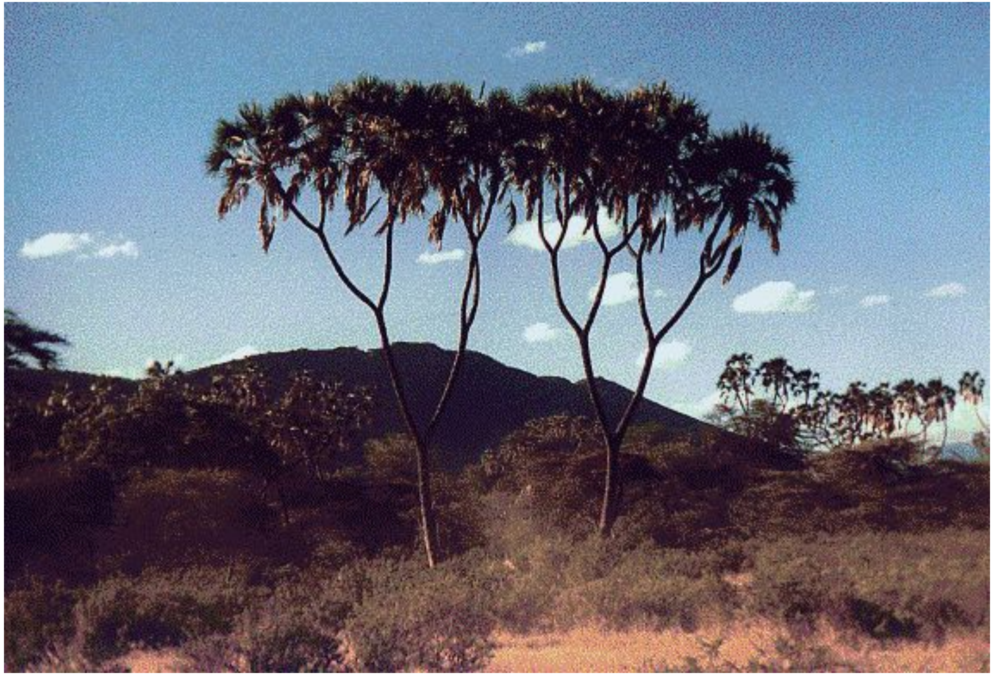
CS61B, 2019

Lecture 16: ADTs and BSTs

- Abstract Data Types
- Binary Search Tree (intro)
- BST Definitions
- BST Operations
- Sets vs. Maps, Summary

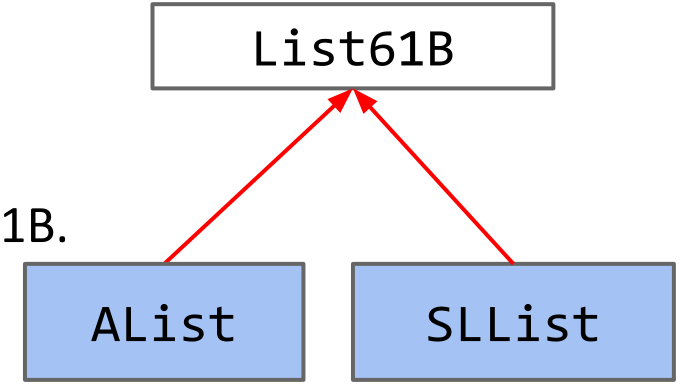


Abstract Data Types

Interfaces vs. Implementation

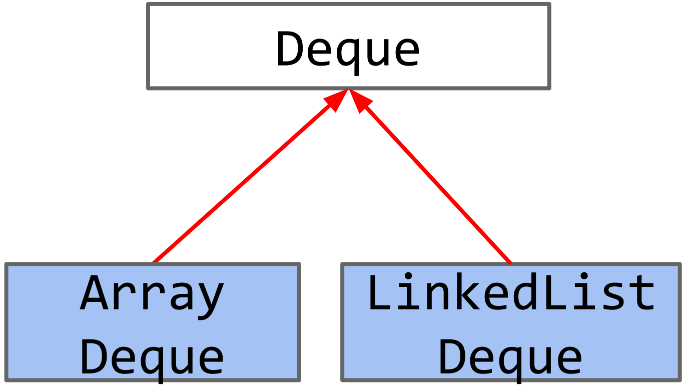
In class:

- Developed ALists and SLLists.
- Created an interface List61B.
    - Modified AList and SLList to implement List61B.
    - List61B provided default methods.



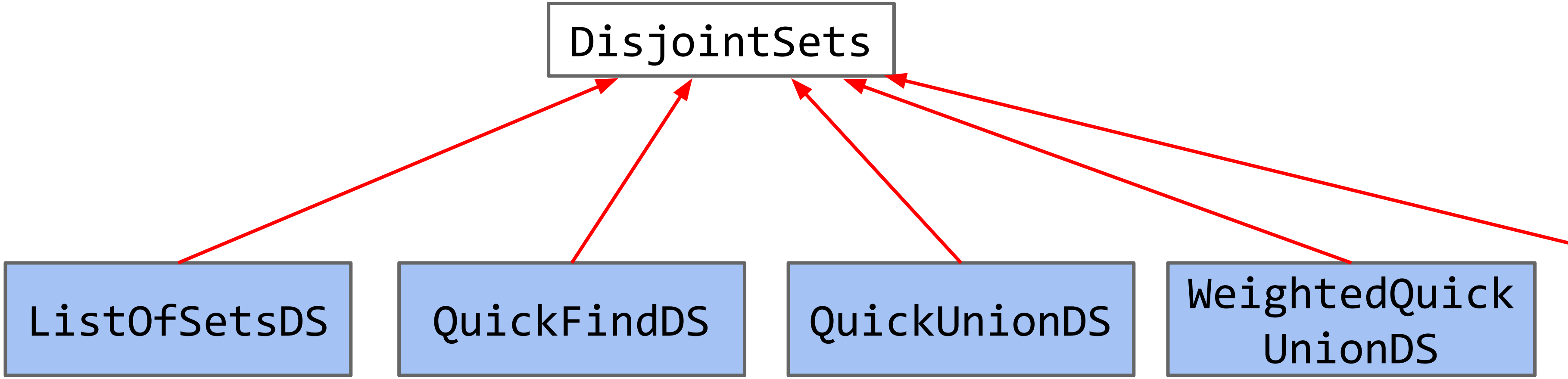
In projects:

- Developed ArrayDeque and LinkedListDeque.
- Created an interface Deque.
    - Modified AD and LLD to implement Deque.



Interfaces vs. Implementation

With DisjointSets, we saw a much richer set of possible implementations.

