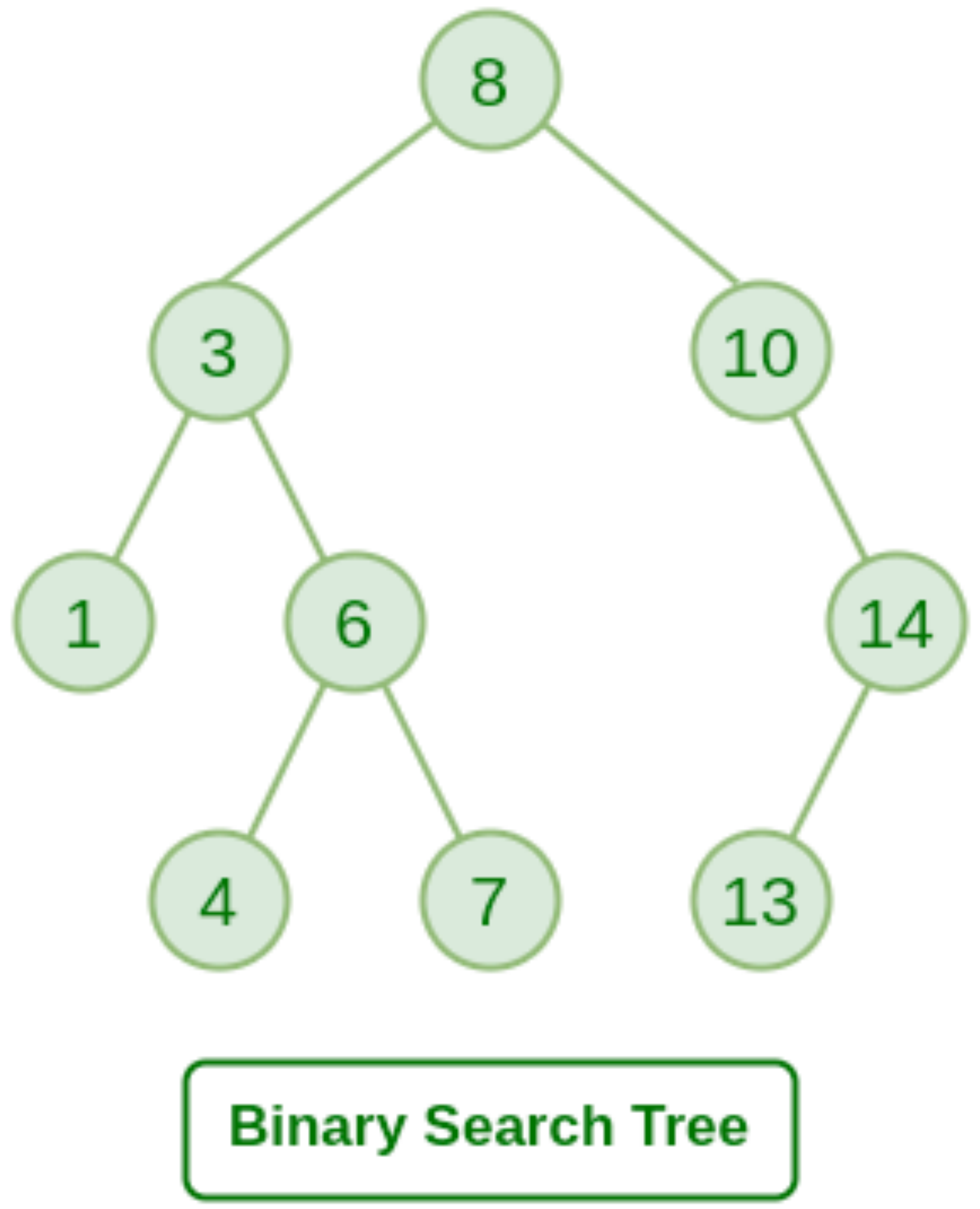
CS62 Class 17: Binary Search Trees & Maps

Searching



BST: For each node, its left child is smaller, and its right child is bigger

Agenda

- Maps/Dictionaries
- Binary Search Trees
    - Derivation/motivation
    - Definition
    - Searching
    - Insert
    - Hibbard deletion

Maps (Dictionaries)

Dictionaries

- Dictionaries (Python) are known as Maps (Java).
- Also known as: symbol tables, maps, indices, associative arrays.
- Key-value pair abstractions that support two operations:
  - Insert a key-value pair.
  - Given a key, search for the corresponding value.

Map Example

Maps are very handy tools for all sorts of tasks. Example: Counting words.

```java
Map<String, Integer> m = new TreeMap<>();
String[] text = {"sumomo", "mo", "momo",
"mo", "momo", "no", "uchi"};
for (String s : text) {
    int currentCount = m.getOrDefault(s,0);
    m.put(s, currentCount + 1);
}
```

sumomo	1
mo	2
momo	2
no	1
uchi	1

```python
m = {}
text = ["sumomo", "mo", "momo", “mo", "momo", "no", "uchi"]
for s in text:
    if s in m.keys():
        m[s] += 1
    else:
        m[s] = 1
```

Python equivalent

Basic dictionary API

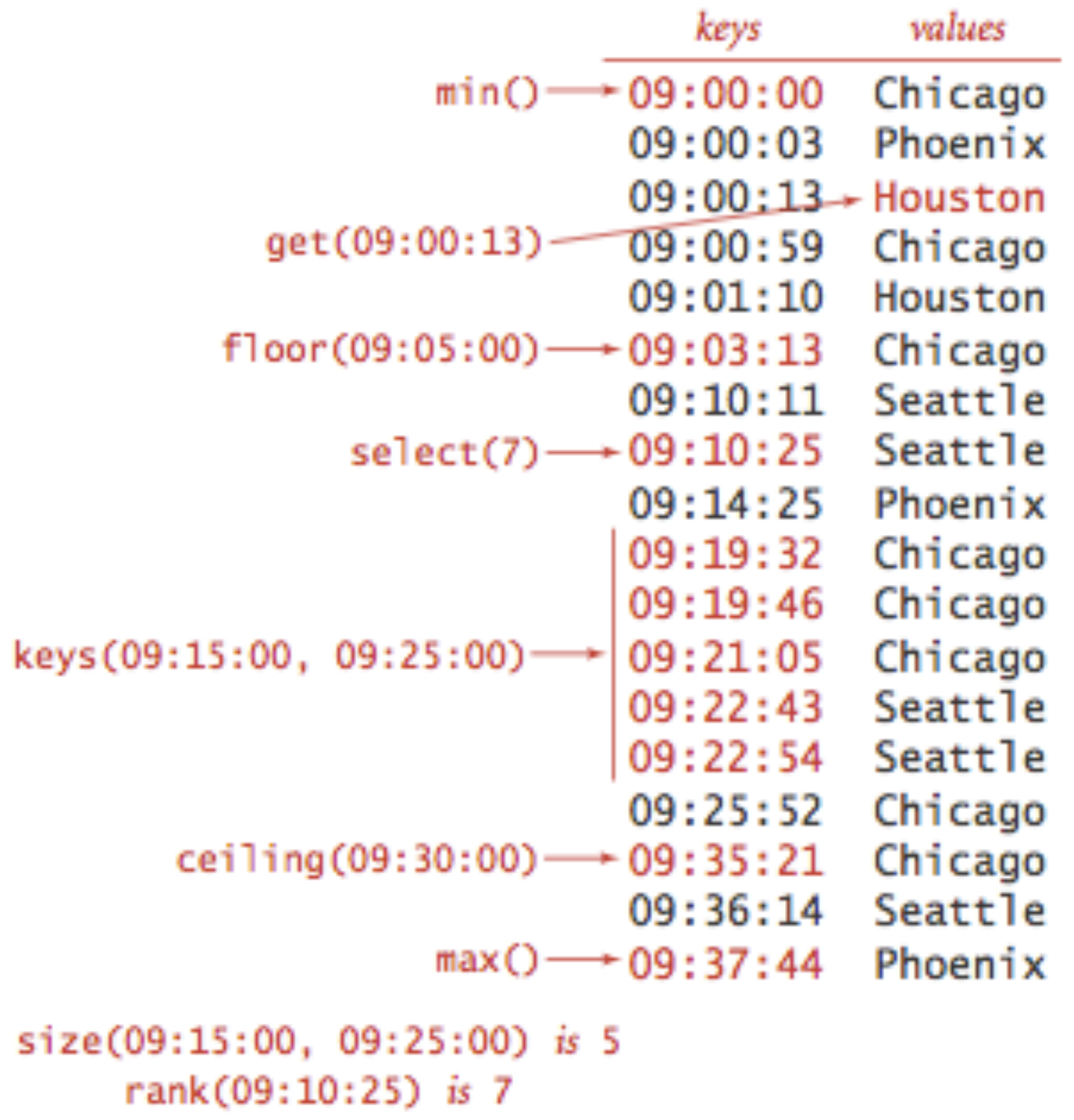
- `public class Dictionary <Key extends Comparable<Key>, Value>`
- Dictionary(): create an empty dictionary. By convention, values are **not** `null`.
- `void put(Key key, Value val)`: insert key-value pair.
    - Overwrites old value with new value if key already exists.
- `Value get(Key key)`: return value associated with key.
    - Returns `null` if key not present. (That’s why values can’t be null.)
- `boolean contains(Key key)`: is there a value associated with key?
- `Iterable keys()`: all the keys in the dictionary.
- `void delete(Key key)`: delete key and associated value.
- `boolean isEmpty()`: is the dictionary empty?
- `int size()`: number of key-value pairs.

floor(x): largest key < x

ceiling(x): smallest key > x

Ordered dictionaries

- Data structure: Maintain parallel arrays for keys and values, sorted by keys.
- Search: Use binary search to find key.
  - At most $O(\log n)$ compares to search a sorted array of length n.
- Insert: Use binary search to find key. If it does not exist, shift all larger keys over.
  - At most $O(n)$ time.
- Note: Remember that in Python, dictionaries were inherently unordered (we used lists if we wanted ordered data structures). Now we can build our own ordered ones!



keys	values
09:00:00	Chicago
09:00:03	Phoenix
09:00:13	Houston
09:00:59	Chicago
09:01:10	Houston
09:03:13	Chicago
09:10:11	Seattle
09:10:25	Seattle
09:14:25	Phoenix
09:19:32	Chicago
09:19:46	Chicago
09:21:05	Chicago
09:22:43	Seattle
09:22:54	Seattle
09:25:52	Chicago
09:35:21	Chicago
09:36:14	Seattle
09:37:44	Phoenix

rank(x): # of keys < x

Ordered dictionary API

- `Key min()`: smallest key.
- `Key max()`: largest key.
- `Key floor(Key key)`: largest key less than or equal to given key.
- `Key ceiling(Key key)`: smallest key greater than or equal to given key.
- `int rank(Key key)`: number of keys less that given key.
- `Key select(int k)`: key with rank `k`.
- `Iterable keys()`: all keys in dictionary in sorted order.
- `Iterable keys(int lo, int hi)`: keys in `[lo, …, hi]` in sorted order.

Binary Search Trees: Motivation

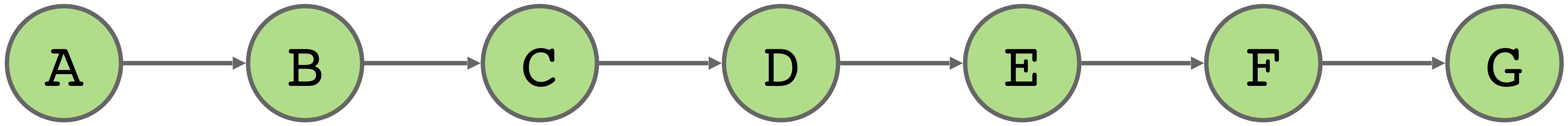
How can we efficiently implement a map?

- Searching is another fundamental problem of computer science: how can we find things quickly and efficiently?
- Our maps/dictionaries should support very fast search operations for key retrieval.

How can data structures support fast searching?

Consider the humble singly linked list (or even ArrayList!)

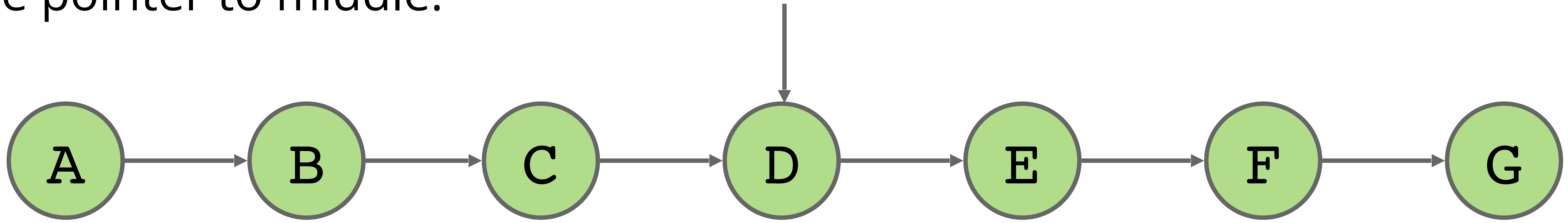
This is horrible for fast searching, because we need to iterate through the whole list: O(n) time.



Optimization: Change the Entry Point

Fundamental Problem: Slow search, even though it's in order.

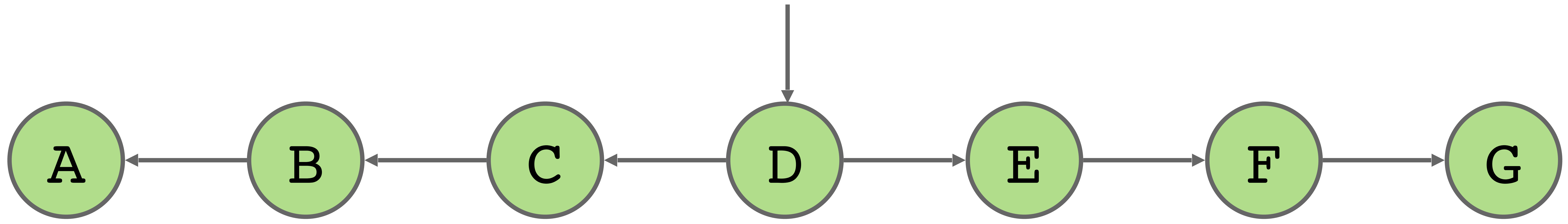
- Move pointer to middle.



Optimization: Change the Entry Point, Flip Links

Fundamental Problem: Slow search, even though it's in order.

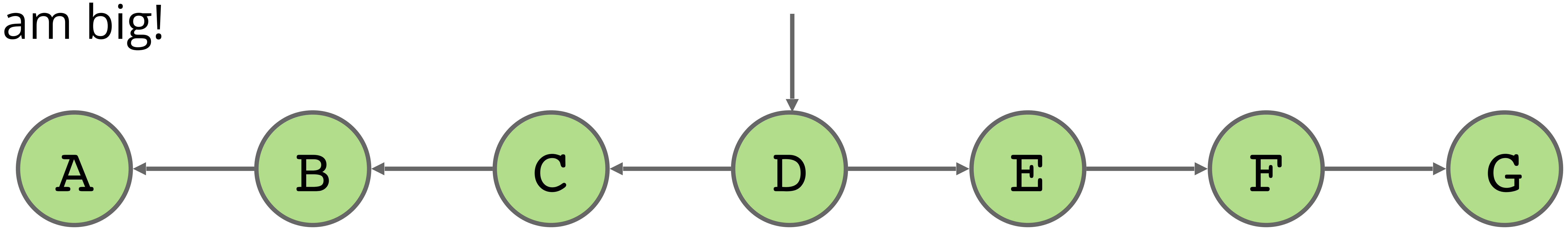
- Move pointer to middle and flip left links. Halved search time!



Optimization: Change the Entry Point, Flip Links

Fundamental Problem: Slow search, even though it's in order.

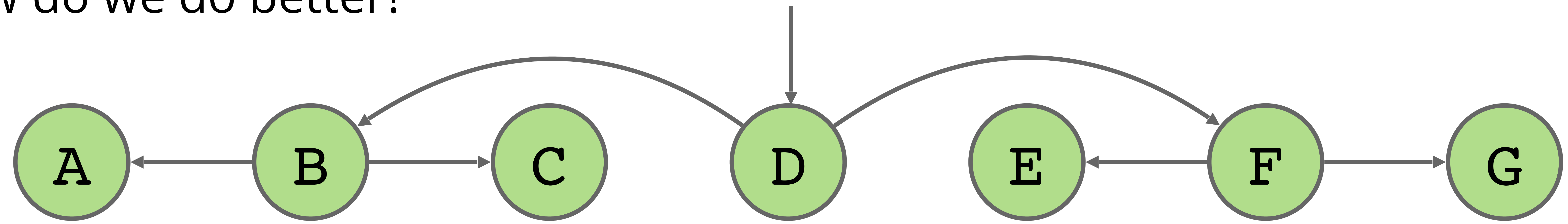
- How do we do even better?
- Dream big!

