ***They have at most one child! Easy deletion***



IMPLEMENTATIONS

- **BST Analysis:**
  - Delete():
    - Worst case(): O(n)

IMPLEMENTATIONS

- **BST Analysis:**
  - Delete():
    - Worst case(): O(n), finding the predecessor/successor takes time

IMPLEMENTATIONS

- **BST Analysis:**
  - Delete():
    - Worst case(): O(n), finding the predecessor/successor takes time. *What is this case?*

IMPLEMENTATIONS

- **BST Analysis:**
  - Delete():
    - Worst case(): O(n), finding the predecessor/successor takes time. *What is this case?*
    - Best case(): O(1) if we're deleting the root from a degenerate tree

IMPLEMENTATIONS

- **BST Analysis:**
  - Delete():
    - Worst case(): O(n), finding the predecessor/successor takes time. *What is this case?*
    - Best case(): O(1) if we're deleting the root from a degenerate tree
      - "Degenerate" trees are those that are very unbalanced.

ANALYSIS

- Height

ANALYSIS

- **Height**
  - Many of our worst cases are when trees are poorly balanced

ANALYSIS

- **Height**
  - Many of our worst cases are when trees are poorly balanced
  - Can we enforce this balance?

ANALYSIS

- **Height**
  - Many of our worst cases are when trees are poorly balanced
  - Can we enforce this balance?
  - What are some possible balance conditions?

ANALYSIS

- **Height**
  - Many of our worst cases are when trees are poorly balanced
  - Can we enforce this balance?
  - What are some possible balance conditions?
    - Number of elements on the left = number on right?

ANALYSIS

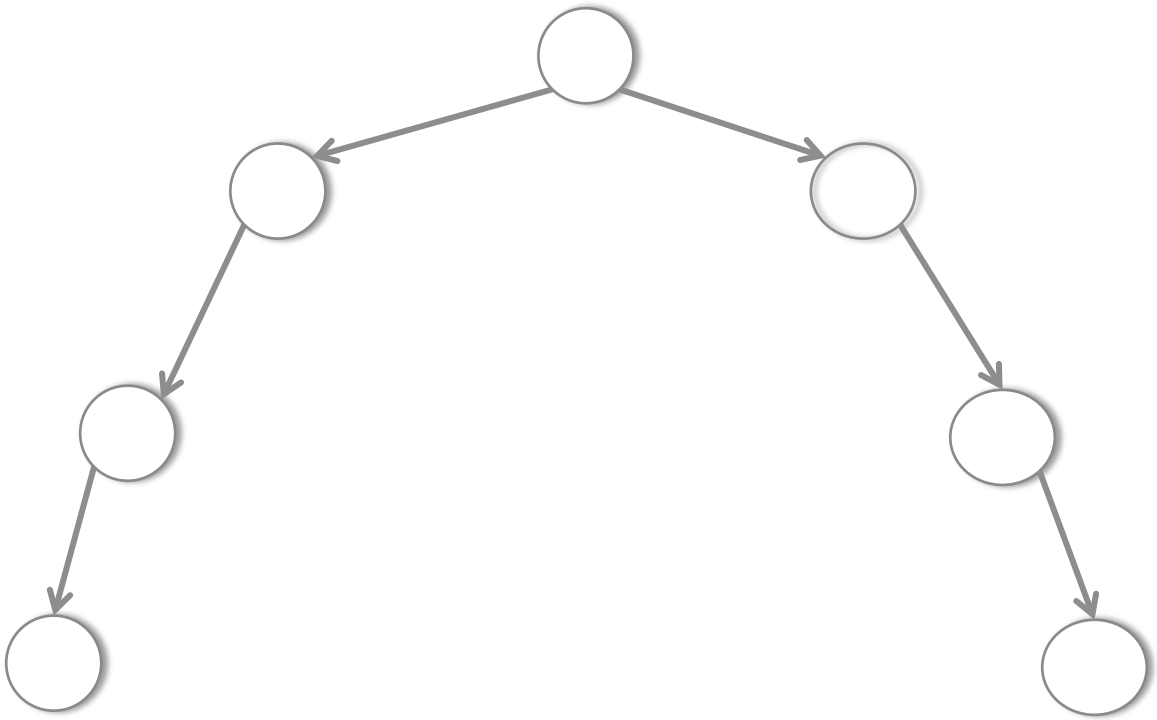
- **Height**
  - Many of our worst cases are when trees are poorly balanced
  - Can we enforce this balance?
  - What are some possible balance conditions?
    - Number of elements on the left = number on right?
    - What we really care about though is the height of the tree

ANALYSIS

- **Height**
  - Many of our worst cases are when trees are poorly balanced
  - Can we enforce this balance?
  - What are some possible balance conditions?
    - Number of elements on the left = number on right?
    - What we really care about though is the height of the tree
    - Height of the left = height on the right?

