CSE 332

JULY 3RD – DICTIONARY ADT

TODAY'S SCHEDULE

- Dictionary ADT
- Binary Search Trees
- Height, Balance and the AVL property

DICTIONARY ADT

- New abstract data type

DICTIONARY ADT

- **New abstract data type**
  - Dictionary (aka Map)
  - Data – Keys and Values

DICTIONARY ADT

- **New abstract data type**
  - Dictionary (aka Map)
  - Data – Keys and Values
    - Keys: must be comparable, used for lookup

DICTIONARY ADT

- **New abstract data type**
  - Dictionary (aka Map)
  - Data – Keys and Values
    - Keys: must be comparable, used for lookup
    - Values: the actual data itself

DICTIONARY ADT

- **New abstract data type**
  - Dictionary (aka Map)
  - Data – Keys and Values
    - Keys: must be comparable, used for lookup
    - Values: the actual data itself
  - Example (Store inventory):

DICTIONARY ADT

- **New abstract data type**
  - Dictionary (aka Map)
  - Data – Key and Value pairs
    - Keys: must be comparable, used for lookup
    - Values: the actual data itself
  - Example (Store inventory):
    - Keys: IDs (barcodes)
    - Values: Product information

DICTIONARY ADT

- Operations

DICTIONARY ADT

- **Operations**
  - `insert(key, value):` inserts the key, value pair into the dictionary. Overwrites the value if the key is already in the dictionary

DICTIONARY ADT

- **Operations**
  - `insert(key, value):` inserts the key, value pair into the dictionary. Overwrites the value if the key is already in the dictionary
  - `find(key)`: returns the stored value for a particular key in the dictionary, returns null if not found.

DICTIONARY ADT

- **Operations**
  - `insert(key, value):` inserts the key, value pair into the dictionary. Overwrites the value if the key is already in the dictionary
  - `find(key)`: returns the stored value for a particular key in the dictionary, returns null if not found.
  - `delete(key)`: removes the key and its corresponding value from the dictionary.

SET ADT

- Slightly different from Dictionary

SET ADT

- Slightly different from Dictionary
- No values, the set only cares if a key is present or not

SET ADT

- Slightly different from Dictionary
- No values, the set only cares if a key is present or not
- Find, insert and delete have few differences

SET ADT

- Slightly different from Dictionary
- No values, the set only cares if a key is present or not
- Find, insert and delete have few differences
- Possible to implement other functions from sets

SET ADT

- **Slightly different from Dictionary**
- **No values, the set only cares if a key is present or not**
- **Find, insert and delete have few differences**
- **Possible to implement other functions from sets**
    - Union, intersection, difference

APPLICATIONS

- **Store information in key, value pairs**
  - Very common usage pattern

APPLICATIONS

- **Store information in key, value pairs**
  - Very common usage pattern
    - Phone directories
    - Indexing
    - OS page tables
    - Databases

IMPLEMENTATIONS

- Important to allow fast operations over the keys

IMPLEMENTATIONS

- **Important to allow fast operations over the keys**
  - Dependent on what the client uses most

IMPLEMENTATIONS

- **Important to allow fast operations over the keys**
  - Dependent on what the client uses most
  - Could be many lookups and few inserts

IMPLEMENTATIONS

- **Important to allow fast operations over the keys**
  - Dependent on what the client uses most
  - Could be many lookups and few inserts
- **Keys and Values should be stored together in some way**

IMPLEMENTATIONS

- **Important to allow fast operations over the keys**
  - Dependent on what the client uses most
  - Could be many lookups and few inserts
- **Keys and Values should be stored together in some way**
  - Both objects in one node
  - Paired arrays (one stores keys and the other values)

IMPLEMENTATIONS

- Simple implementations

IMPLEMENTATIONS

- **Simple implementations**

	insert	find	delete
Unsorted linked-list			
Unsorted array			
Sorted linked list			
Sorted array			

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**
  - **For now, assume 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()?

IMPLEMENTATIONS

- **BST Analysis:**
  - What is our time for the three functions?
    - Insert()? Delete()? Find()?
    - *Take 5 minutes to discuss*

IMPLEMENTATIONS

- **BST Analysis:**
  - What is our time for the three functions?
    - Insert()? Delete()? Find()?
    - *Take 5 minutes to discuss*
    - *Consider best and worst-case.*

IMPLEMENTATIONS

- **BST Analysis:**
  - What is our time for the three functions?
    - Insert()? Delete()? Find()?
    - *Take 5 minutes to discuss*
    - *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(log n)

IMPLEMENTATIONS

- **BST Analysis:**
  - Insert():
    - Worst case: O(n)
    - Best case: O(log n)
    - What is the general case here?