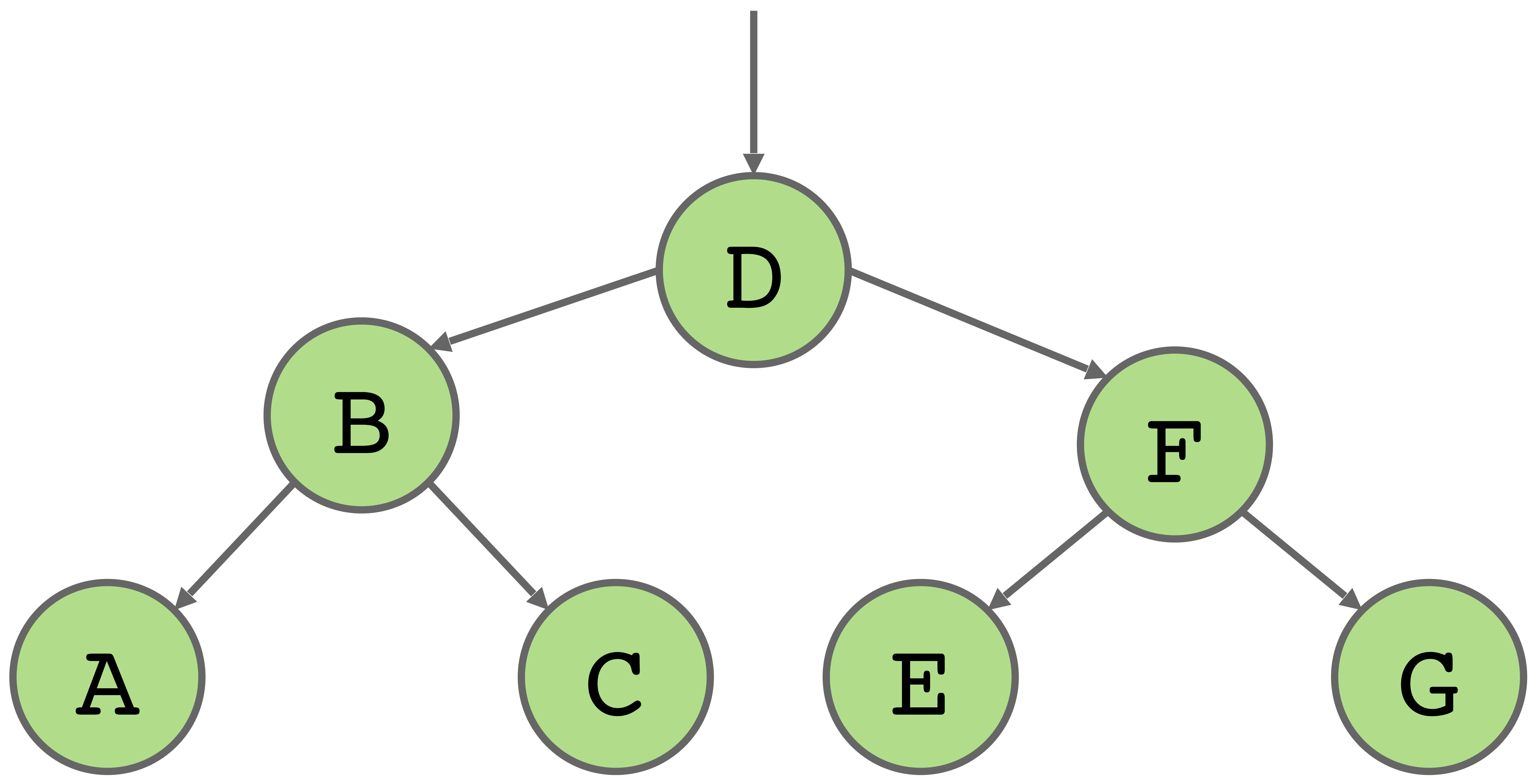


Optimization: Change Entry Point, Flip Links, Allow Big Jumps

Fundamental Problem: Slow search, even though it's in order.

- How do we do better?





Binary Search Trees: Definition

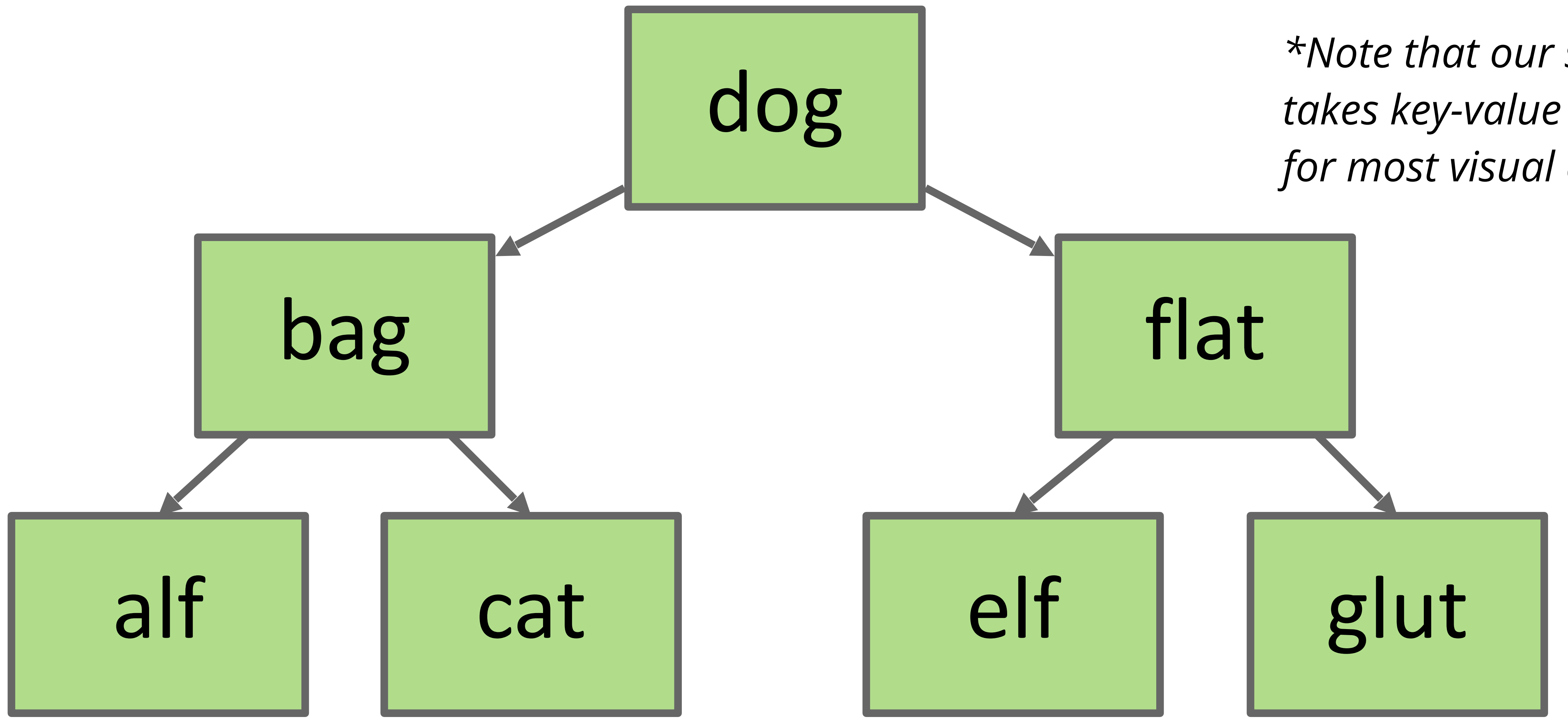
Binary Search Trees

A binary search tree is a rooted binary tree that is symmetrically ordered.

Symmetric order property of BSTs. For every node X in the tree:

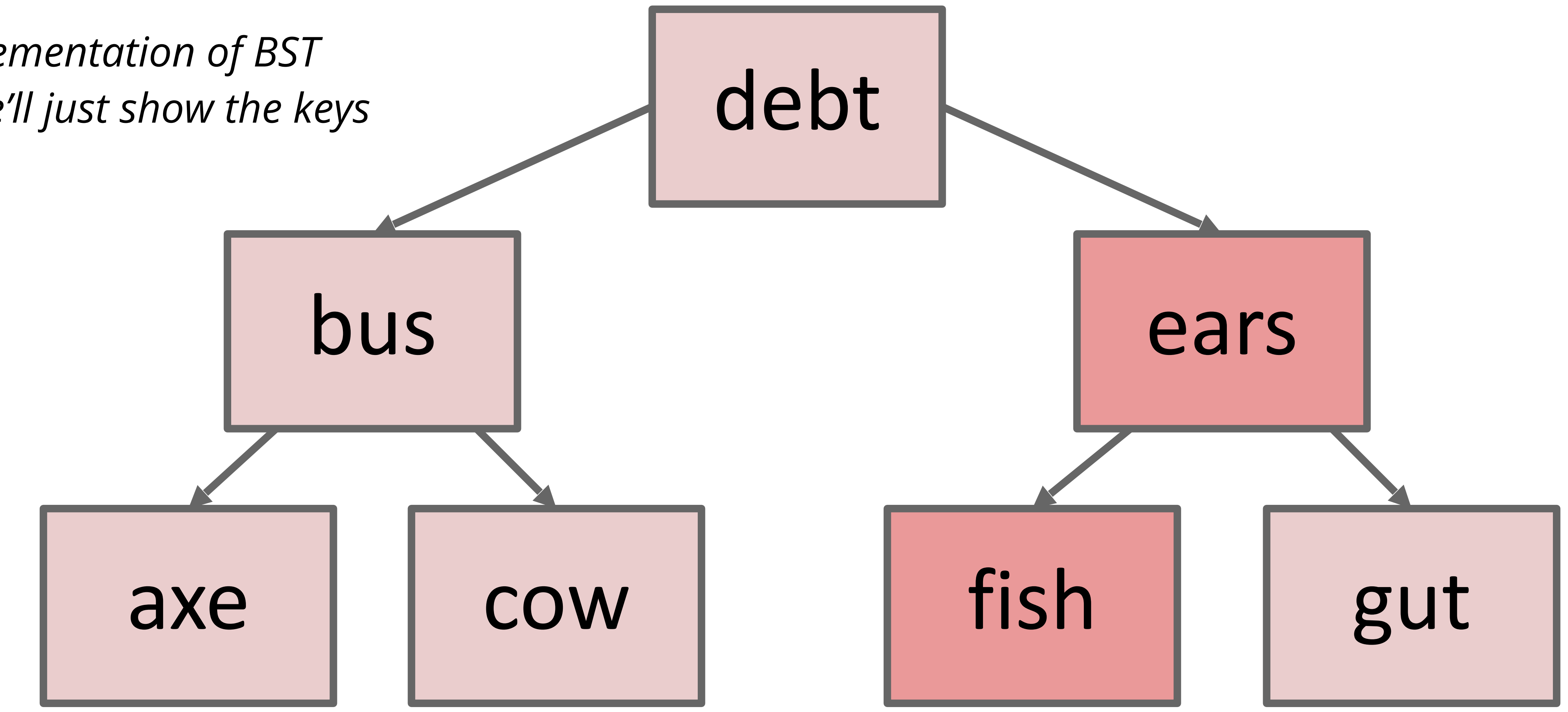
- Every key in the **left** subtree is **less** than X's key.
- Every key in the **right** subtree is **greater** than X's key.

- Q: What kind of traversal of the tree returns the nodes in sorted order? (pre-order, in-order, post-order, or level-order)?



Binary Search Tree

**Note that our specific implementation of BST takes key-value pairs, but we'll just show the keys for most visual examples*



Binary Tree, but not a Binary Search Tree

Binary Search Trees

A: An **in-order** (left, root, right) traversal of the nodes returns the nodes in sorted order.

Given keys p and q:

- Exactly one of $p < q$ and $q < p$ are true.
- $p < q$ and $q < r$ imply $p < r$.

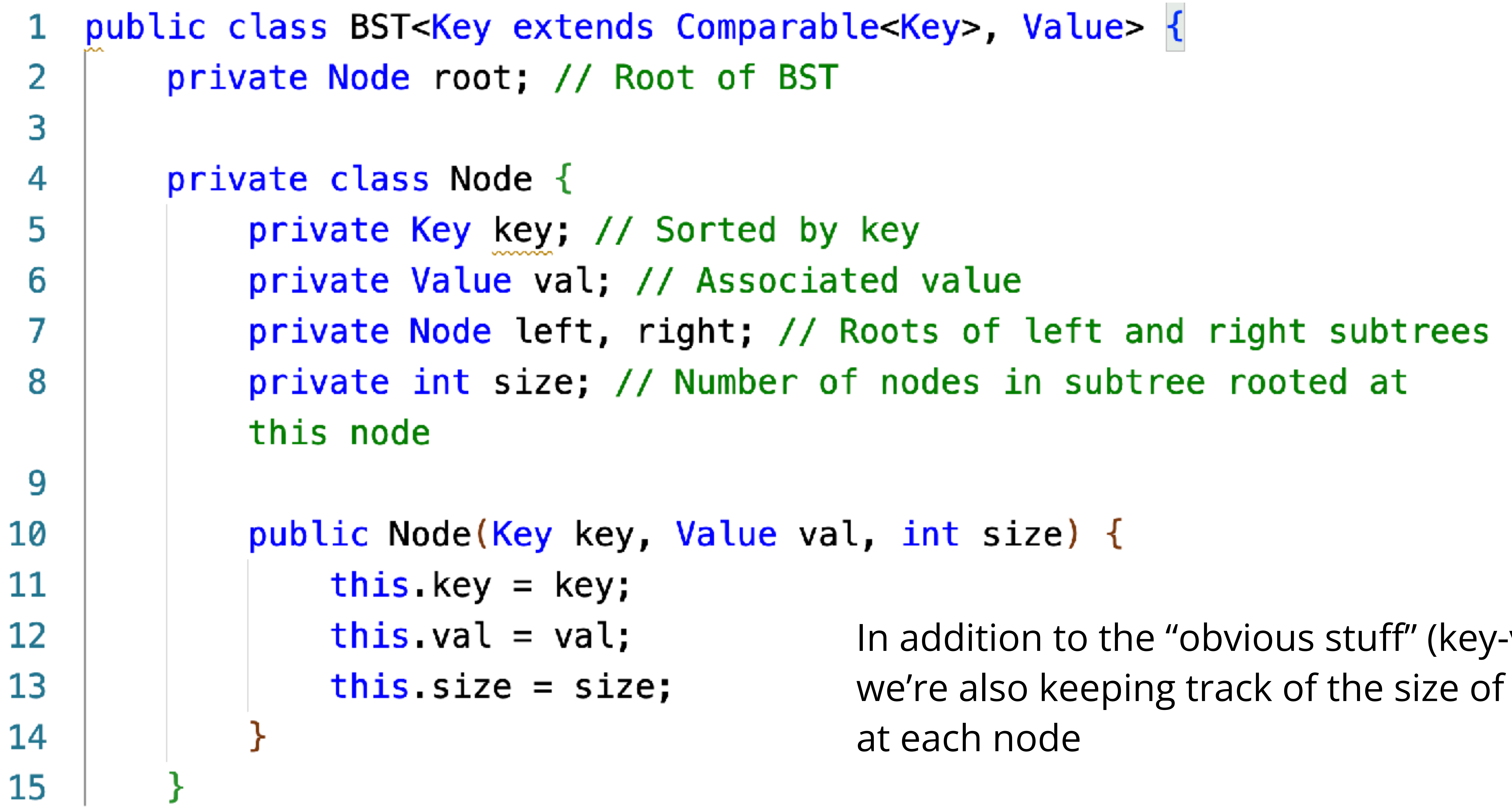
One consequence of these rules: No duplicate keys allowed!

- Keeps things simple. Most real world implementations follow this rule.