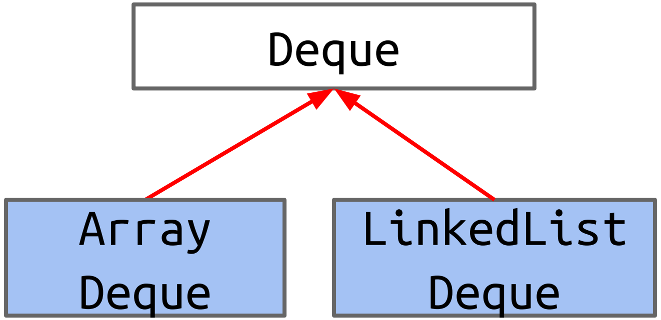


Abstract Data Types

An **Abstract Data Type (ADT)** is defined only by its operations, not by its implementation.

Deque ADT:

- `addFirst(Item x);`
- `addLast(Item x);`
- `boolean isEmpty();`
- `int size();`
- `printDeque();`
- `Item removeFirst();`
- `Item removeLast();`
- `Item get(int index);`

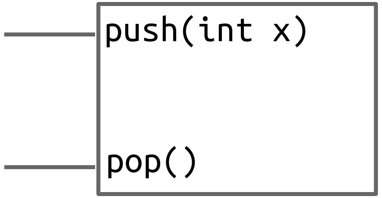


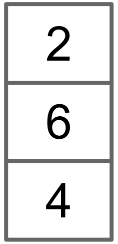
ArrayDeque and LinkedList Deque are implementations of the Deque ADT.

Another example of an ADT: The Stack

The Stack ADT supports the following operations:

- push(int x): Puts x on top of the stack.
- int pop(): Removes and returns the top item from the stack.





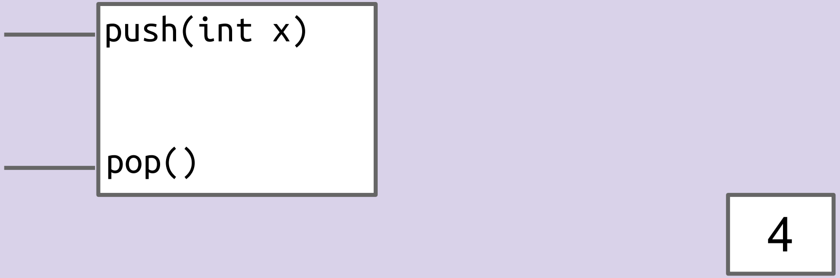
The Stack ADT: yellkey.com/?

The Stack <u>ADT</u> supports the following operations:

- push(int x): Puts x on top of the stack.
- int pop(): Removes and returns the top item from the stack.

Which <u>implementation</u> do you think would result in faster overall performance?

A. Linked List
B. Array



The Stack ADT: yellkey.com/?

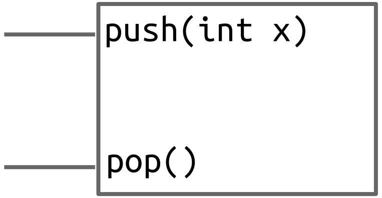
The Stack ADT supports the following operations:

- push(int x): Puts x on top of the stack.
- int pop(): Removes and returns the top item from the stack

Which implementation do you think would result in faster overall performance?

A. Linked List

B. Array



Both are about the same. No resizing for linked lists, so probably a lil faster.

The GrabBag ADT: yellkey.com/?

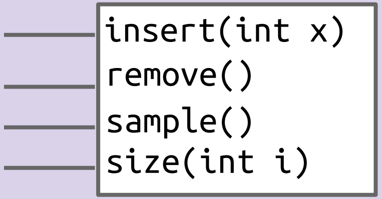
The GrabBag <u>ADT</u> supports the following operations:

- insert(int x): Inserts x into the grab bag.
- int remove(): Removes a random item from the bag.
- int sample(): Samples a random item from the bag (without removing!)
- int size(): Number of items in the bag.

Which <u>implementation</u> do you think would result in faster overall performance?

A. Linked List

B. Array



The GrabBag ADT: yellkey.com/?

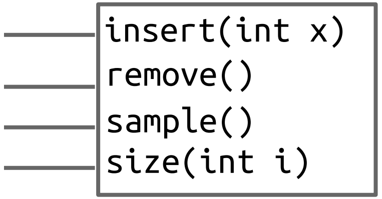
The GrabBag ADT supports the following operations:

- insert(int x): Inserts x into the grab bag.
- int remove(): Removes a random item from the bag.
- int sample(): Samples a random item from the bag (without removing!)
- int size(): Number of items in the bag.

Which implementation do you think would result in faster overall performance?

A. Linked List

B. Array

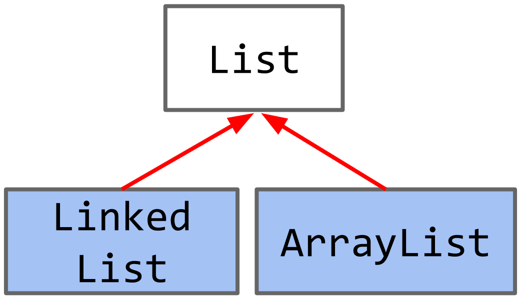


Abstract Data Types in Java

One thing I particularly like about Java is the syntax differentiation between abstract data types and implementations.

- Note: Interfaces in Java aren't purely abstract as they can contain some implementation details, e.g. default methods.

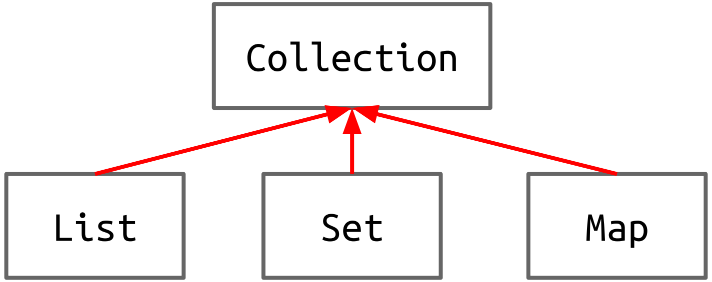
Example: `List<Integer> L = new ArrayList<>();`



Collections

Among the most important interfaces in the java.util library are those that extend the Collection interface (btw interfaces can extend other interfaces).

- Lists of things.
- Sets of things.
- Mappings between items, e.g. jhug's grade is 88.4, or Creature c's north neighbor is a Plip.
  - Maps also known as associative arrays, associative lists (in Lisp), symbol tables, dictionaries (in Python).