IMPLEMENTATIONS

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

IMPLEMENTATIONS

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

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

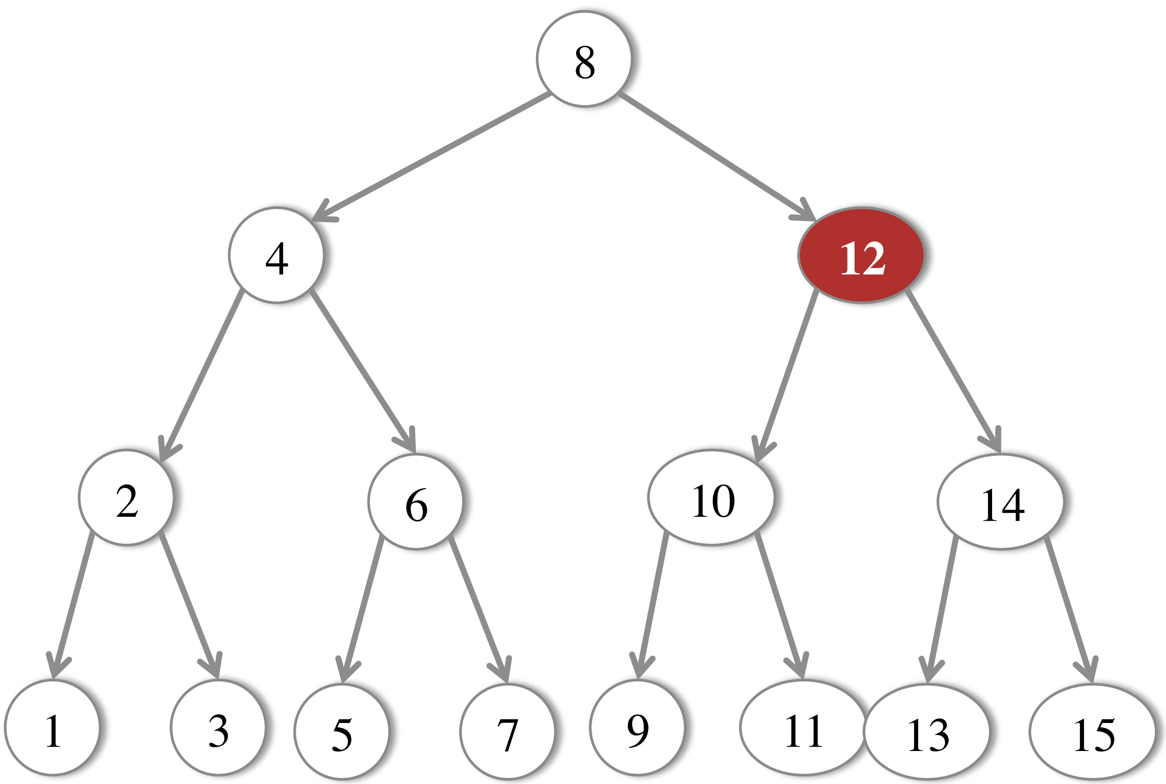
- **BST Analysis:**
  - Delete():
    - What are some strategies for deleting?
    - Are there any cases where deleting is easy?
    - Case 0: The element is not in the data structure
      - Don’t change the data, possibly throw an exception
    - Case 1: The key is a leaf in the tree
      - Remove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