Differences between heaps and BSTs

	Heap	**BST**
Used to implement	Priority queues	Dictionaries
Supported operations	Insert, delete max	insert, search, delete, ordered operations
What is inserted	Keys	Key-value pairs
Underlying data structure	(Resizing) array	Linked nodes
Tree shape	Complete binary tree	Depends on data
Ordering of keys	Heap-ordered	Symmetrically-ordered
Duplicate keys allowed?	Yes	No

BST and Node implementation



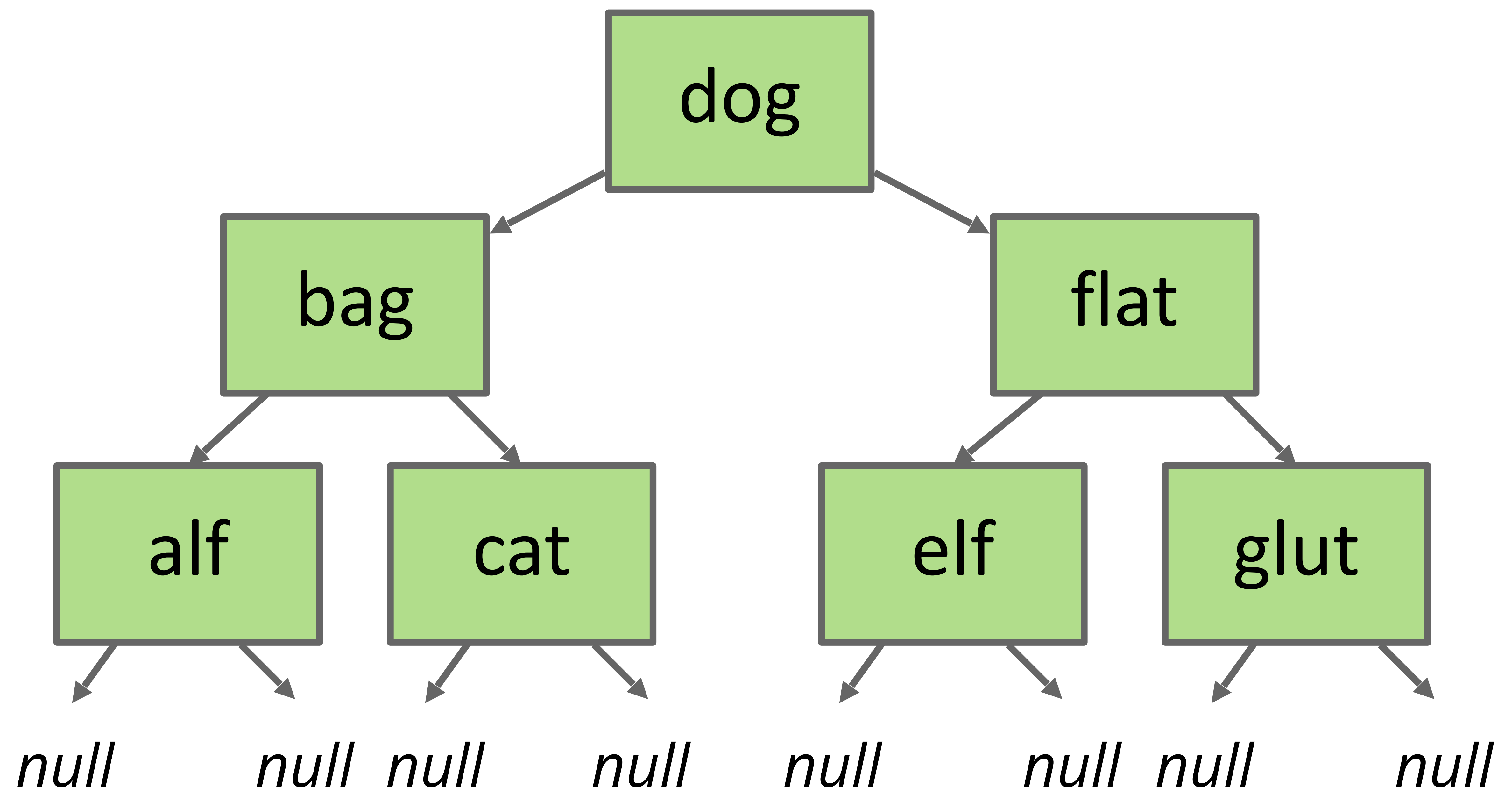
```java
public class BST<Key extends Comparable<Key>, Value> {
    private Node root; // Root of BST

    private class Node {
        private Key key; // Sorted by key
        private Value val; // Associated value
        private Node left, right; // Roots of left and right subtrees
        private int size; // Number of nodes in subtree rooted at this node

        public Node(Key key, Value val, int size) {
            this.key = key;
            this.val = val;
            this.size = size;
        }
    }
```

In addition to the “obvious stuff” (key-value pairs), we’re also keeping track of the size of the subtree at each node

BSTs: a recursive data structure

Base case: Node is null

Otherwise, a BST is a node and the BST made from its Nodes



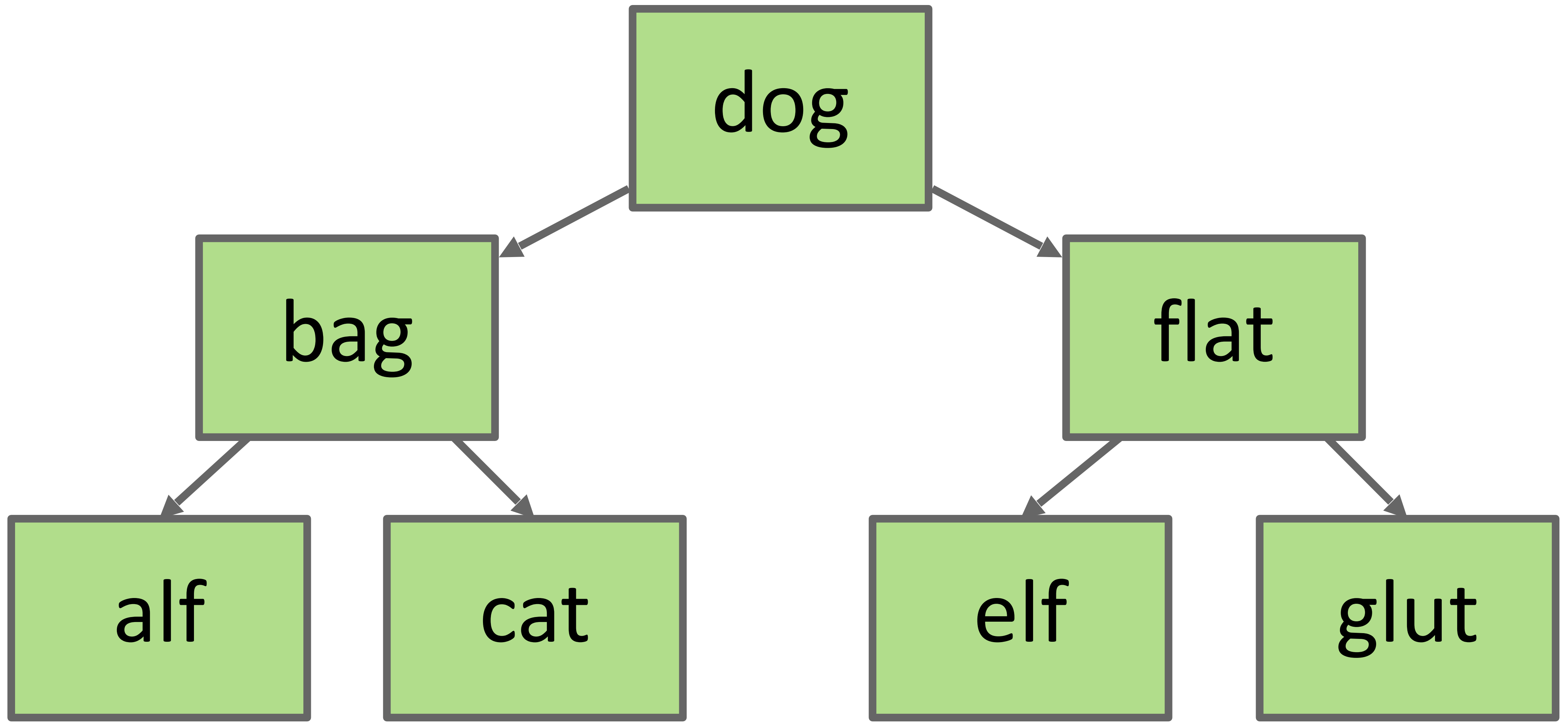
**Note that our specific implementation of BST takes key-value pairs, but we'll just show the keys for most visual examples*

Binary Search Trees: Searching

Finding a searchKey in a BST

If searchKey equals Node.key, return.

- If searchKey < Node.key, search Node.left.
- If searchKey > Node.key, search Node.right.

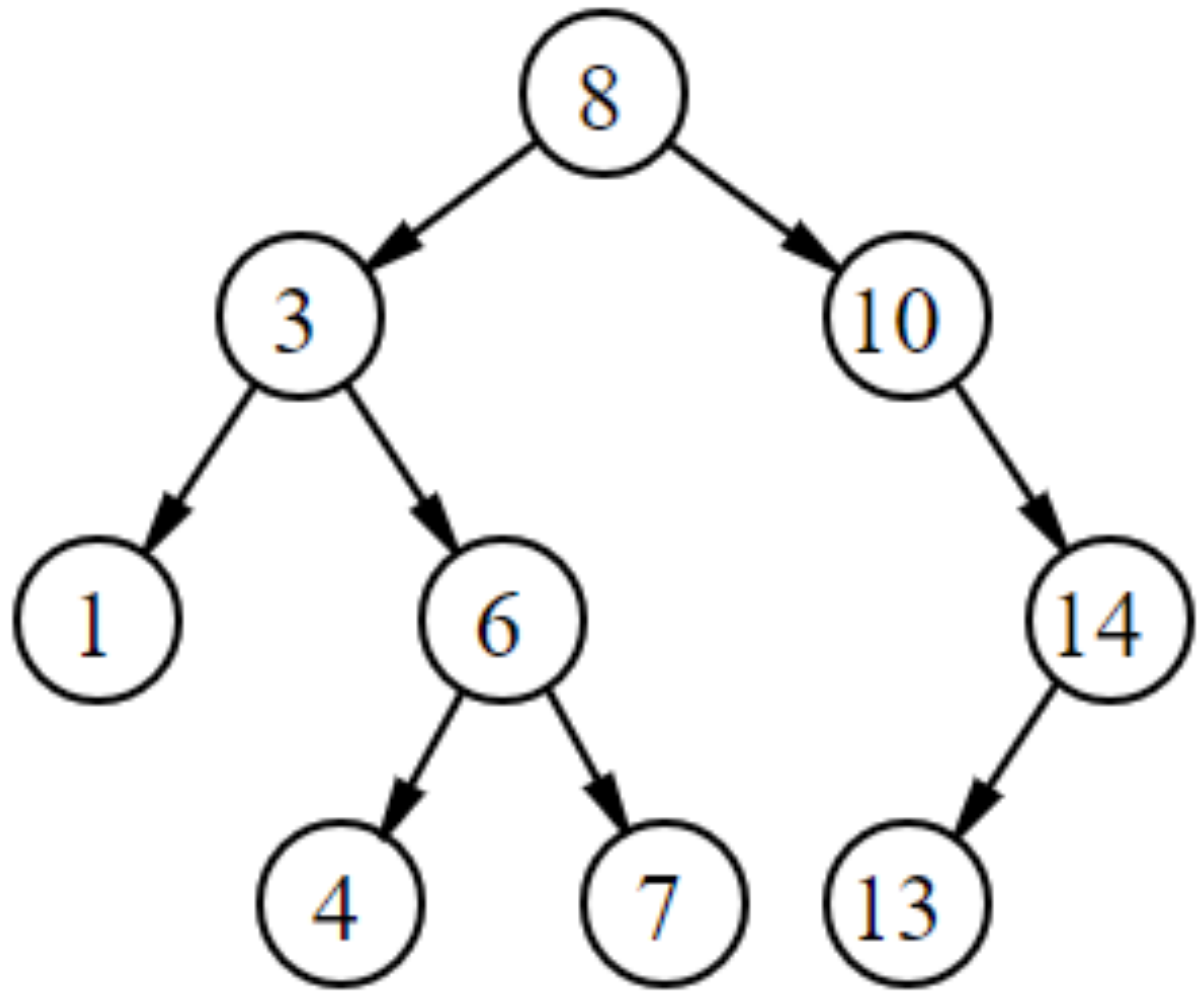


Search - recursive implementation

```
public Value get(Key key) { //recursive implementation
    return get(root, key);
}

private Value get(Node x, Key key) {
    if (x == null) return null;    If we've reached a child, the key doesn't exist
    int cmp = key.compareTo(x.key);
    if (cmp < 0) return get(x.left, key);  Recursively search left (smaller)
    else if (cmp > 0) return get(x.right, key);  Recursively search right (bigger)
    else return x.val;  We found the node
}
```

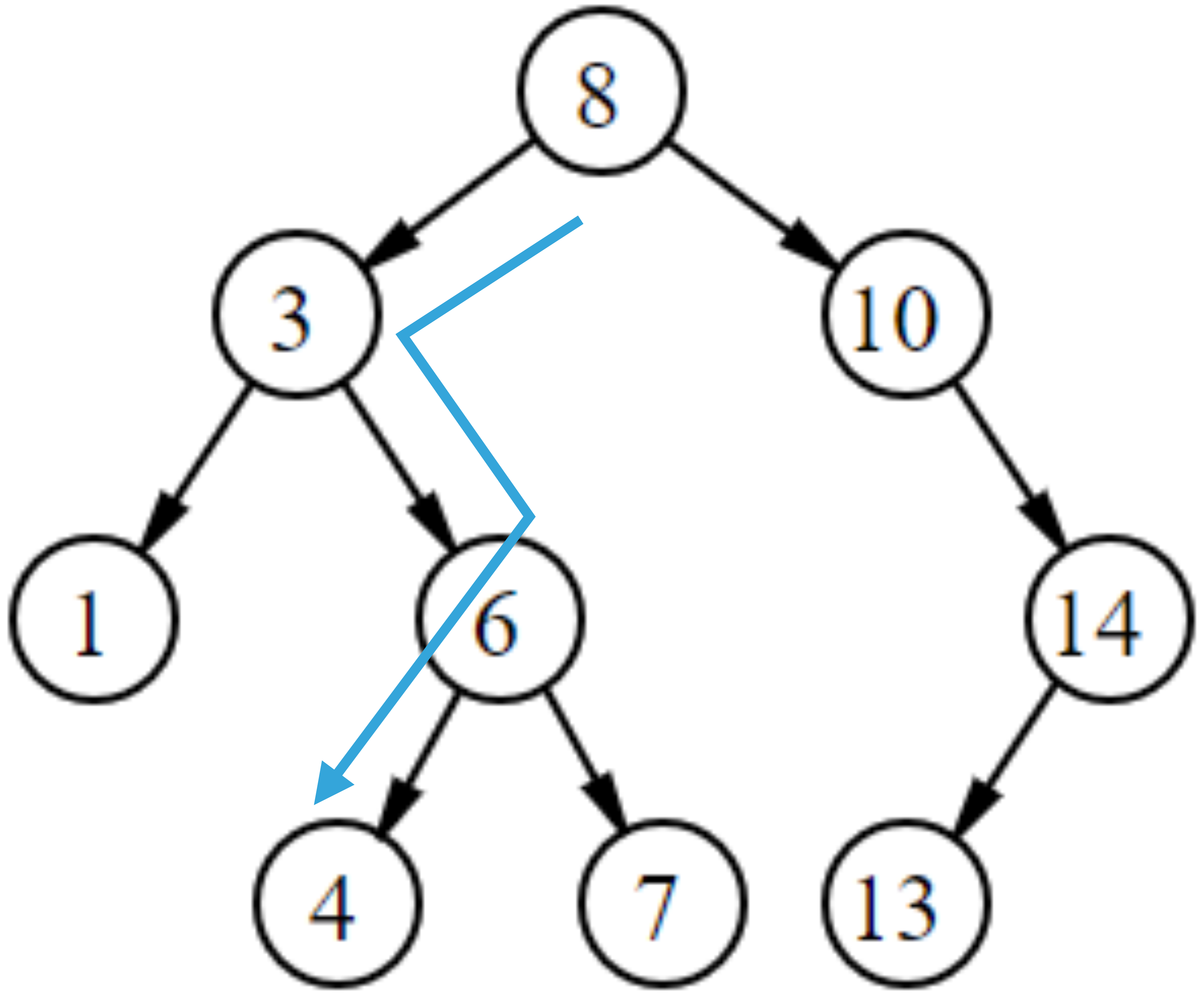
Worksheet time!

- Find 4 and 9 in the following BST. Draw the route the search takes.