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:
    current_count = m.get(s, 0)
    m[s] = current_count + 1
```

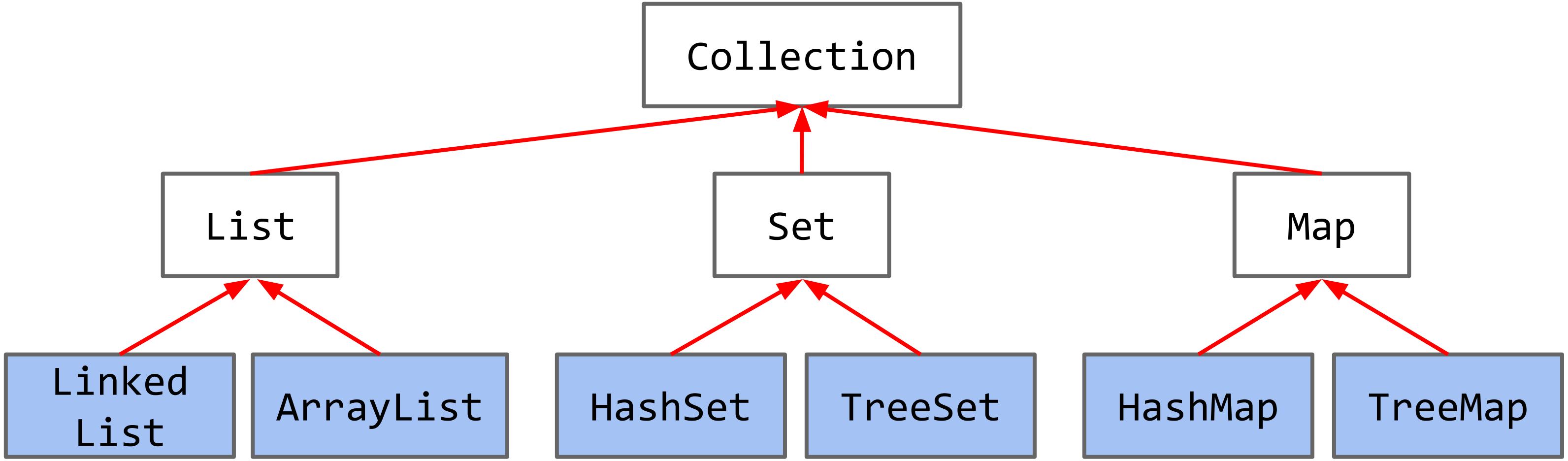
Python equivalent

Java Libraries

The built-in java.util package provides a number of useful:

- Interfaces: ADTs (lists, sets, maps, priority queues, etc.) and other stuff.
- Implementations: Concrete classes you can use.

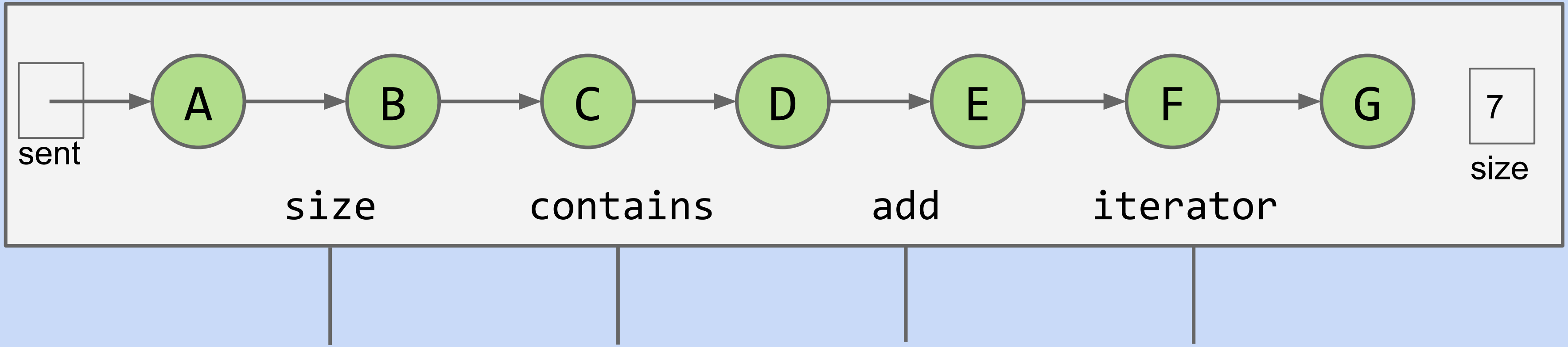
Today, we'll learn the basic ideas behind the TreeSet and TreeMap.



Binary Search Trees

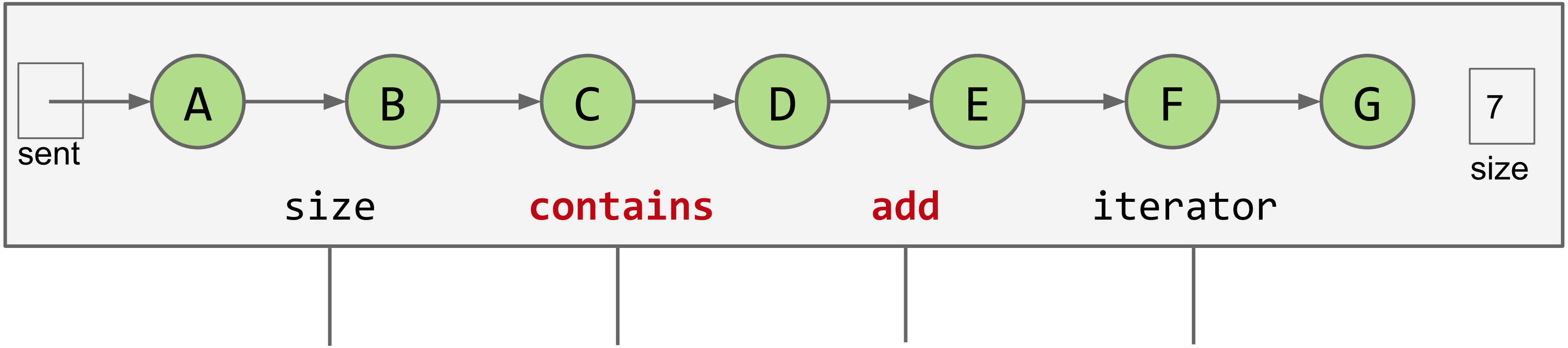
Analysis of an OrderedLinkedListSet<Character>

In an earlier lecture, we implemented a set based on unordered arrays. For the ***order linked list*** set implementation below, name an operation that takes worst case linear time, i.e. Θ(N).



Analysis of an OrderedLinkedListSet<Character>

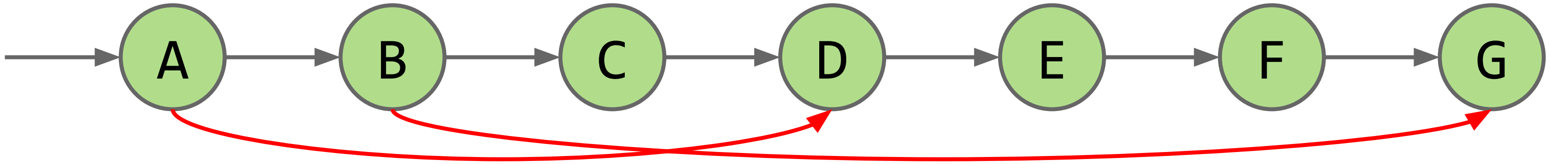
In an earlier lecture, we implemented a set based on [unordered arrays](). For the ***order linked list*** set implementation below, name an operation that takes worst case linear time, i.e. Θ(N).