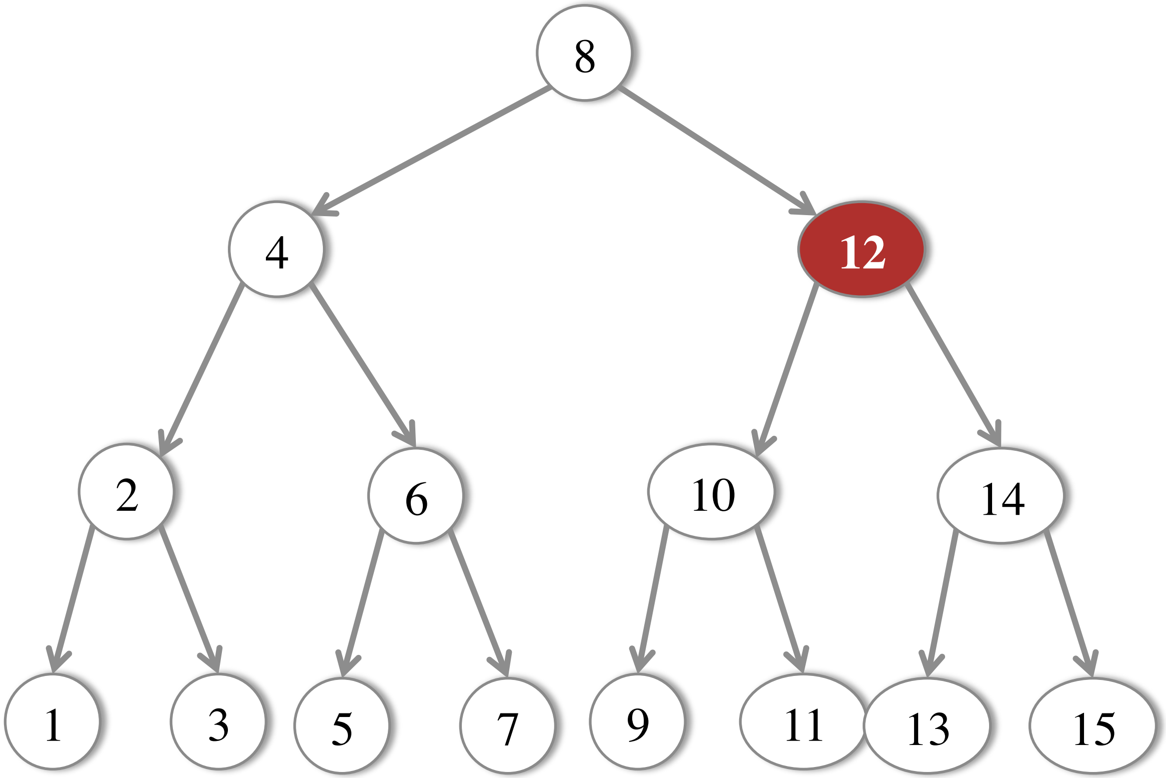
IMPLEMENTATIONS

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



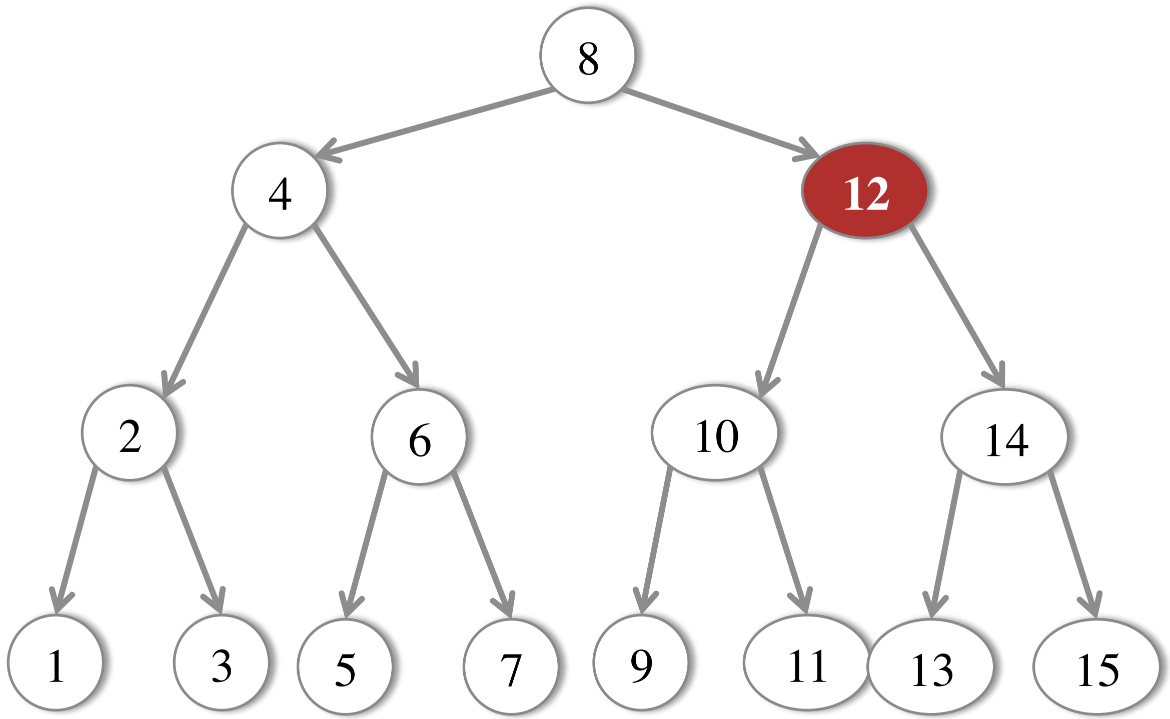
IMPLEMENTATIONS

- **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