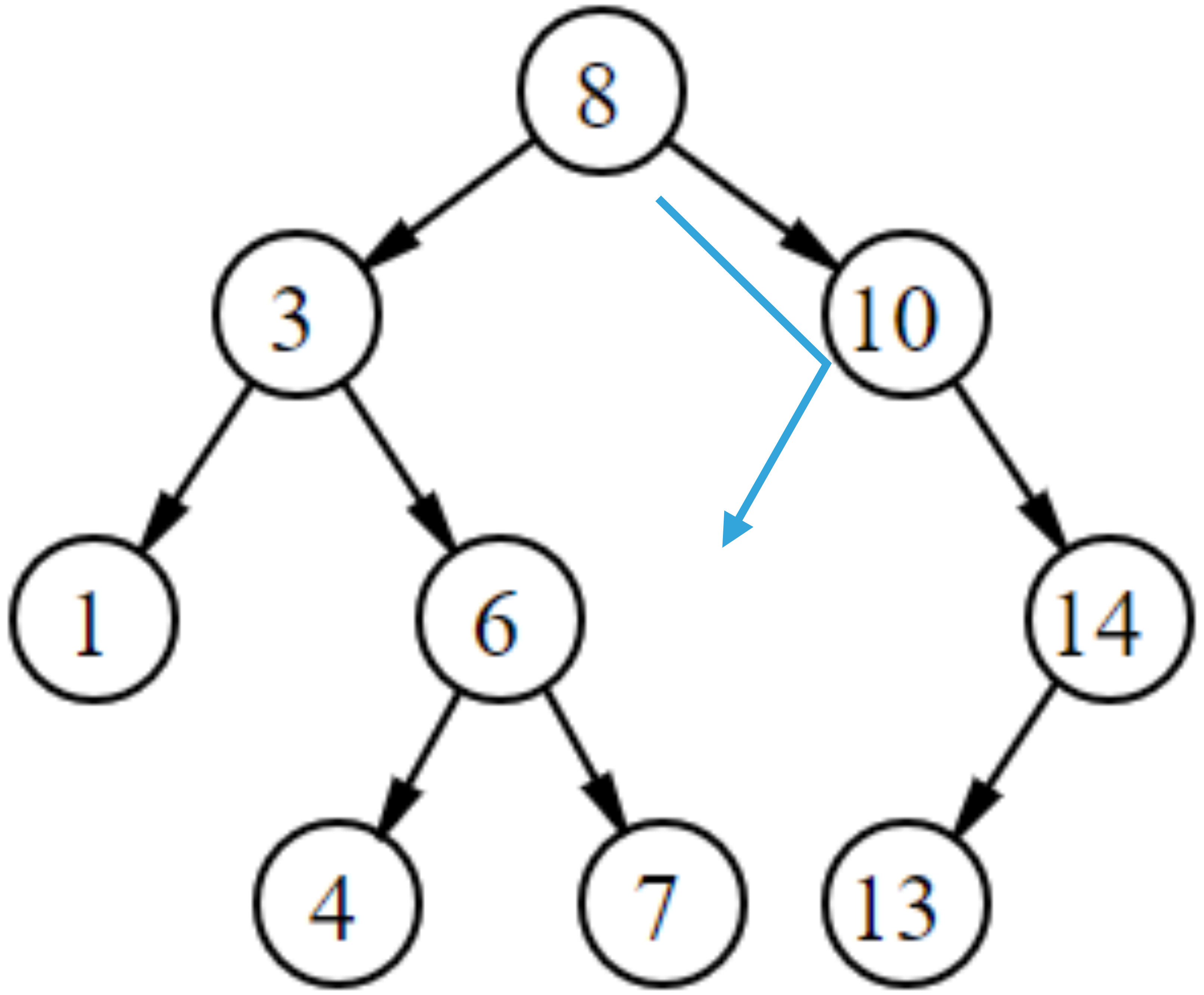


Worksheet answers

- 4: 8 -> 3 -> 6 -> 4



- 9: 8 -> 10 -> null



Search – iterative implementation

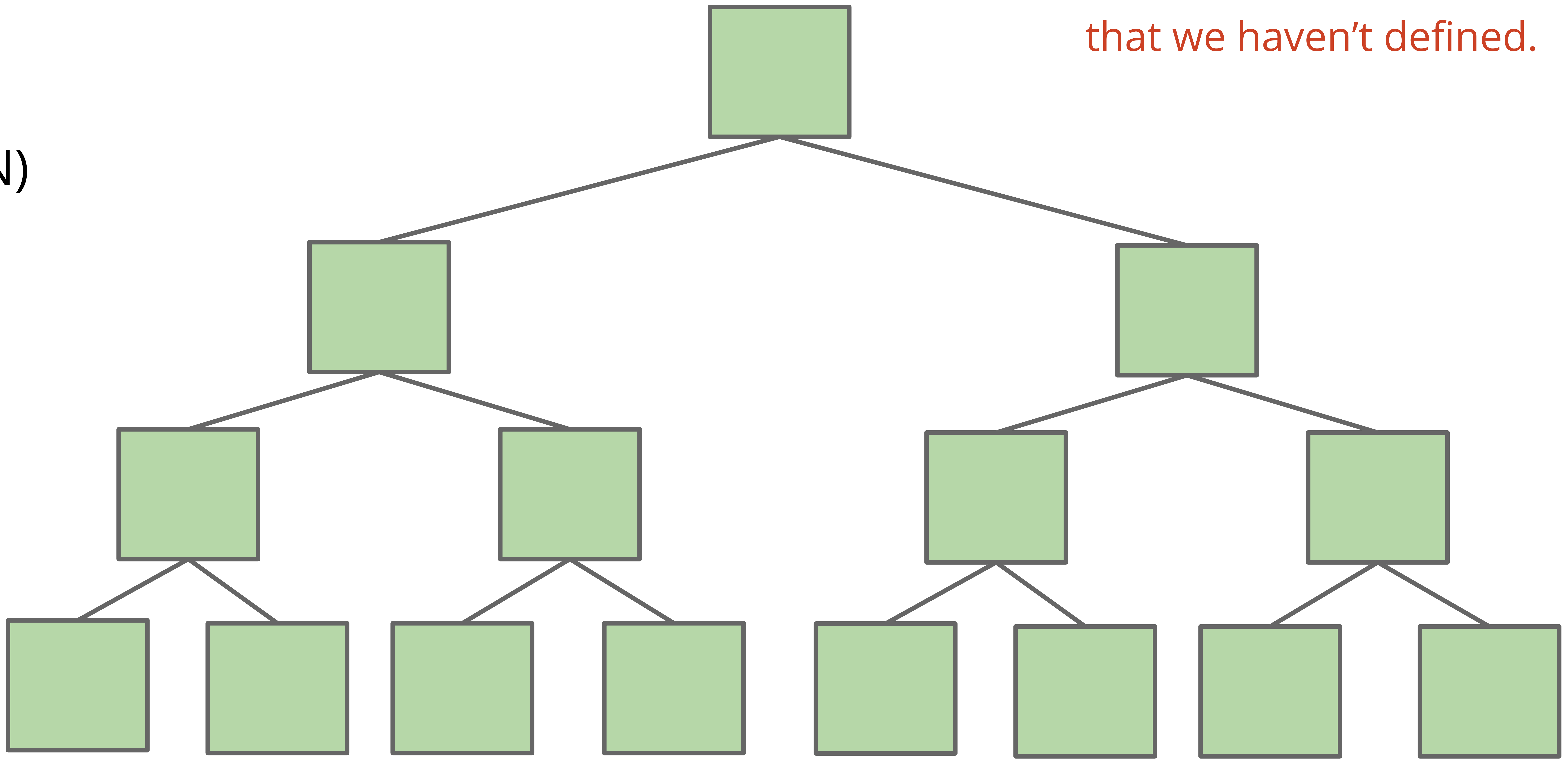
```
public Value get(Key key) {
     Node x = root;
     while (x != null) {
           int cmp = key.compareTo(x.key);
           if (cmp < 0)
                   x = x.left;
           else if (cmp > 0)
                   x = x.right;
           else if (cmp == 0)
                   return x.val;
      }
      return null;
}
```

Question

What is the runtime to complete a search on a “bushy” BST in the worst case, where N is the number of nodes?

“bushiness” is an intuitive concept that we haven’t defined.

A. Θ(log N)
B. Θ(N)
C. Θ(N log N)
D. $Θ(N^2)$
E. $Θ(2^N)$



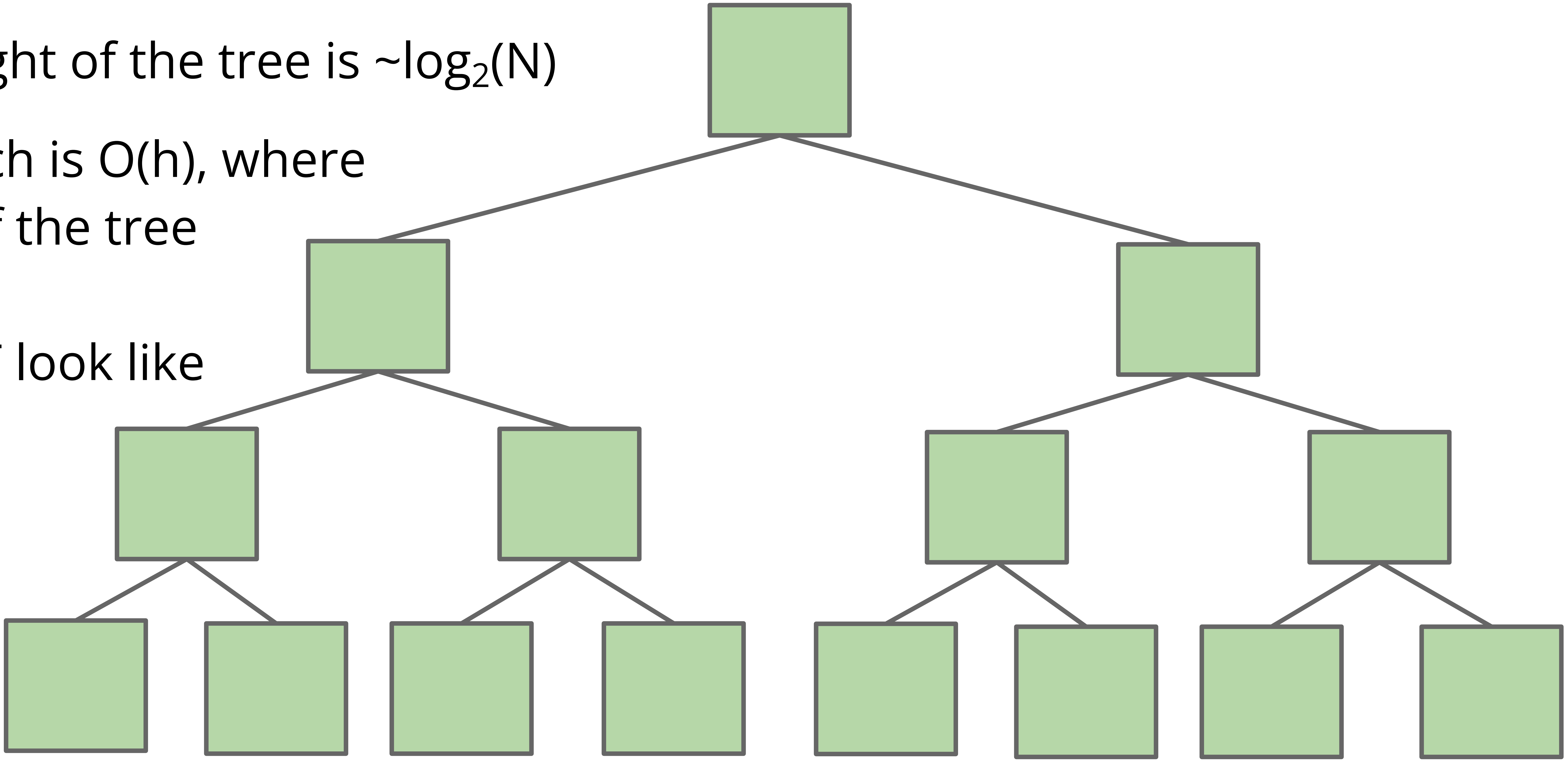
BST Search

What is the runtime to complete a search on a “bushy” BST in the worst case, where N is the number of nodes?

A. **Θ(log N) :** Height of the tree is ~$\log_2(N)$

Worst case search is O(h), where h is the height of the tree

What does a BST look like that has O(n) search?



BSTs

Bushy BSTs are extremely fast.

- At 1 microsecond per operation, can find something from a tree of size 10^{300000} in one second.

Much (perhaps most?) computation is dedicated towards finding things in response to queries.

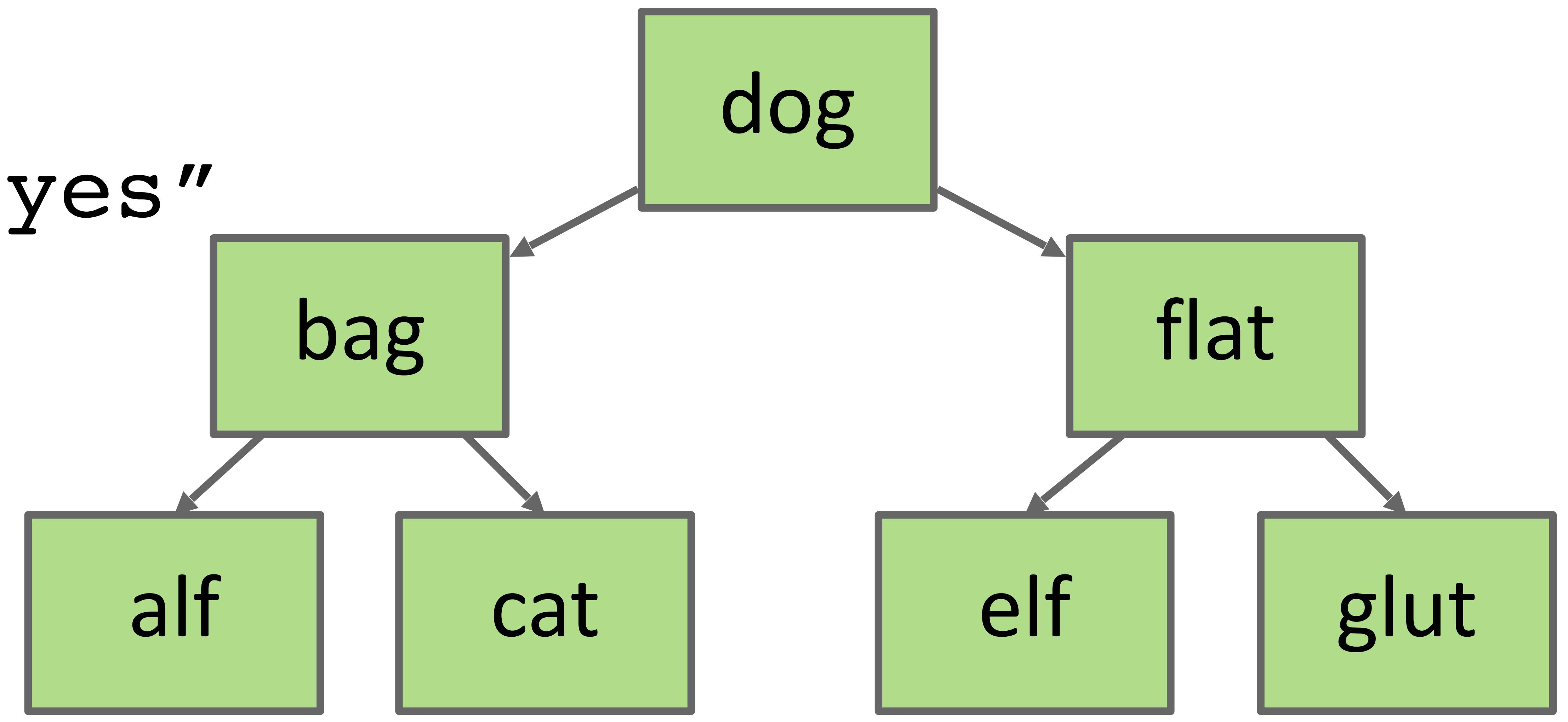
- It’s a good thing that we can do such queries almost for free.