Optimization: Extra Links

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

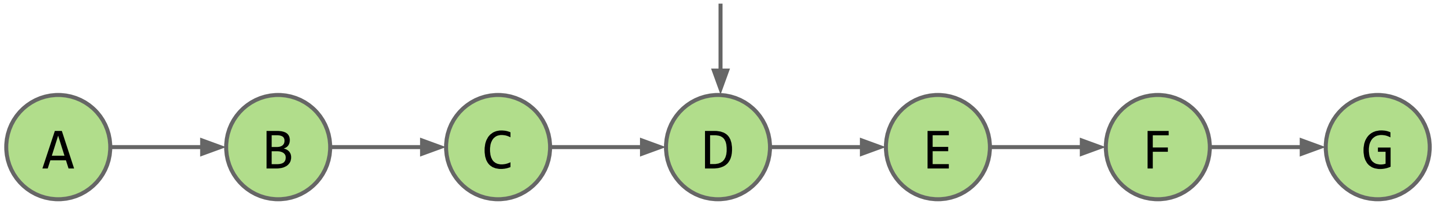
- Add (random) express lanes. Skip List (won't discuss in 61B)



Optimization: Change the Entry Point

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

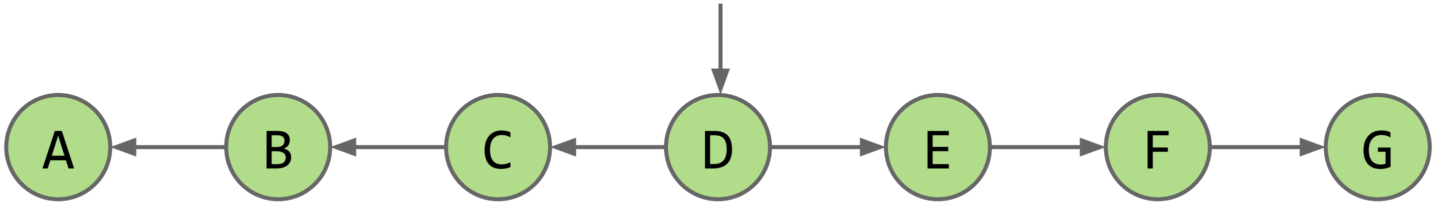
- Move pointer to middle.



Optimization: Change the Entry Point, Flip Links

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

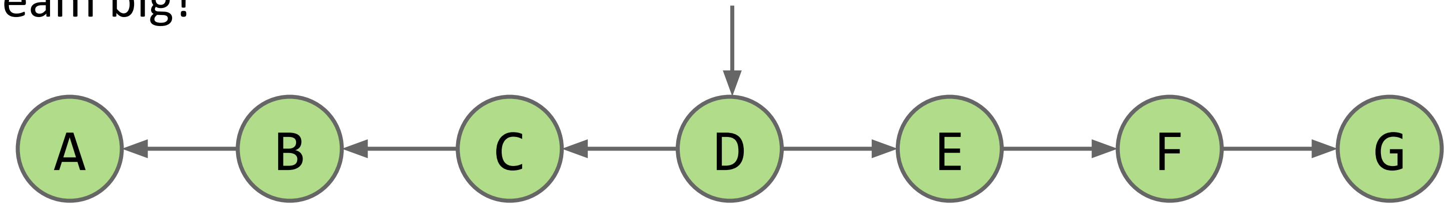
- Move pointer to middle and flip left links. Halved search time!



Optimization: Change the Entry Point, Flip Links

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

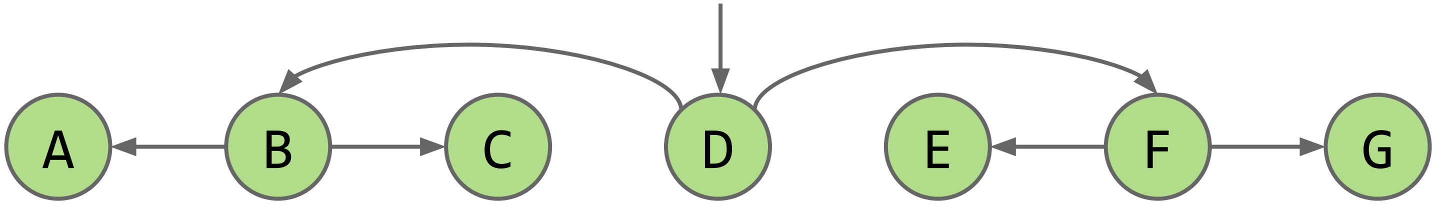
- How do we do even better?
- Dream big!

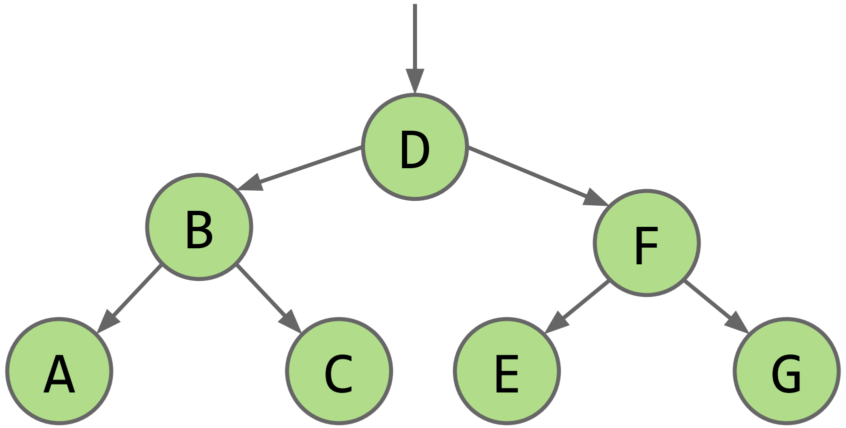


Optimization: Change Entry Point, Flip Links, Allow Big Jumps

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

- How do we do better?