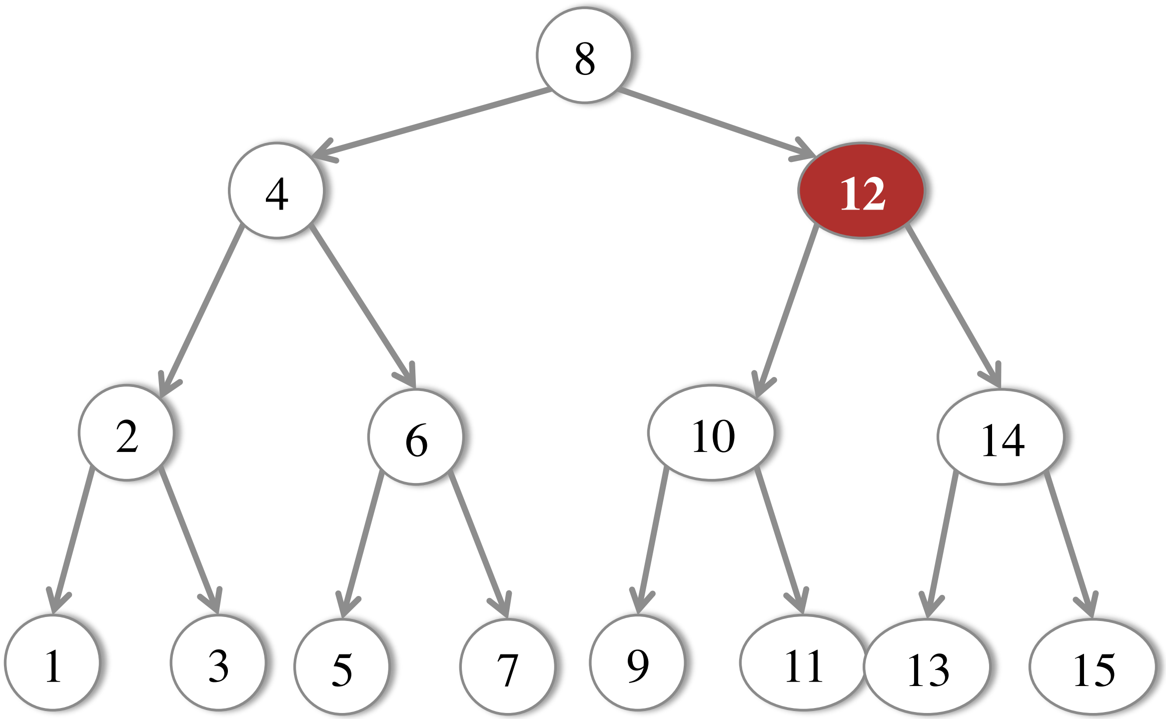
- **Deleting nodes with 2 children**
  - If a node has 2 children, then we can “delete” it by over writing the node with a different <key, value> pair

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)