BSTs: Insertion

Inserting a New Key into a BST

Example:

`insert “eyes”`

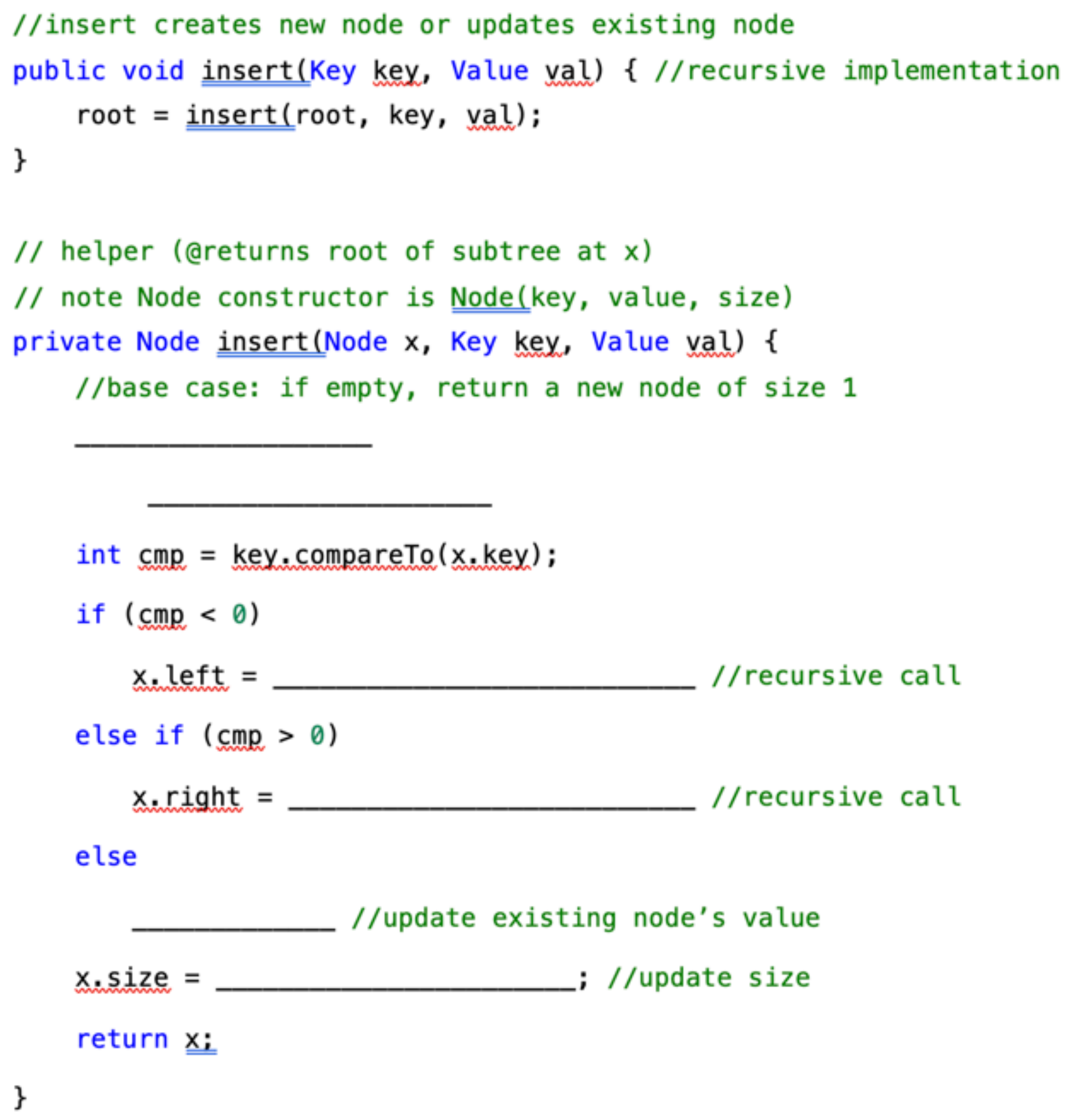


Search for key.

- If found, do nothing.
- If not found:
  - Create new node.
  - Set appropriate link.
- Number of compares is equal to the depth of the node + 1.

Worksheet time!

- Fill in the blanks to implement insert.



```
//insert creates new node or updates existing node
public void insert(Key key, Value val) { //recursive implementation
    root = insert(root, key, val);
}

// helper (@returns root of subtree at x)
// note Node constructor is Node(key, value, size)
private Node insert(Node x, Key key, Value val) {
    //base case: if empty, return a new node of size 1
    ___________________

        ______________________

    int cmp = key.compareTo(x.key);
    if (cmp < 0)
        x.left = ___________________________ //recursive call
    else if (cmp > 0)
        x.right = __________________________ //recursive call
    else
        _____________ //update existing node’s value
    x.size = _______________________; //update size
    return x;
}
```

```
//insert creates new node or updates existing node
public void insert(Key key, Value val) { //recursive implementation
    root = insert(root, key, val);
}

// helper (@returns root of subtree at x)
// note Node constructor is Node(key, value, size)
private Node insert(Node x, Key key, Value val) {
    //base case: if empty, return a new node of size 1
    ___________________                if (x == null)

      ______________________               return new Node(key, val, 1)

    int cmp = key.compareTo(x.key);

    if (cmp < 0)
                  insert(x.left, key, val)
        x.left = ___________________________ //recursive call

    else if (cmp > 0)
                  insert(x.right, key, val)
        x.right = __________________________ //recursive call

    else
        x.val = val
        _____________ //update existing node's value

    x.size = _______________________; //update size
                size(x.left) + size(x.right) + 1
    return x;
}
```

We have a recursive definition of size: the size of a subtree is that is not null is 1 (itself) + the size of its left and right subtrees

http://algs4.cs.princeton.edu

3.2 BINARY SEARCH TREE DEMO

BSTs mathematical analysis

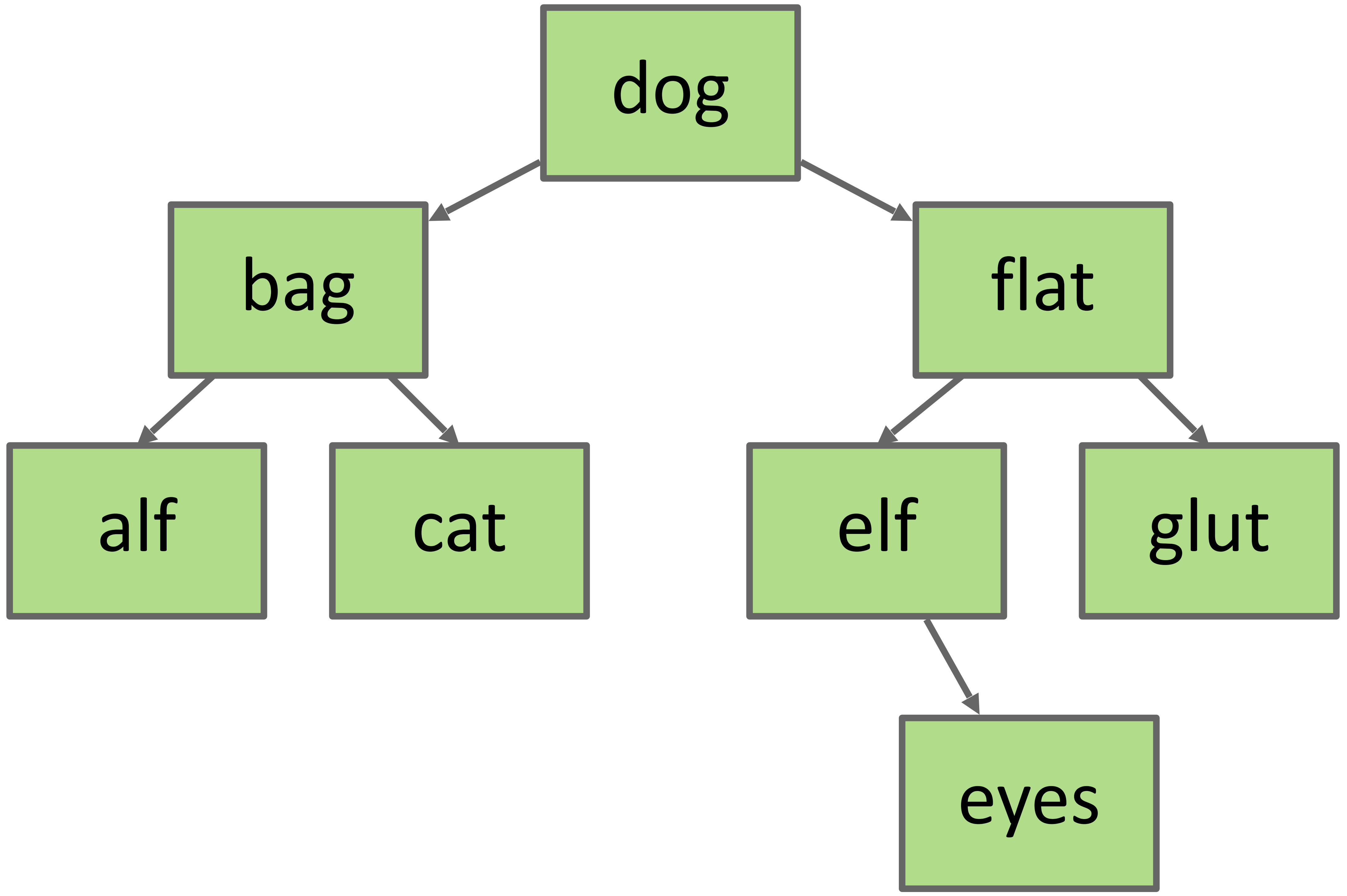
- If n distinct keys are inserted into a BST in random order, the expected number of compares of search/insert is $O(\log n)$.
  - If n distinct keys are inserted into a BST in random order, the expected height of tree is $O(\log n)$. [Reed, 2003].
- Worst case height is n but highly unlikely.
  - Keys would have to come (reversely) sorted!
- All ordered operations in a dictionary implemented with a BST depend on the height of the BST. You can assume the BST is reasonably “bushy” (log(n) time).

BSTs: Hibbard Deletion

Deleting from a BST

3 Cases:

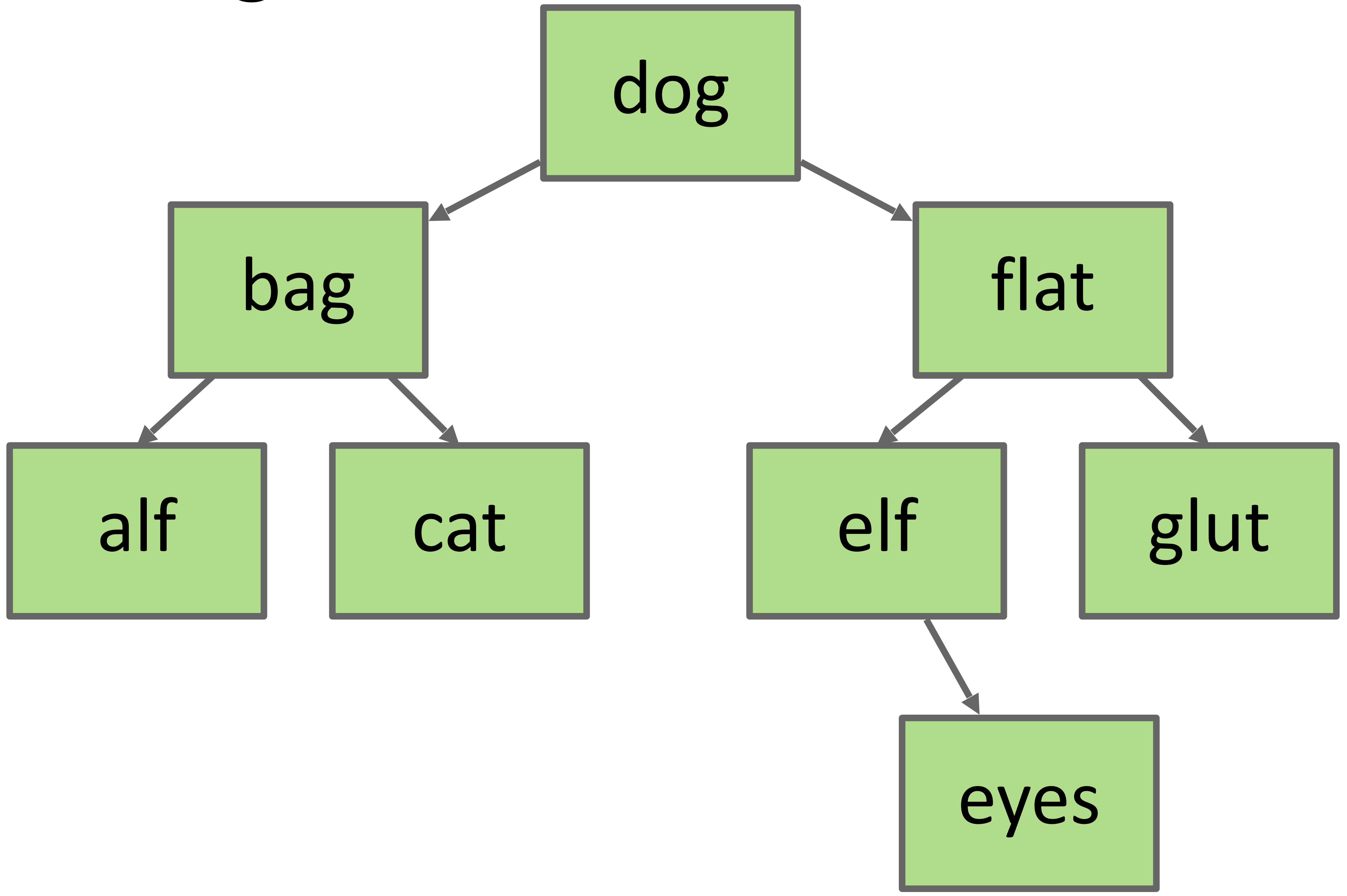
- Deletion key has no children.
- Deletion key has one child.
- Deletion key has two children.



Case 1: Deleting from a BST: Key with no Children

Deletion key has no children (“glut”):

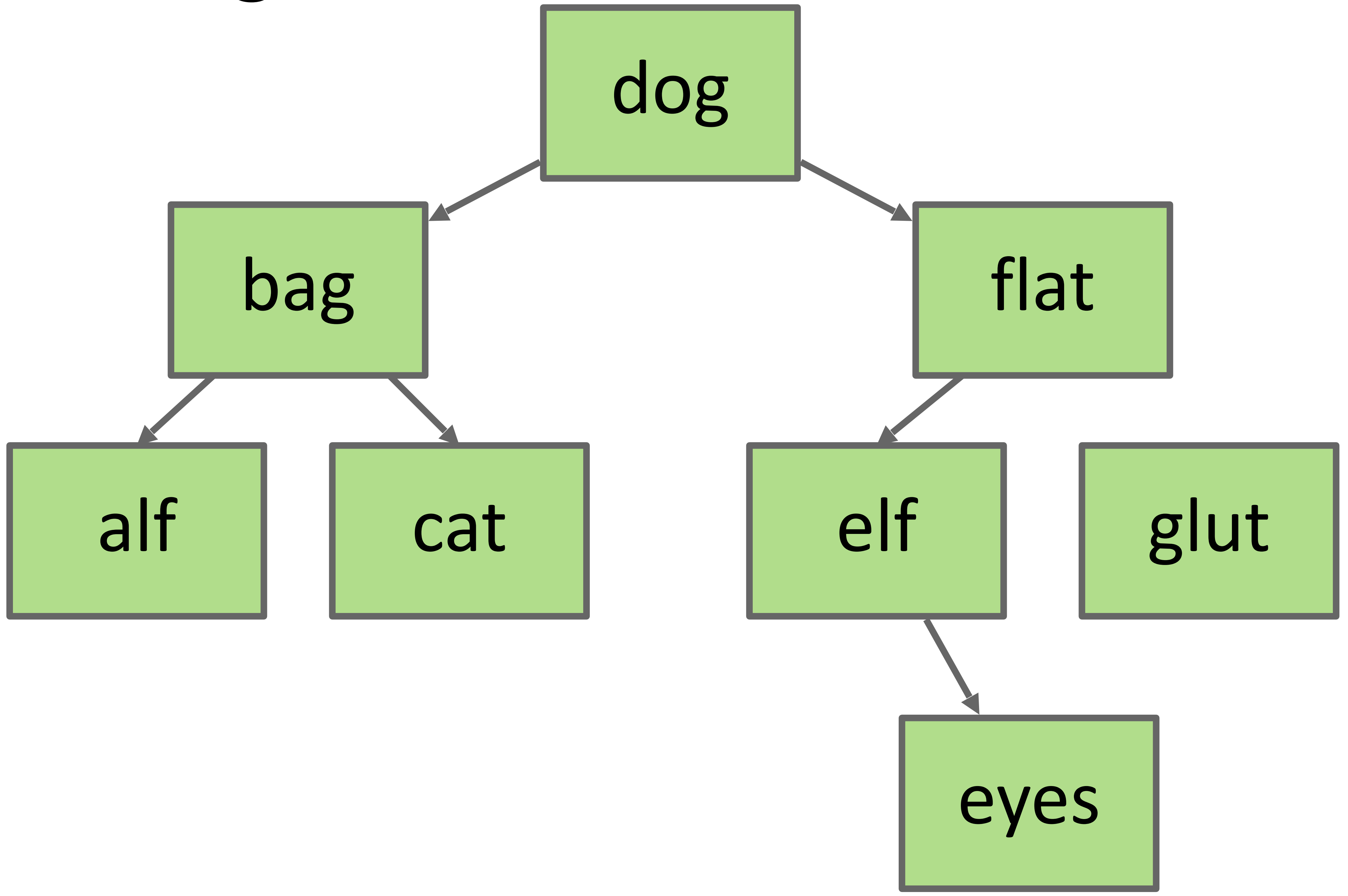
- Just sever the parent’s link.
- What happens to “glut” node?