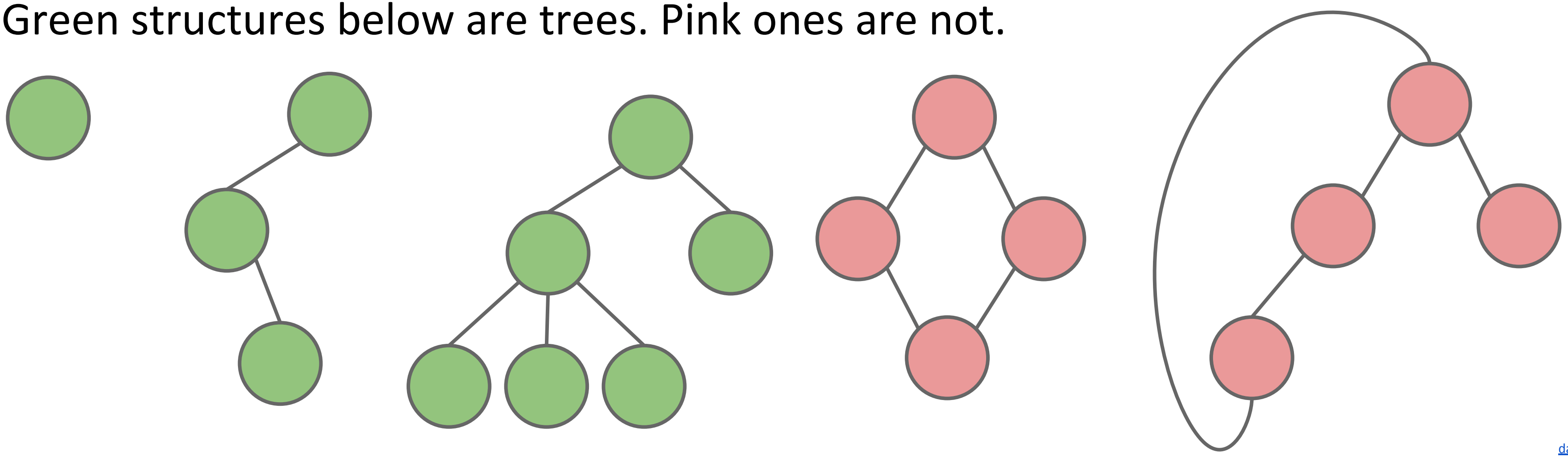
BST Definitions

Tree

A tree consists of:

- A set of nodes.
- A set of edges that connect those nodes.
  - Constraint: There is exactly one path between any two nodes.

Green structures below are trees. Pink ones are not.

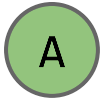


Rooted Trees and Rooted Binary Trees

In a rooted tree, we call one node the root.

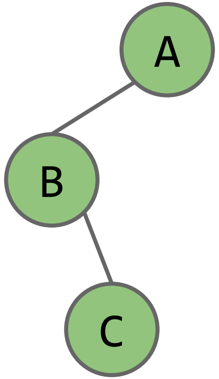
- Every node N except the root has exactly one parent, defined as the first node on the path from N to the root.
- Unlike (most) real trees, the root is usually depicted at the top of the tree.
- A node with no child is called a leaf.

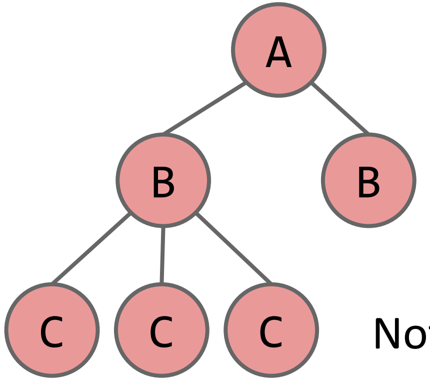
In a rooted binary tree, every node has either 0, 1, or 2 children (subtrees).



For each of these:

- A is the root.
- B is a child of A. (and C of B)
- A is a parent of B. (and B of C)





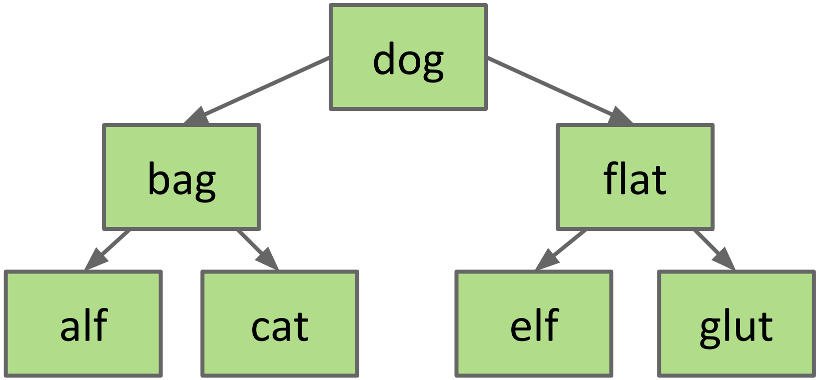
Not binary!

Binary Search Trees

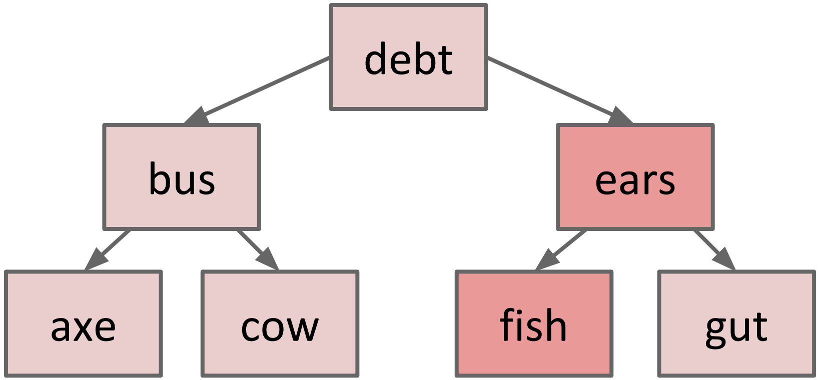