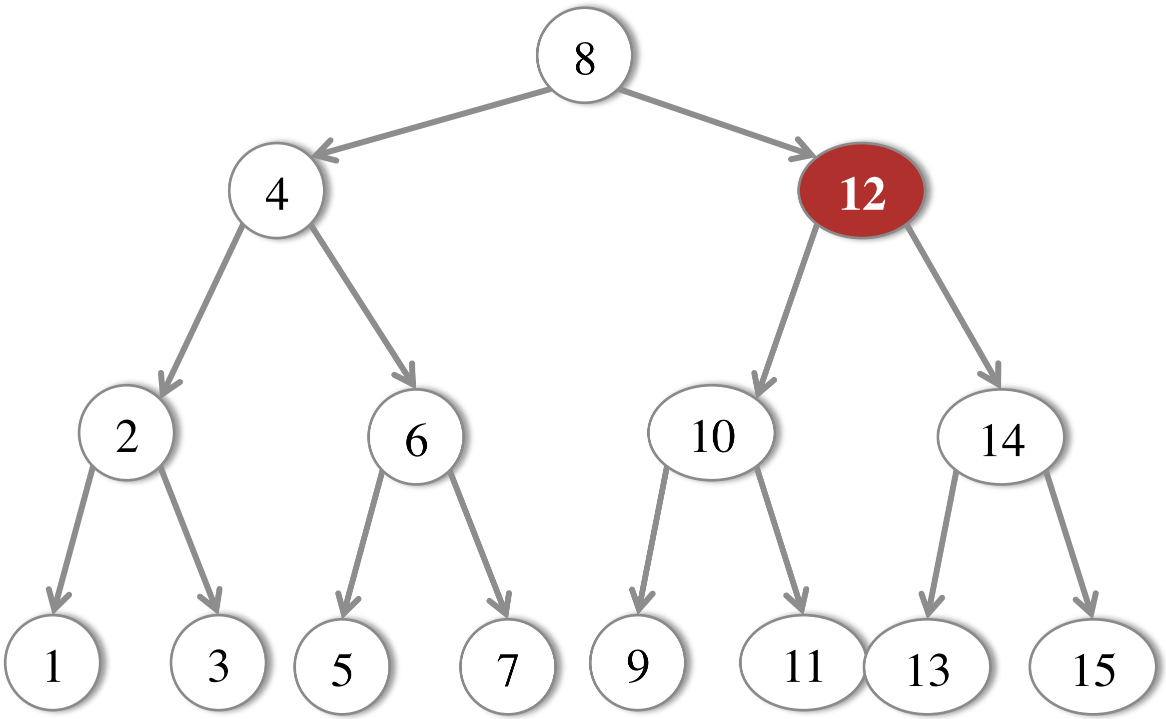
IMPLEMENTATIONS

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



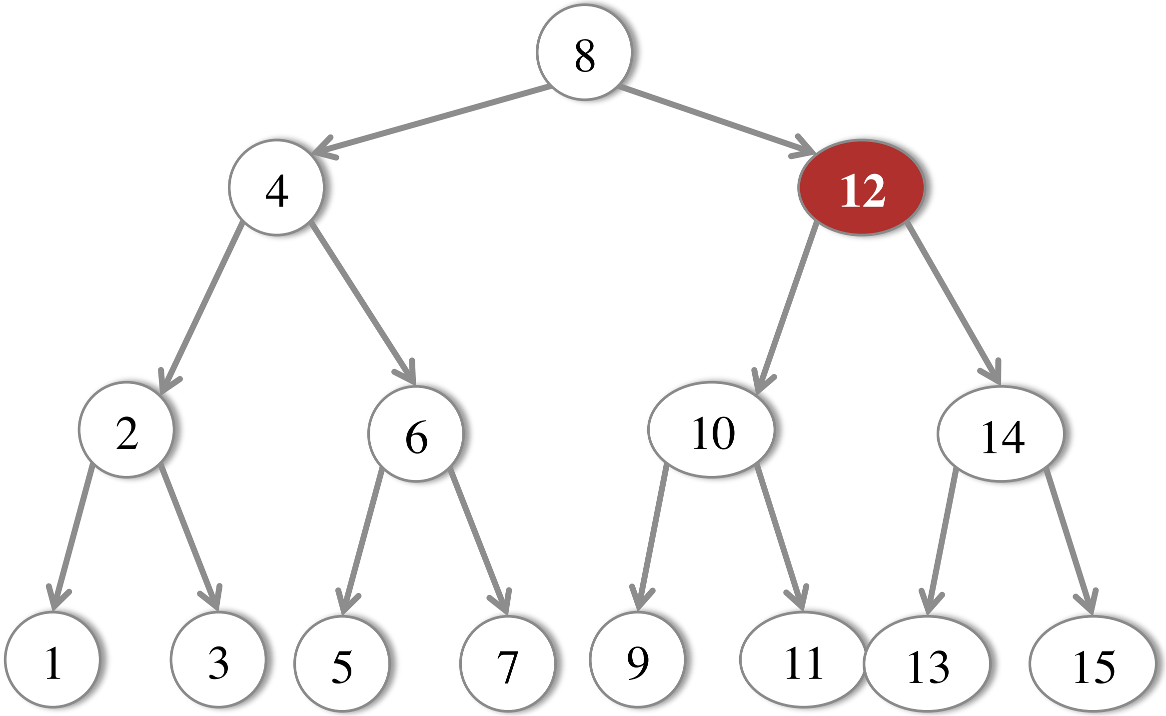
IMPLEMENTATIONS

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



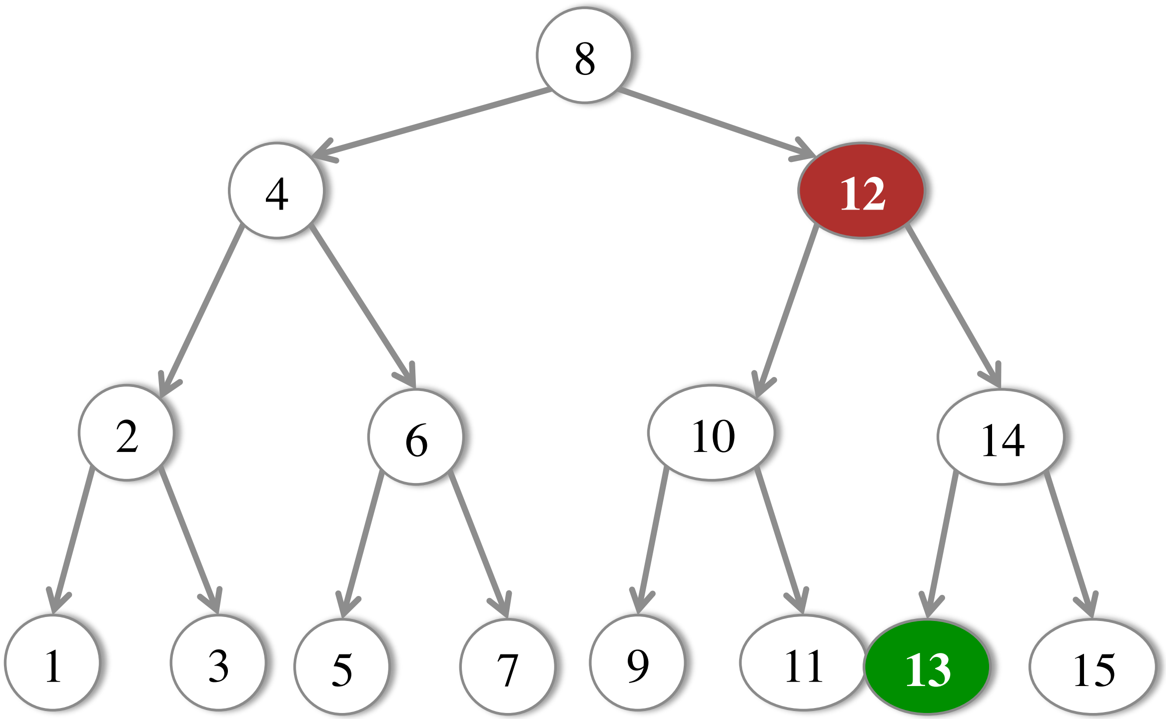