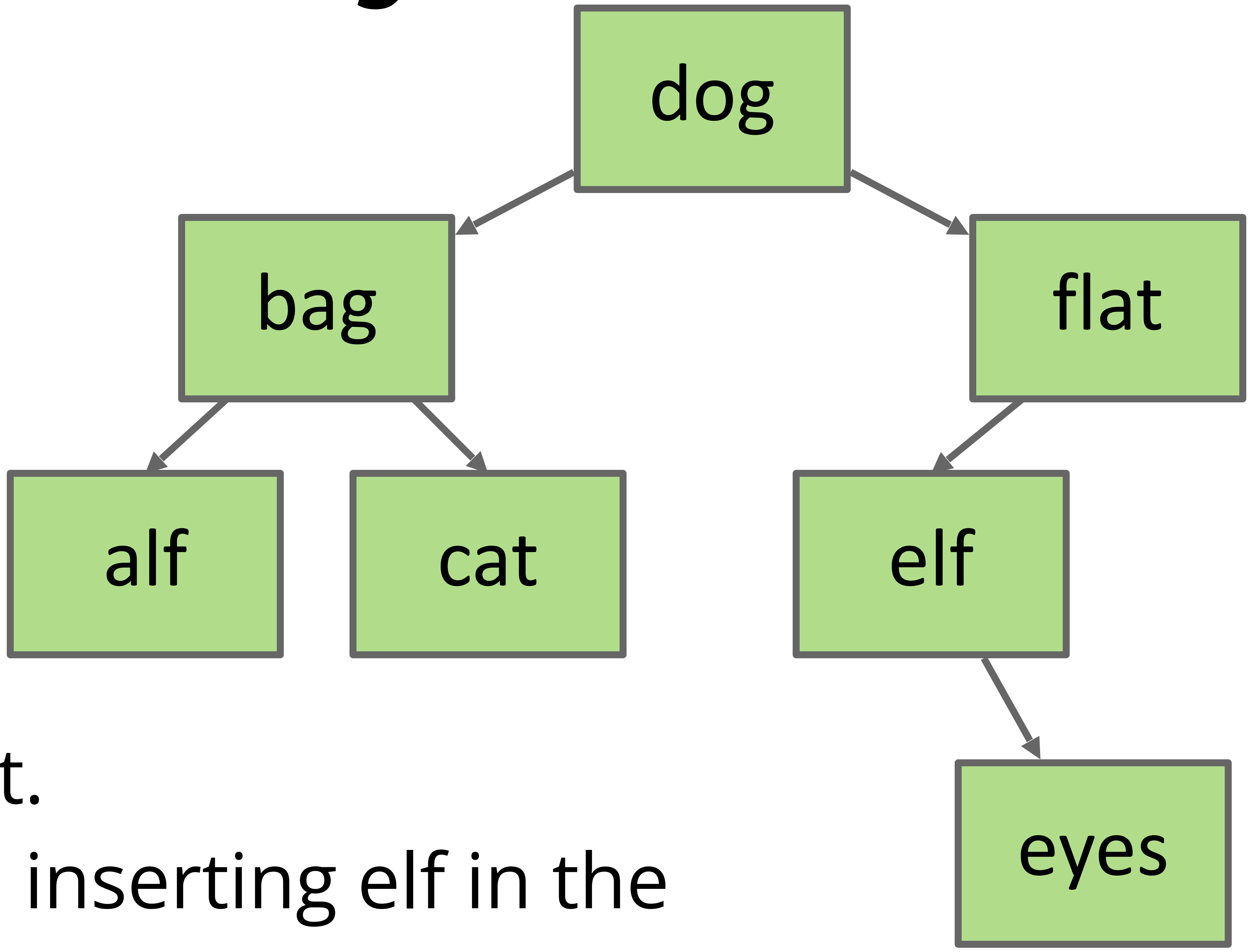
Case 1: Deleting from a BST: Key with no Children

Deletion key has no children (“glut”):

- Just sever the parent’s link.
- What happens to “glut” node?
  - Garbage collected.



Case 2: Deleting from a BST: Key with one Child



Example: delete("flat"):

Goal:

- Maintain symmetric order (BST property).
- Flat's child elf is still larger than dog.
  - Safe to just move that child into flat's spot.
  - Why? Because of the BST property. When inserting elf in the BST originally, it had to have gone to the right of the dog.

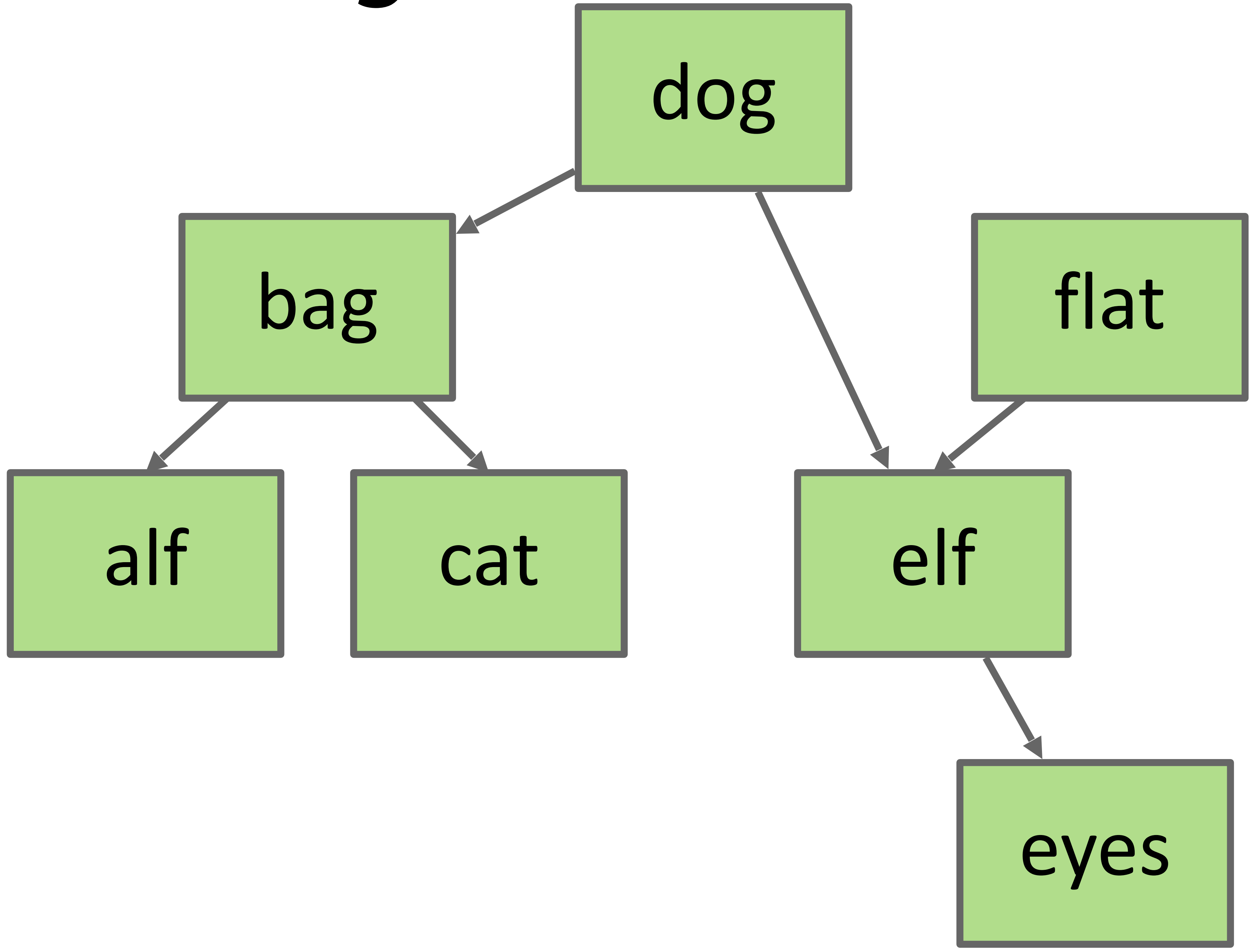
Thus: Move flat's parent's pointer to flat's child.

Case 2: Deleting from a BST: Key with one Child

Example: delete(“flat”):

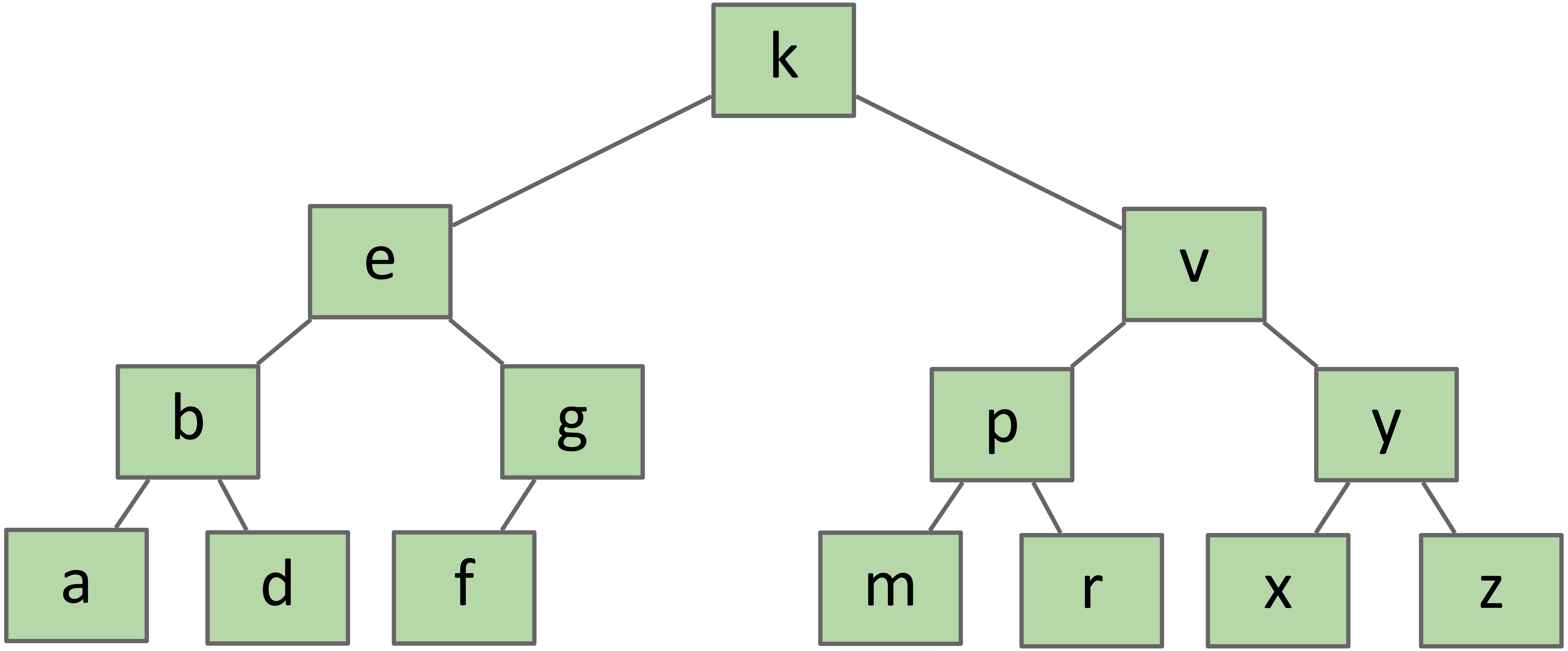
Thus: Move flat’s parent’s pointer to flat’s child.

- Flat will be garbage collected (along with its instance variables).
- Even though flat still links to elf, we can’t access it because nothing points to it.



Hard Challenge

Delete k. How do you choose the new root?

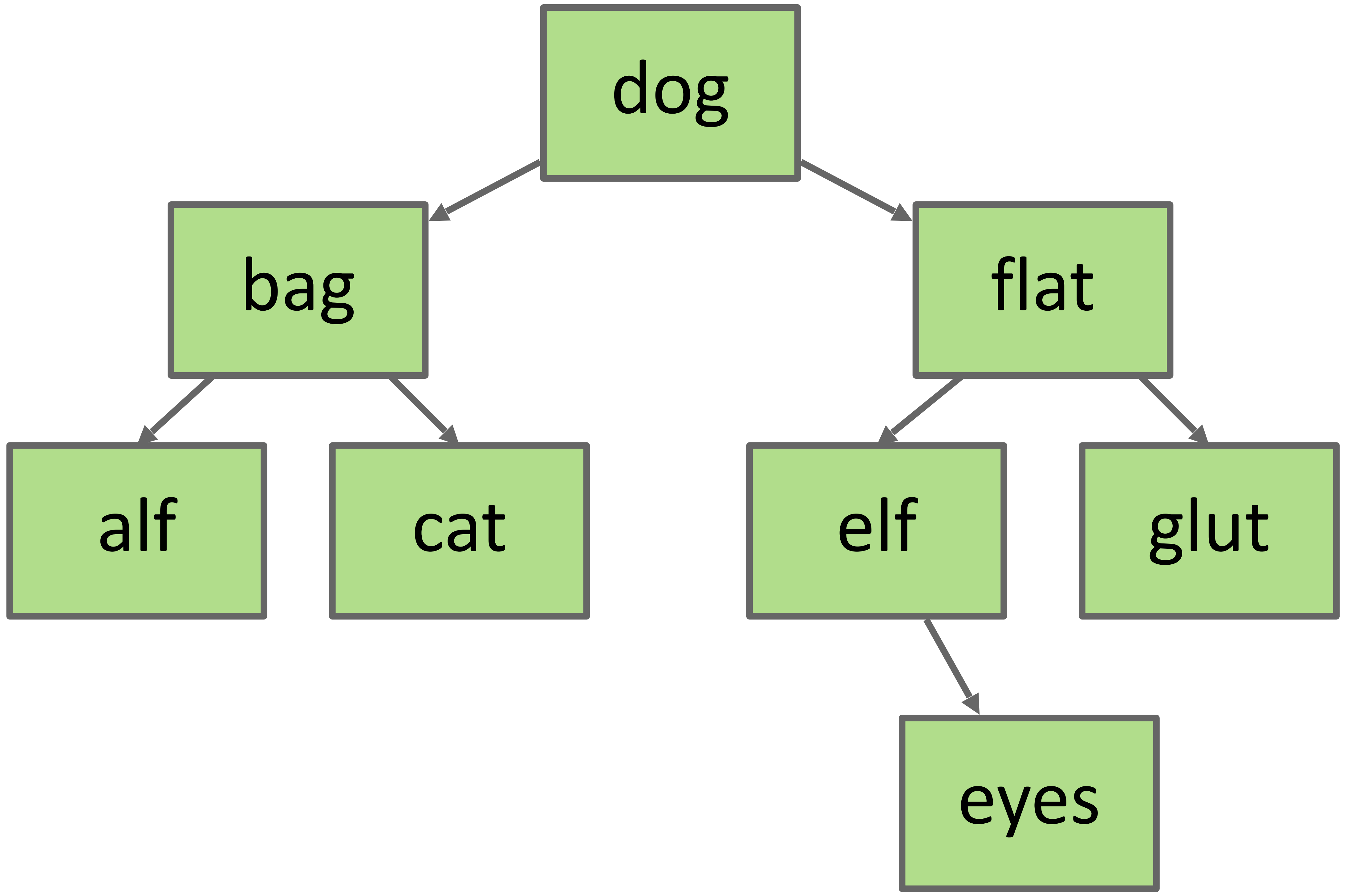


Case 3: Deleting from a BST: Deletion with two Children (Hibbard)

Example: delete(“dog”)

Goal:

- Find a new root node.
- Must be > than everything in left subtree.
- Must be < than everything right subtree.



Would bag work?