A binary search tree is a rooted binary tree with the BST property.

BST Property. 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.



Binary Search Tree



Binary Tree, but not a Binary Search Tree

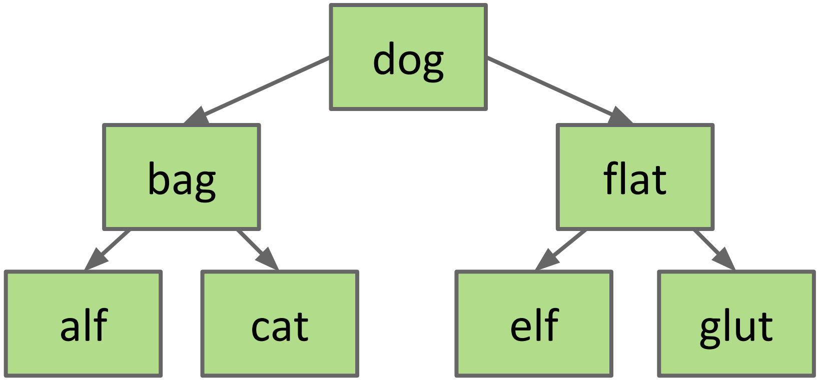
Binary Search Trees

Ordering must be complete, transitive, and antisymmetric. Given keys p and q:

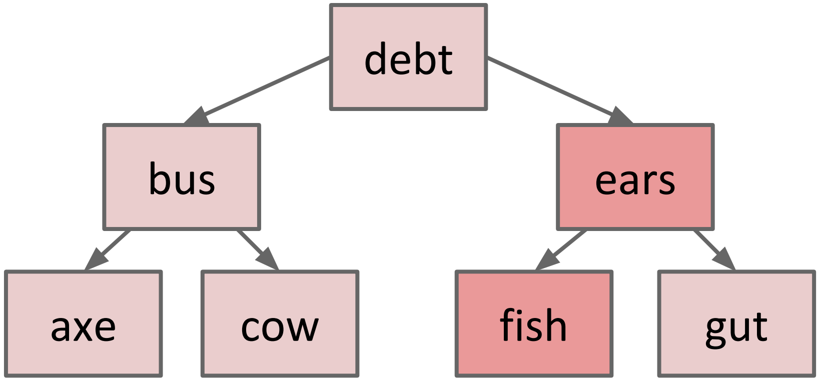
- Exactly one of p ≺ q and q ≺ p are true.
- p ≺ q and q ≺ r imply p ≺ r.

One consequence of these rules: No duplicate keys allowed!

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



Binary Search Tree



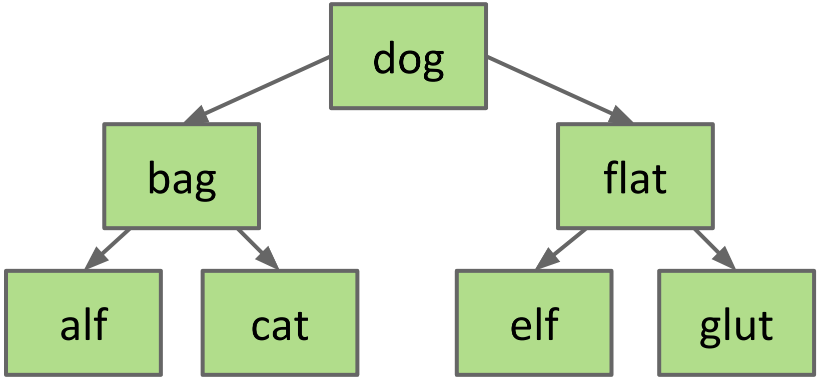
Binary Tree, but not a Binary Search Tree

BST Operations: Search

Finding a searchKey in a BST (come back to this for the BST lab)

If searchKey equals T.key, return.

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

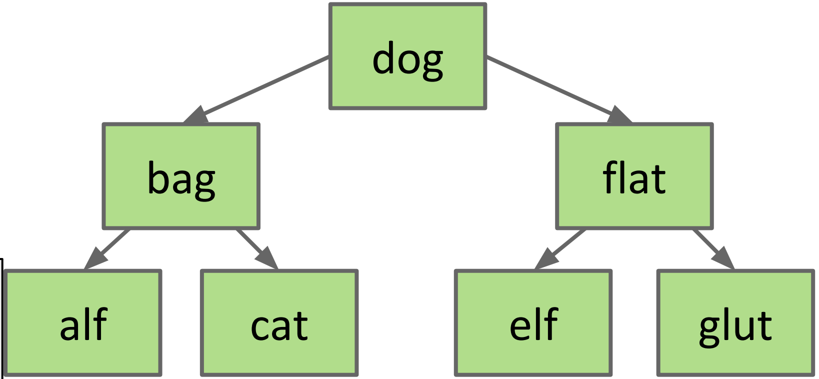


Finding a searchKey in a BST

If searchKey equals T.key, return.

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