IMPLEMENTATIONS

- **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***



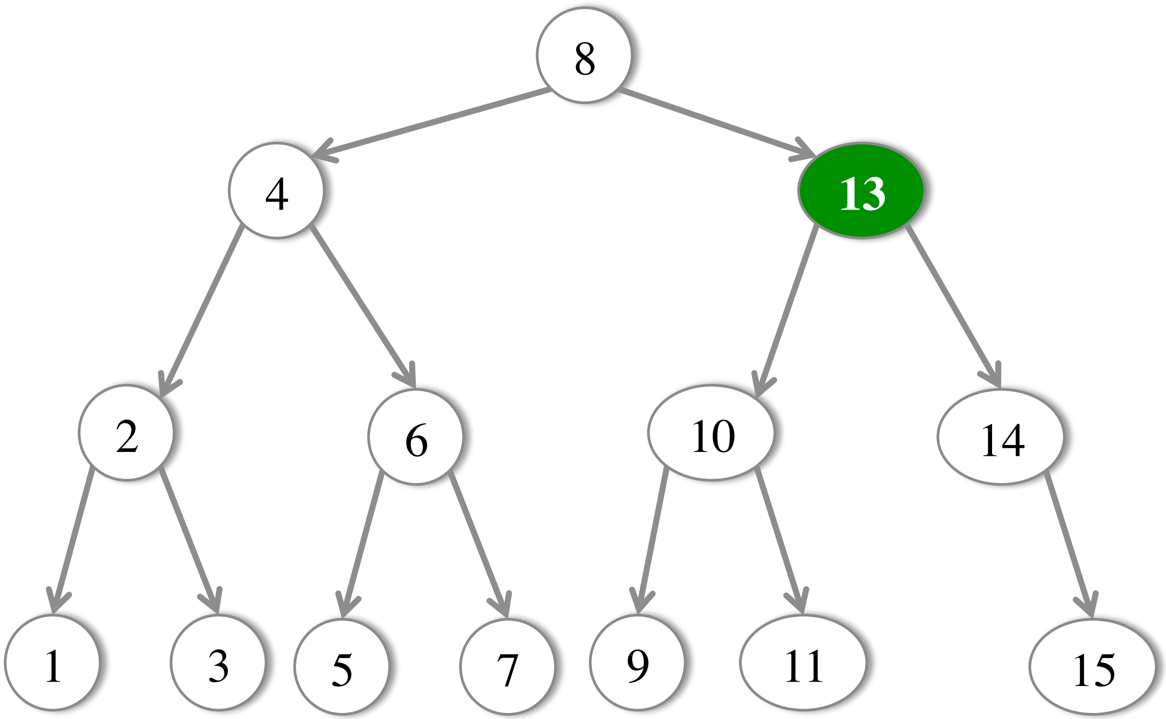
IMPLEMENTATIONS

- **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

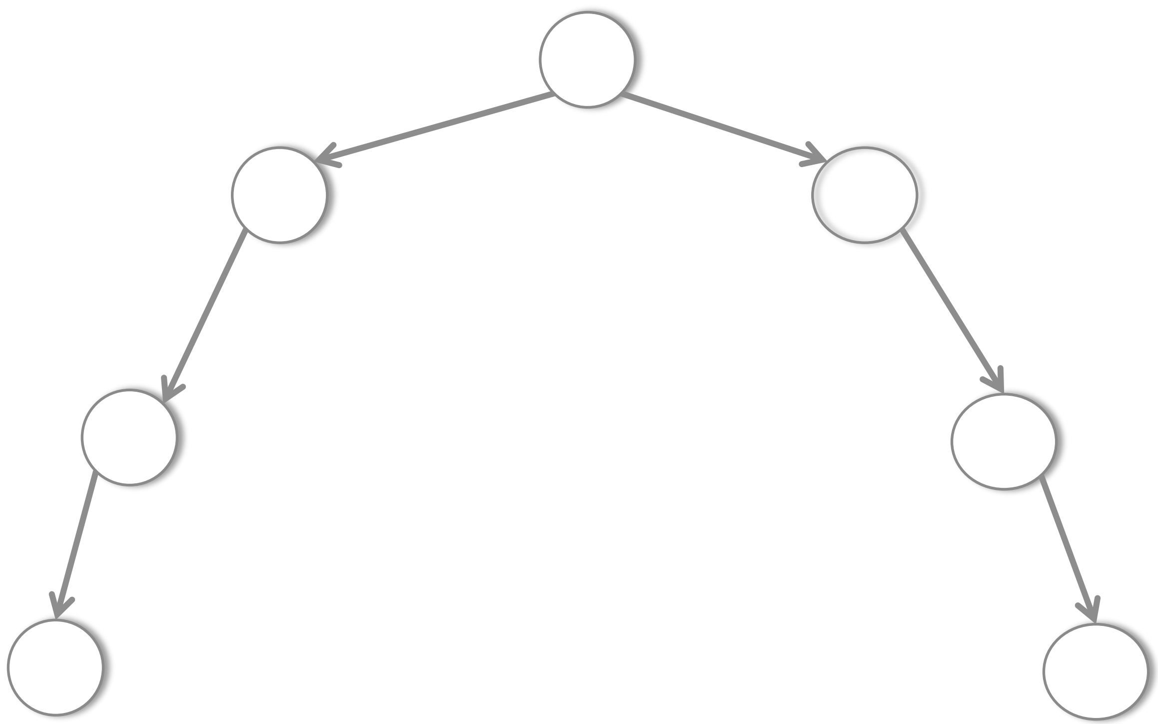
- **Height**
  - Many of our worst cases are when trees are poorly balanced
  - Can we enforce this balance?
  - What are some possible balance conditions?
    - Number of elements on the left = number on right?
    - What we really care about though is the height of the tree