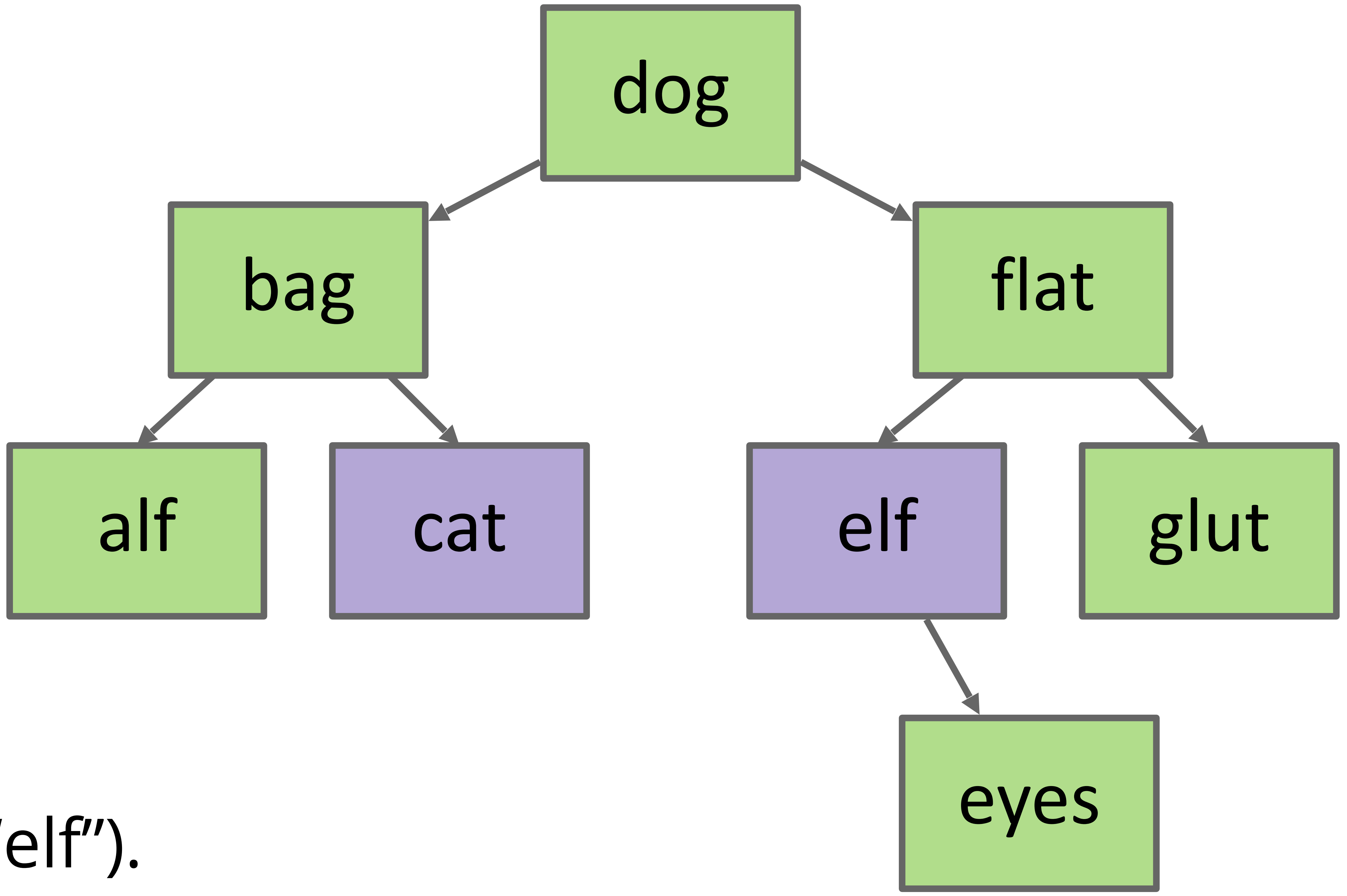
No: We can keep alf as its left child, but where does cat go? Replacing flat with cat requires too many movements/adjustments and the cases get really messy quickly

Case 3: Deleting from a BST: Deletion with two Children (Hibbard)

Example: delete(“dog”)

Goal:

- Find a new root node.
- Must be > than everything in left subtree.
- Must be < than everything right subtree.



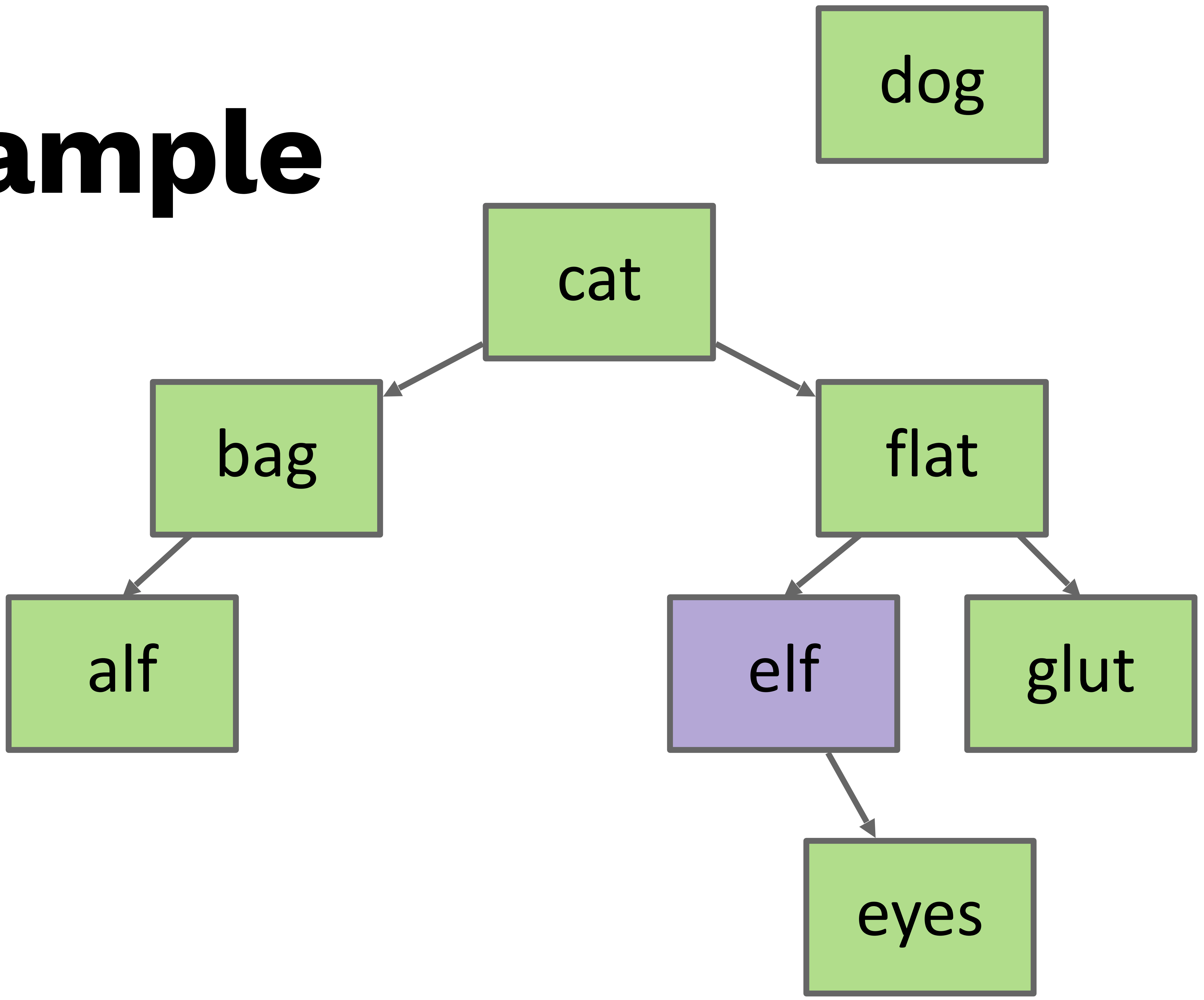
Choose either predecessor (“cat”) or successor (“elf”).

- Predecessor = largest key in left subtree
- Successor = smallest key in right subtree
- Delete “cat” or “elf”, and stick a new copy of that node in the root position:
  - This deletion guaranteed to be either case 1 or 2.
  - By deleting “cat” or “elf”, we replace that node with its subtree
- This strategy is sometimes known as “Hibbard deletion”.

Choose predecessor example

Example: delete("dog")

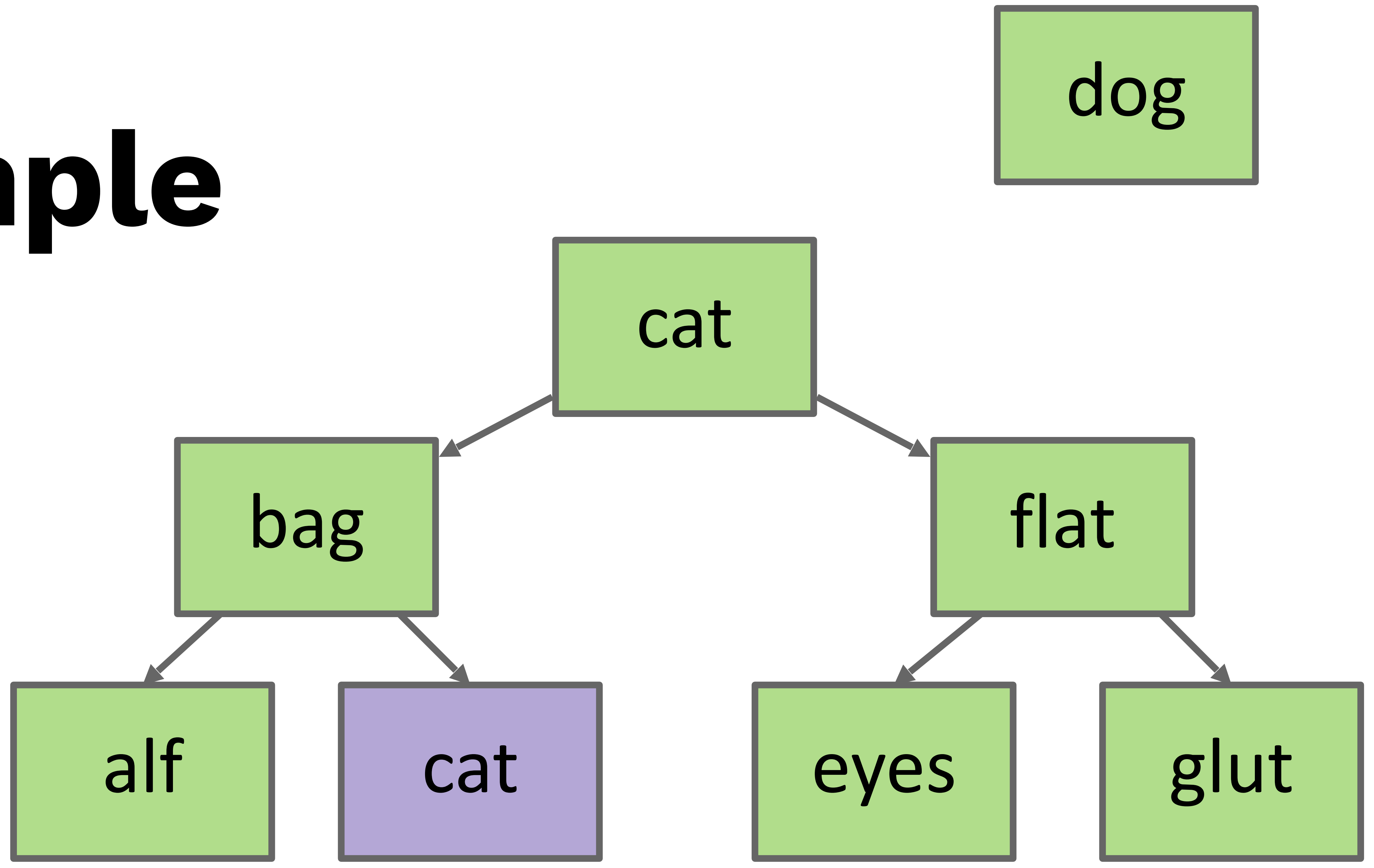
- cat has replaced dog
- cat's subtree (null) is in the place of cat



Choose successor example

Example: delete(“dog”)

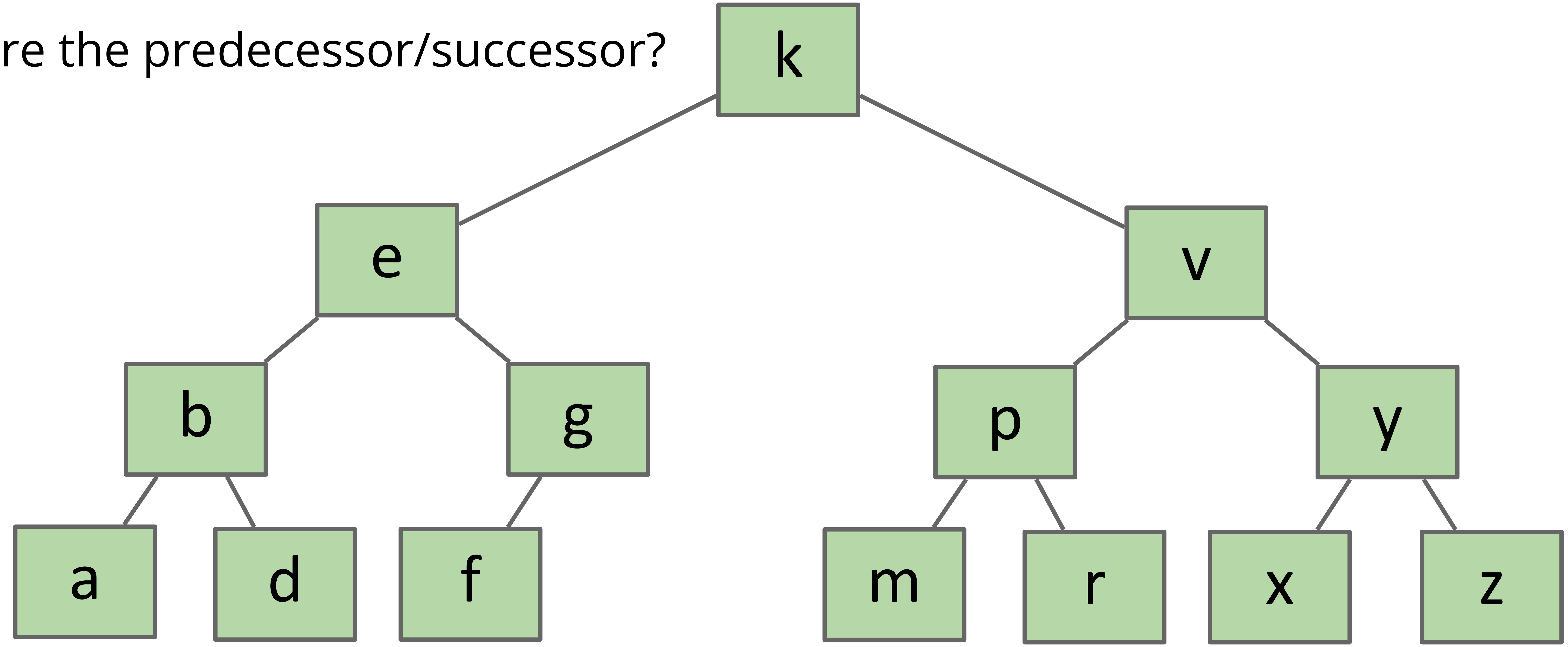
- elf has replaced dog
- elf’s subtree (eyes) is in the place of elf



Hard Challenge (Hopefully Now Easy)

Delete k.