```
static BST find(BST T, Key sk) {
   if (T == null)
      return null;
   if (sk.equals(T.key))
      return T;
   else if (sk < T.key)
      return find(T.left, sk);
   else
      return find(T.right, sk);
}
```

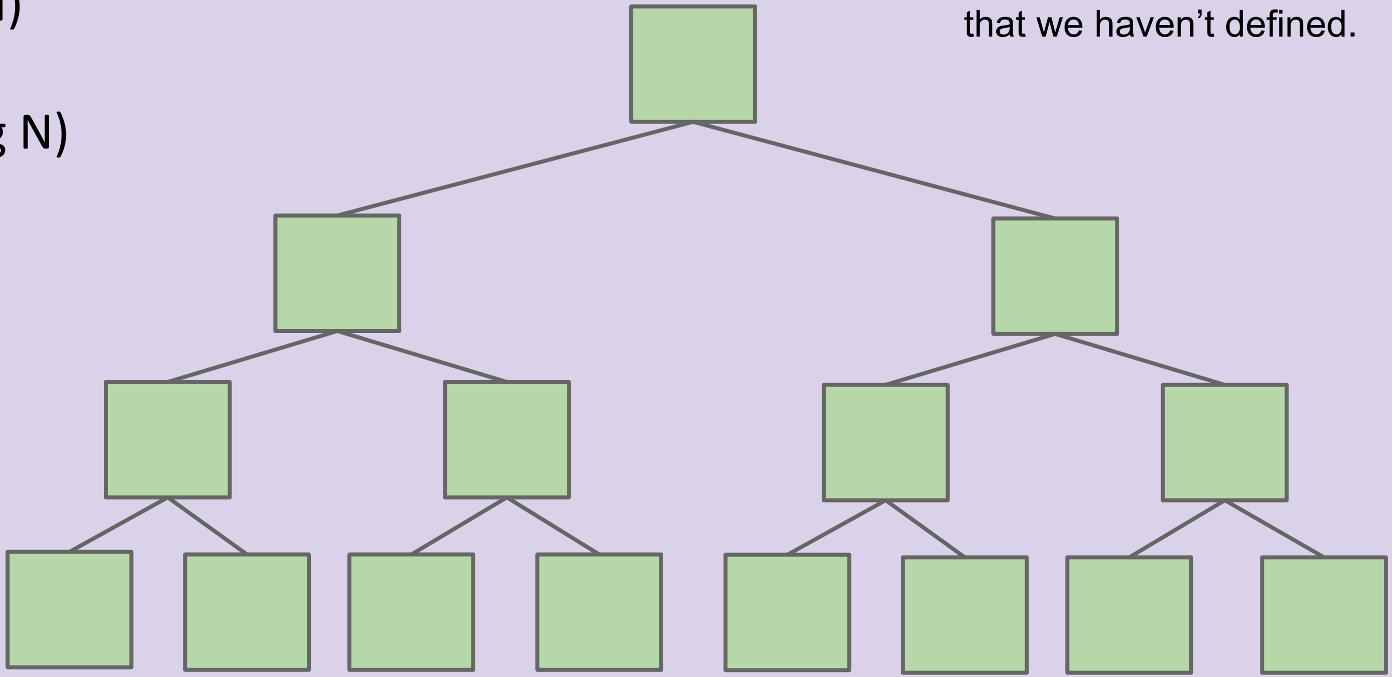


BST Search: http://yellkey.com/?

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

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

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

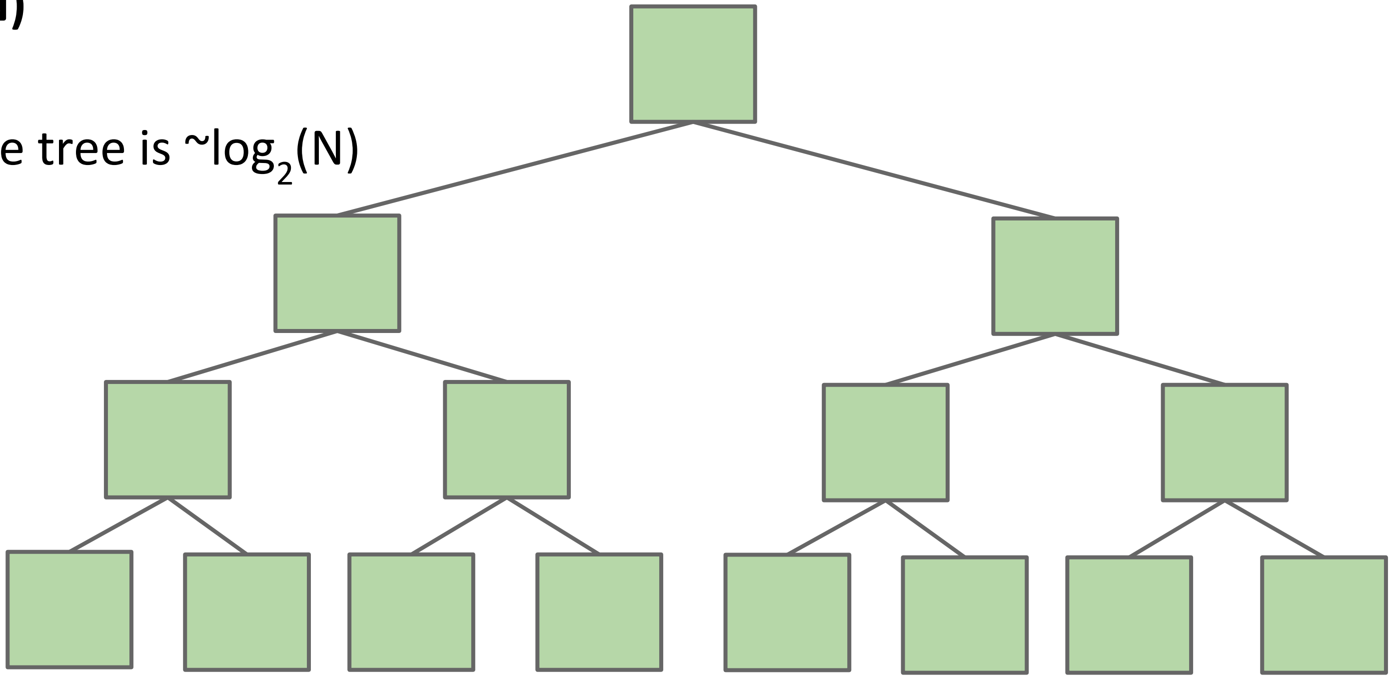


BST Search

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 ~$\log_2(N)$



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.

BST Operations: Insert

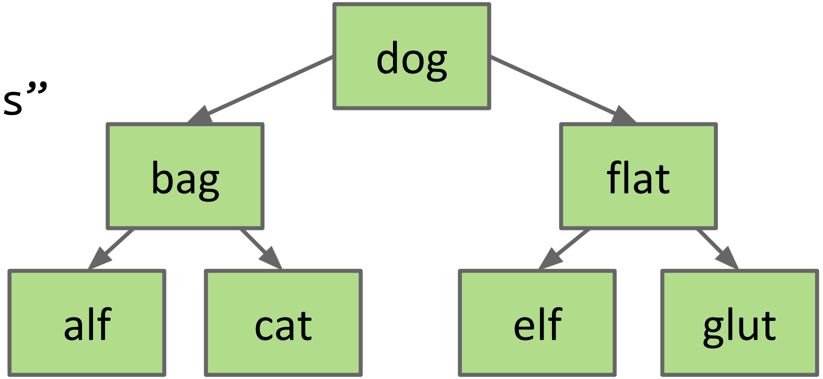
Inserting a New Key into a BST

Search for key.

- If found, do nothing.
- If not found:
  - Create new node.
  - Set appropriate link.

Example:

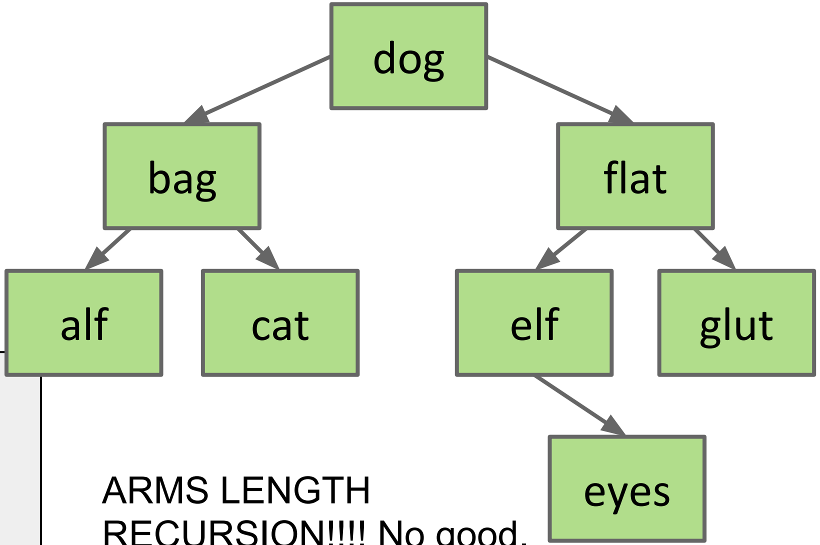
`insert “eyes”`



Inserting a New Key into a BST

Search for key.

- If found, do nothing.
- If not found:
  - Create new node.
  - Set appropriate link.