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?

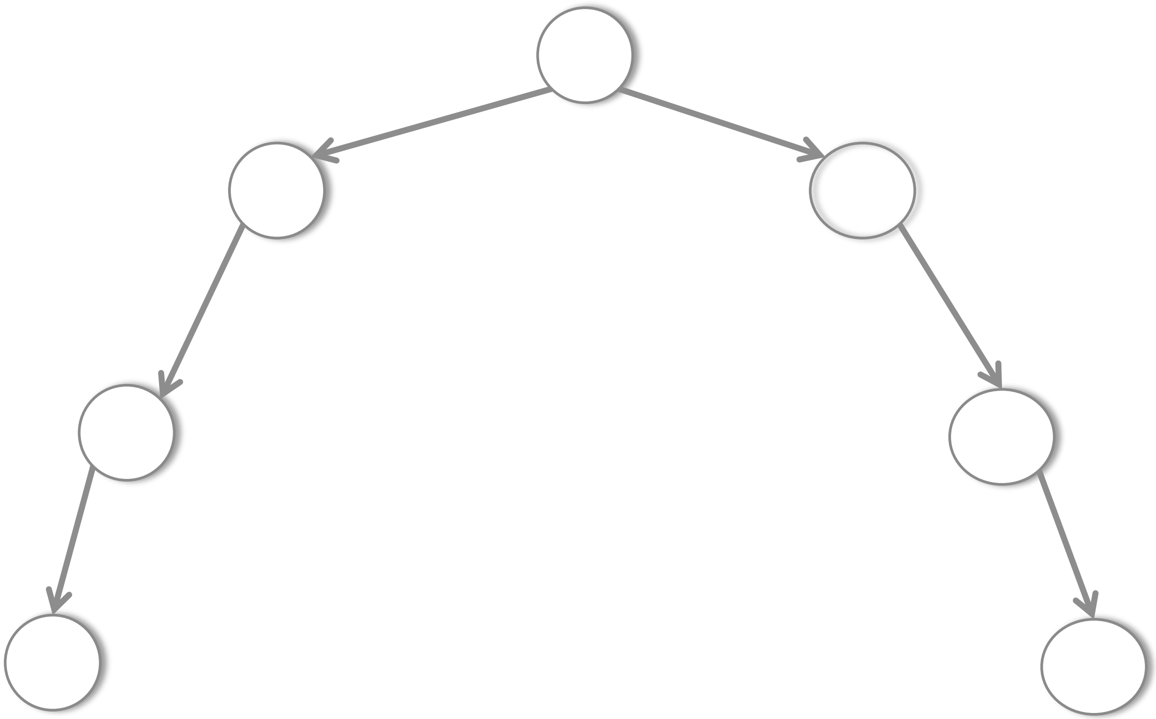
ANALYSIS

- This doesn't help much



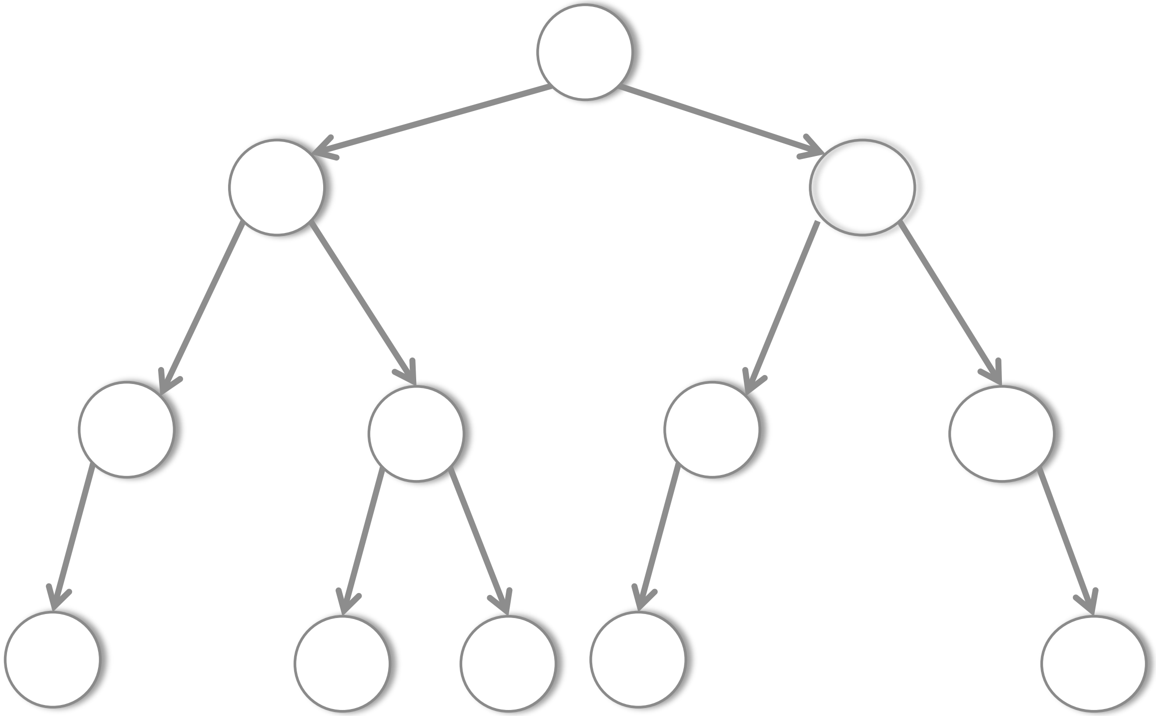
ANALYSIS

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



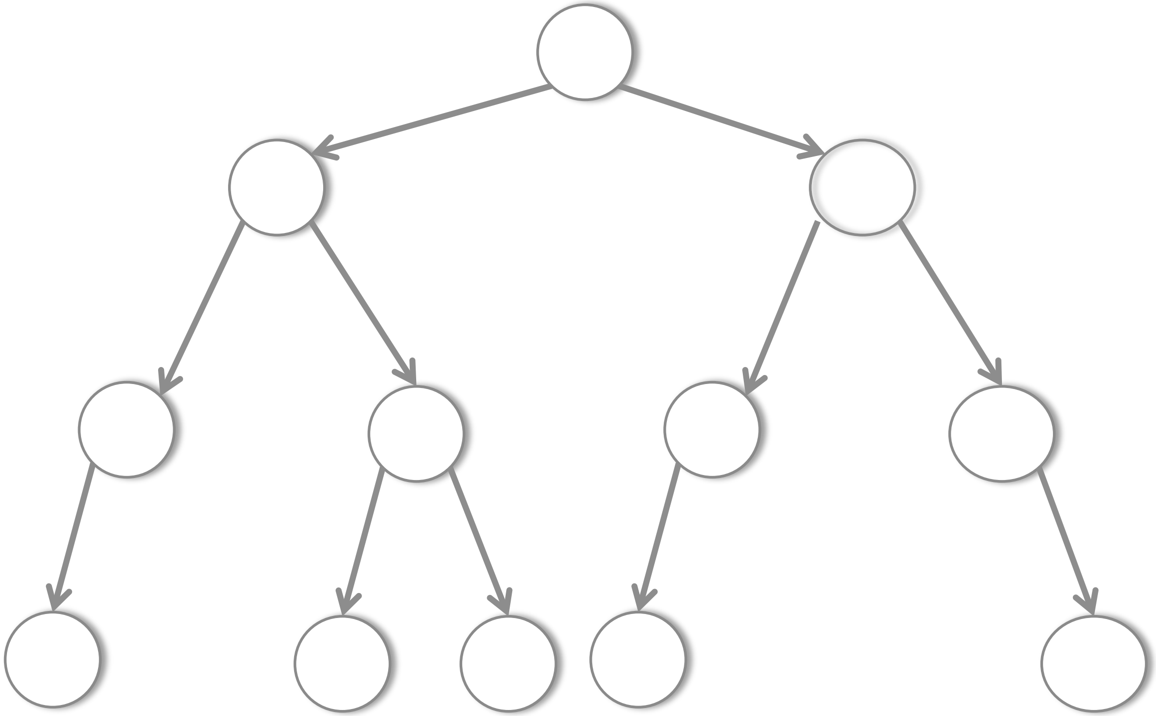
ANALYSIS

- 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
- $|(\text{height}(\text{left}) - \text{height}(\text{right})| \leq 1$

ANALYSIS

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

NEXT CLASS

- AVL Trees and implementation