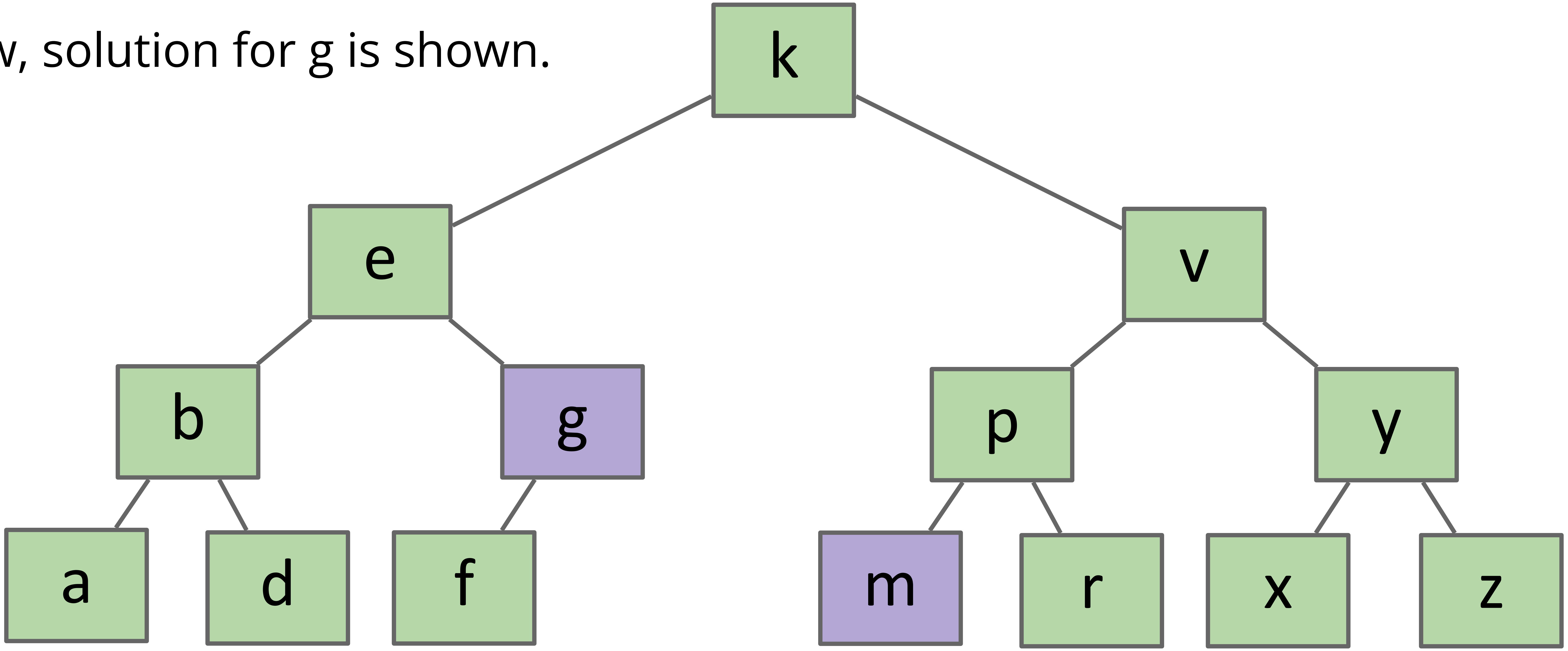
What are the predecessor/successor?



Hard Challenge (Hopefully Now Easy)

Delete k. Two solutions: Either promote g or m to be in the root.

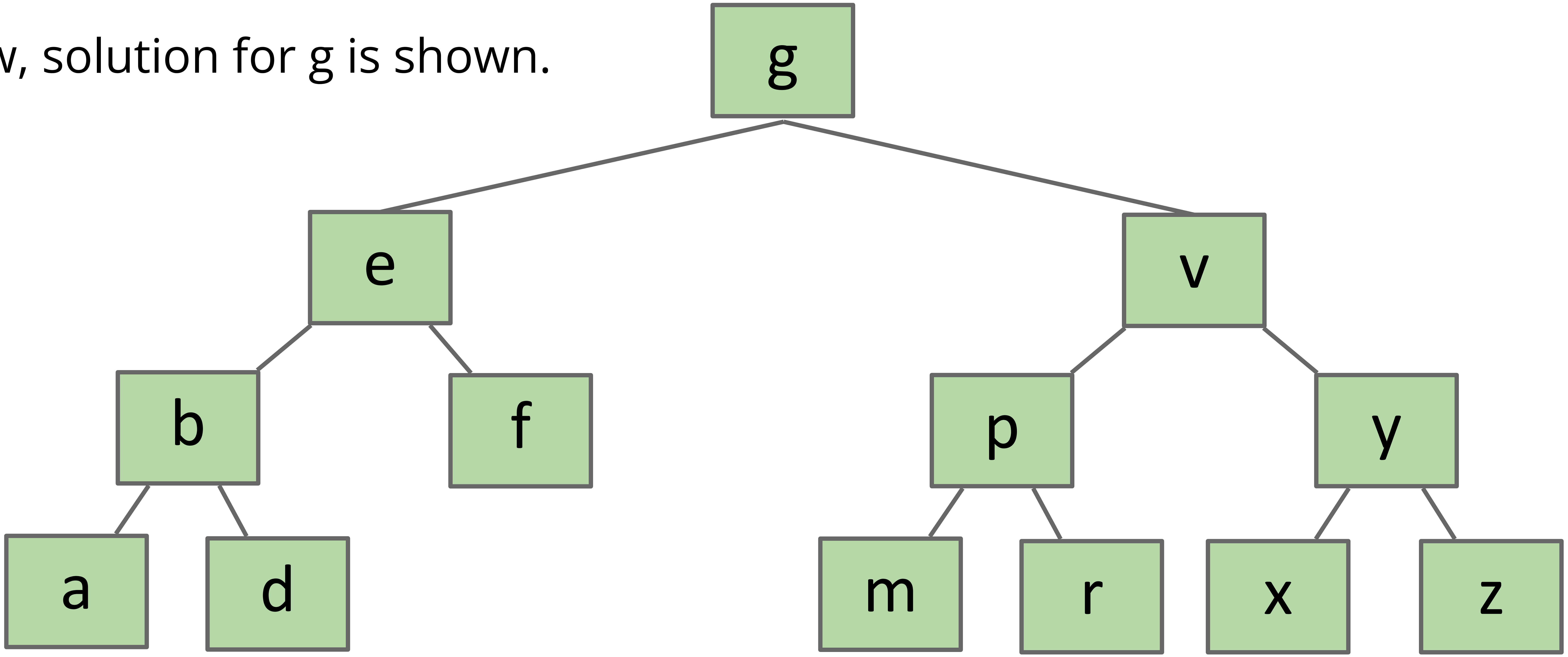
- Below, solution for g is shown.



Hard Challenge (Hopefully Now Easy)

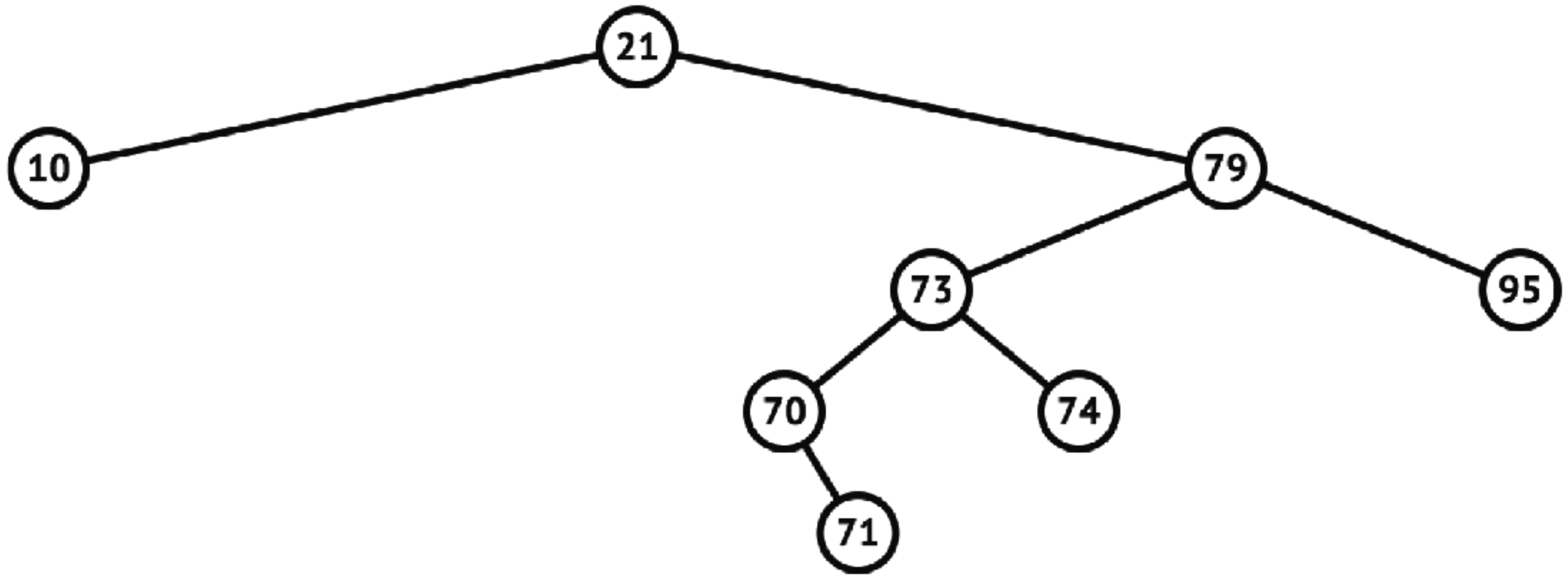
Two solutions: Either promote g or m to be in the root.

- Below, solution for g is shown.



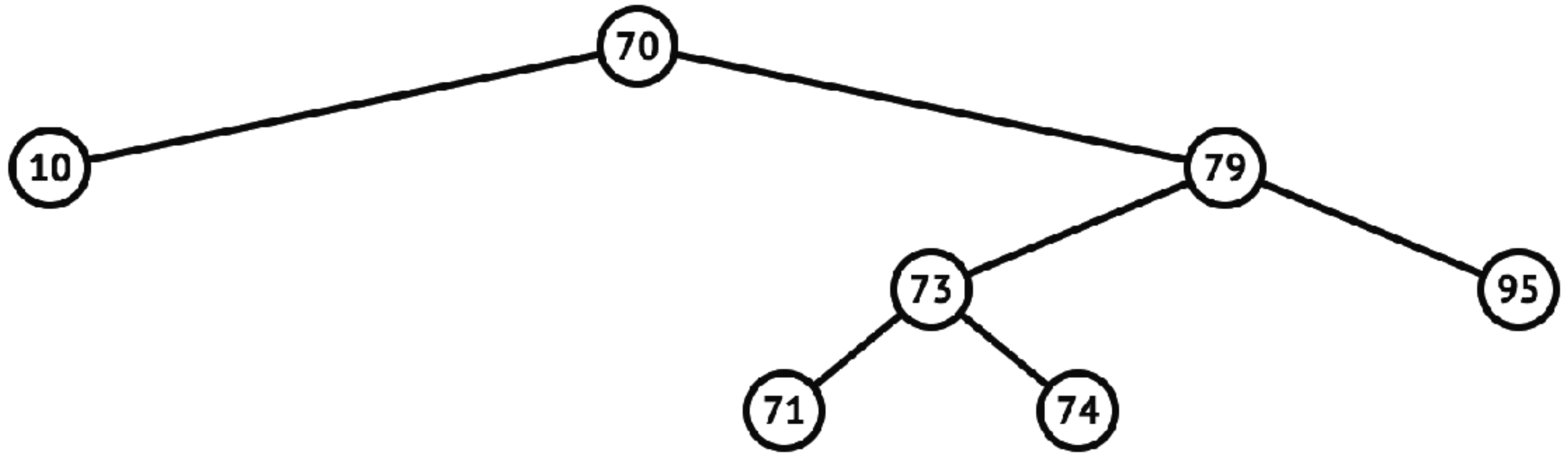
Worksheet time!

- Delete 21 in this tree. Choose the successor.



Worksheet answer

- 70 is the successor, and its subtree (71) moves into 70’s place



Hibbard deletion

```
public void delete(Key key) { //recursive implementation
    root = delete(root, key);
}

//helper (@returns root of new subtree at x)
private Node delete(Node x, Key key) {
    if (x == null) return null;
    //search part
    int cmp = key.compareTo(x.key);
    if (cmp < 0) x.left = delete(x.left, key);
    else if (cmp > 0) x.right = delete(x.right, key);
    //found the node, now the 3 cases
    else {
        if (x.right == null) return x.left; //1 & 2 - no or single child
        if (x.left == null) return x.right;
        Node temp = x; //3. replace with successor
        x = min(temp.right); //changes root to new successor - min key of right subtree
        x.right = deleteMin(temp.right); //new root right is old root's right side minus successor
        x.left = temp.left; //new root left is old root's left
    }
    x.size = size(x.left) + size(x.right) + 1; //recalculate size given size of subtrees plus self
    // decrements size because subtree (x.left / x.right) was probably set to null
    return x;
}
```

Hibbard’s deletion

- Unsatisfactory solution. If we were to perform many insertions and deletions the BST ends up being not symmetric and skewed to the left.
    - Extremely complicated analysis, but average cost of deletion ends up being $\sqrt{n}$. Let’s simplify things by saying it stays $O(\log n)$.
    - No one has proven that alternating between the predecessor and successor will fix this.
- Hibbard devised the algorithm in 1962. Still no algorithm for efficient deletion in Binary Search Trees!
- Overall, BSTs can have $O(n)$ worst-case for search, insert, and delete. We want to do better for dictionaries/maps (and will learn how to in future lectures!)

Lecture 17 wrap-up

- Exit ticket: https://forms.gle/Ez9hzu1CF8gAWzMu7
- HW7: Autocomplete due next Tues 11:59pm
- This is the last lecture on checkpoint 2, lab next week is checkpoint review

Resources

- Reading from textbook: Chapters 3.2 (Pages 396–414); https://algs4.cs.princeton.edu/32bst/
- BST visualization: https://visualgo.net/en/bst
- Practice problems behind this slide

Problem 1

- Draw the BST that results when you insert the keys 5, 1, 19, 25, 17, 5, 19, 20, 9, 15, 14 in that order.

Problem 2

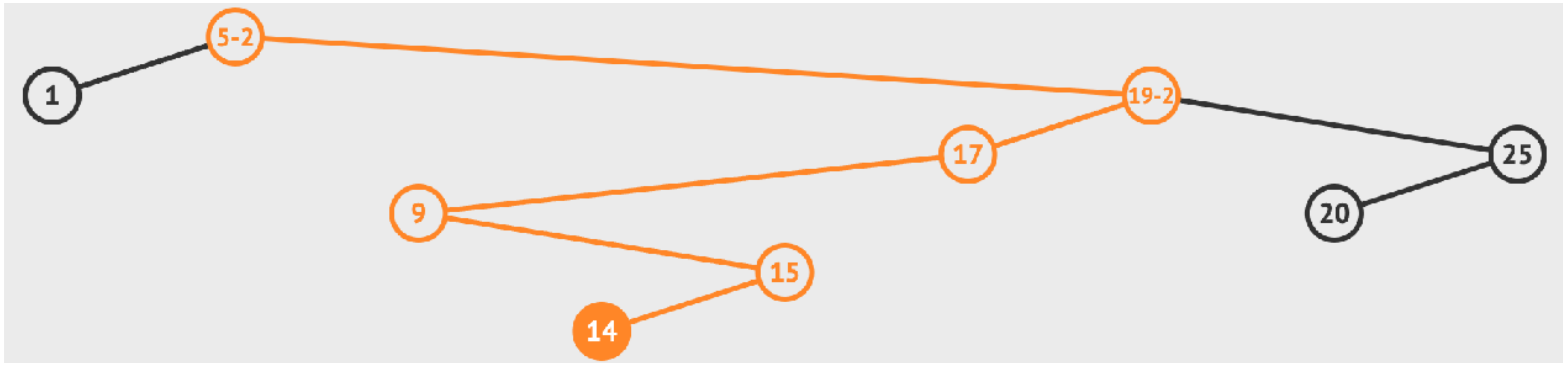
- Inserting the keys in the order A X C S E R H into an initially empty BST gives a worst-case tree where every node has one null link (one child), except one at the bottom that has two null links (it's a leaf). Give five other orderings of these keys that produce worst-case trees.

Problem 3

- Give five orderings of the keys A X C S E R H that when inserted into an initially empty binary search tree, produce best-case trees.

ANSWER 1

- Draw the BST that results when you insert the keys 5, 1, 19, 25, 17, 5, 19, 20, 9, 15, 14 in that order.
- -2 indicates that this node has been updated to the second value associated with that key.



ANSWER 2

- Inserting the keys in the order A X C S E R H into an initially empty BST gives a worst-case tree where every node has one null link (one child), except one at the bottom that has two null links (it's a leaf). Give five other orderings of these keys that produce worst-case trees.
- A C E H R S X
- X S R H E C A
- X A S C R E H
- X A S C R H E
- A X C S E H R

ANSWER 3

- Inserting the keys in the order A X C S E R H into an initially empty BST gives a worst-case tree where every node has one null link (one child), except one at the bottom that has two null links (it's a leaf). Give five other orderings of these keys that produce worst-case trees.
- H C S A E R X
- H C A E S R X
- H C E A S R X
- H S R X C A E
- H S X R C A E