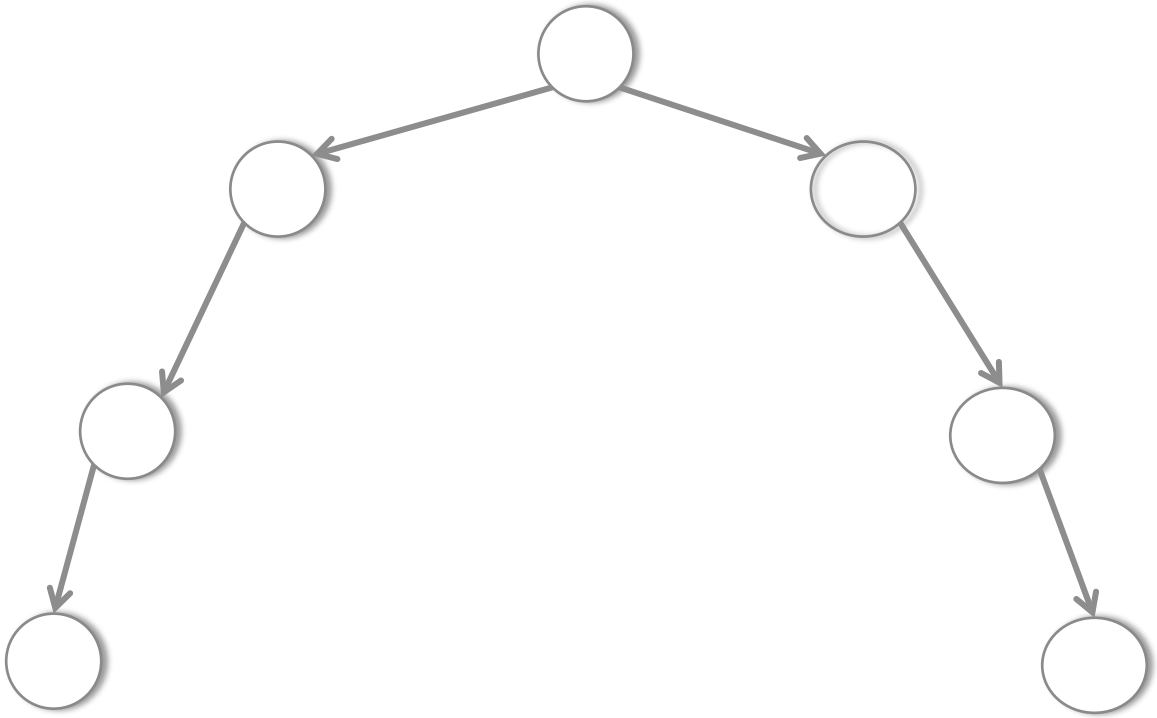
ANALYSIS

- **This doesn’t help much**



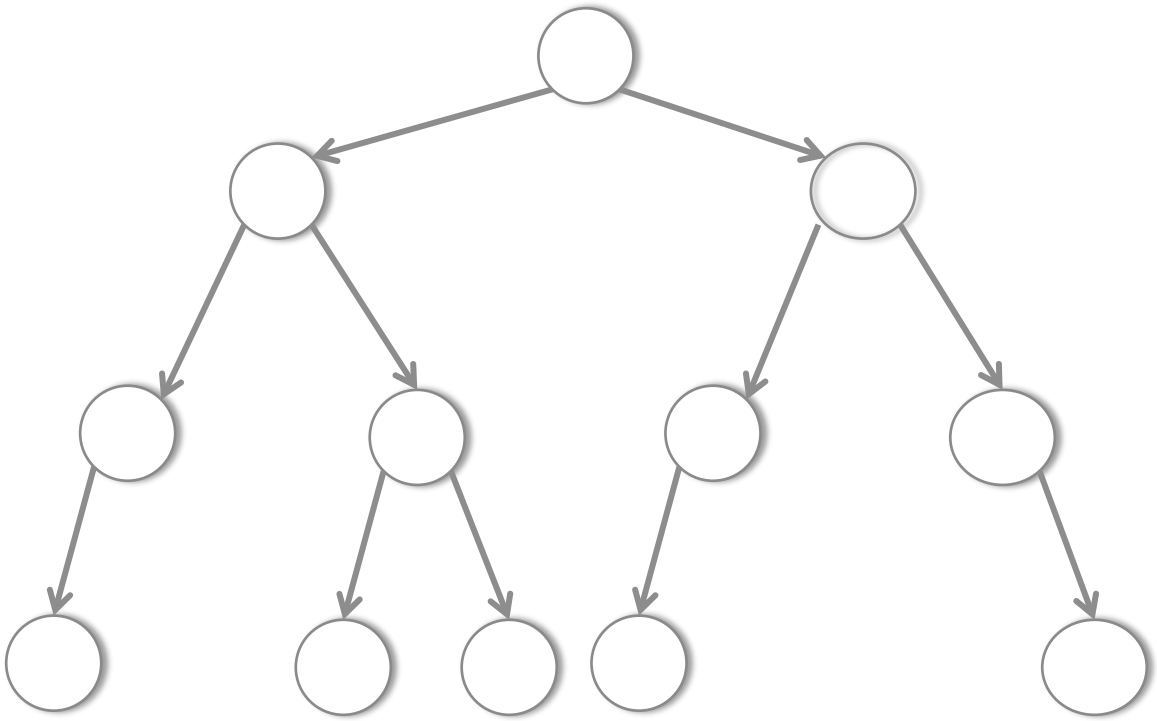
ANALYSIS

- **This doesn't help much**
  - Need the definition to be recursive
  - Let height(left) = height(right) for all nodes



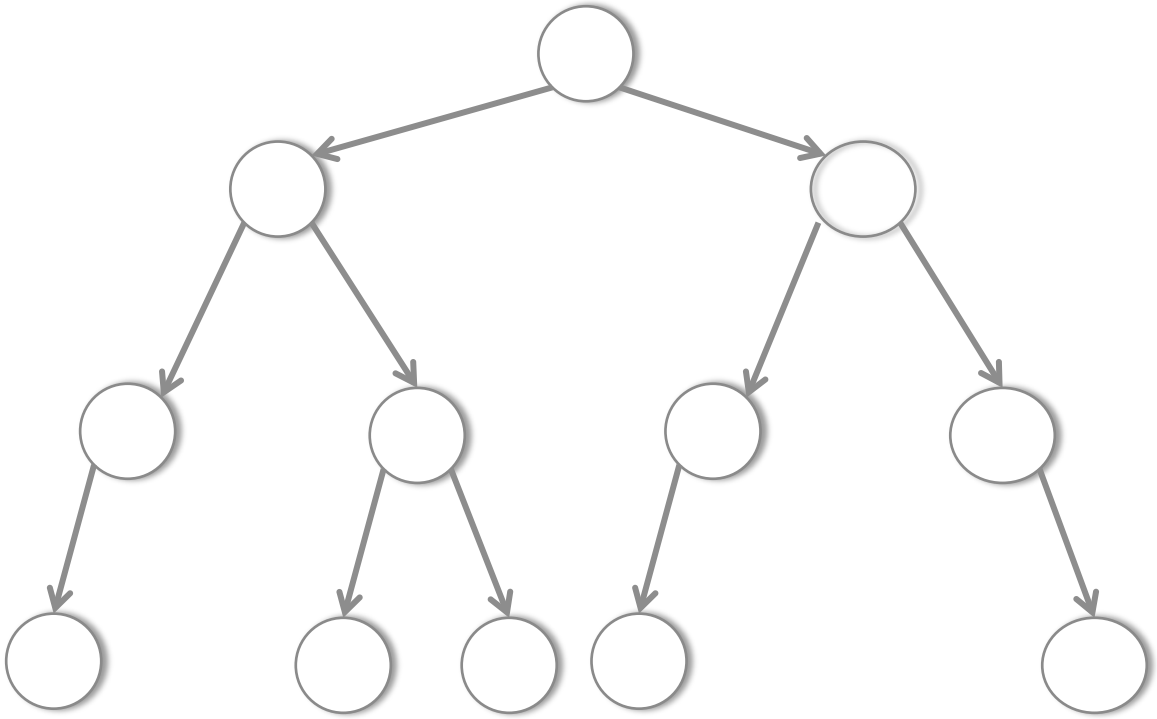
ANALYSIS

- **Now what's wrong?**



ANALYSIS

- **Now what's wrong?**
  - Only perfect trees (with 2^k children) can exist



ANALYSIS

- For each node in the tree, the height of its left and right subtrees can differ by at most one

ANALYSIS

- For each node in the tree, the height of its left and right subtrees can differ by at most one
- $|(\text{height(left)} - \text{height(right)}| \leq 1$

ANALYSIS

- For each node in the tree, the height of its left and right subtrees can differ by at most one
- $|(height(left) - height(right)| \leq 1$
- This is the AVL property, and we can use it to create self balancing trees