```
static BST insert(BST T, Key ik) {
  if (T == null)
    return new BST(ik);
  if (ik < T.key)
    T.left = insert(T.left, ik);
  else if (ik > T.key)
    T.right = insert(T.right, ik);
  return T;
}
```

ARMS LENGTH RECURSION!!!! No good.

A common rookie bad habit to avoid:

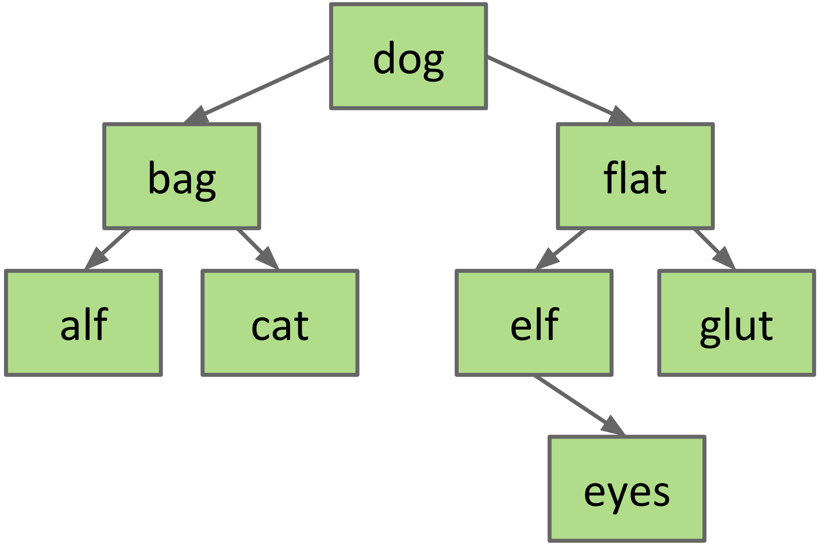
```
if (T.left == null)
  T.left = new BST(ik);
else if (T.right == null)
  T.right = new BST(ik);
```

BST Operations: Delete

Deleting from a BST

3 Cases:

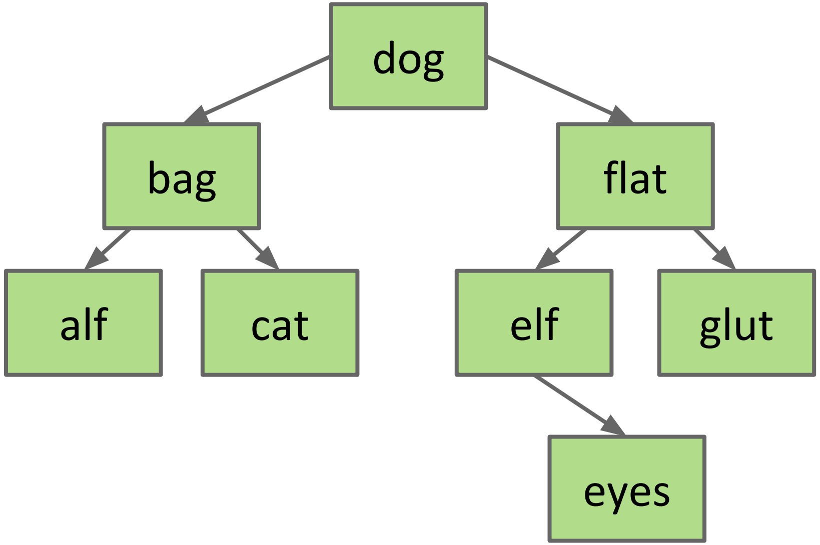
- Deletion key has no children.
- Deletion key has one child.
- Deletion key has two children.



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

Deletion key has no children ("glut"):

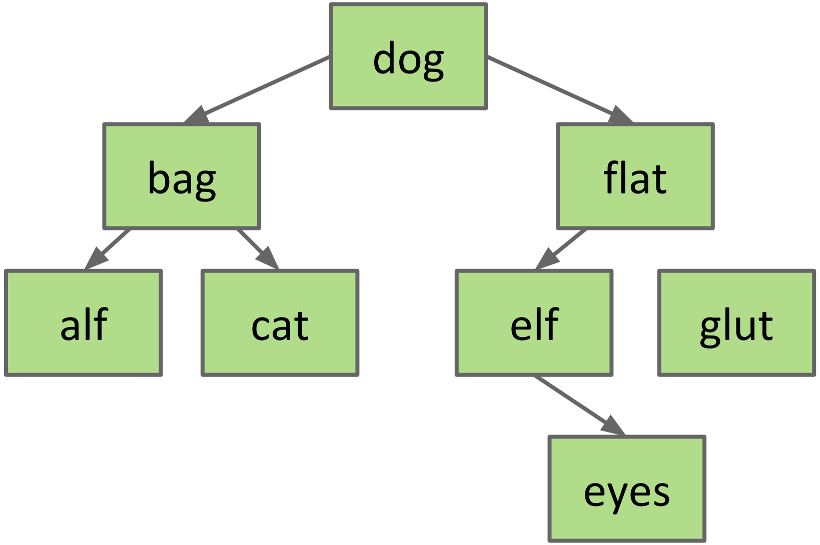
- Just sever the parent's link.
- What happens to "glut" node?



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

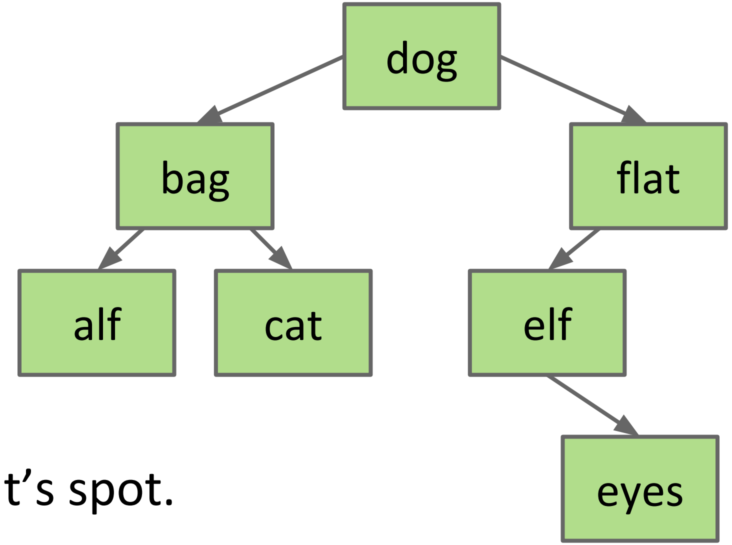
Deletion key has no children ("glut"):

- 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"):



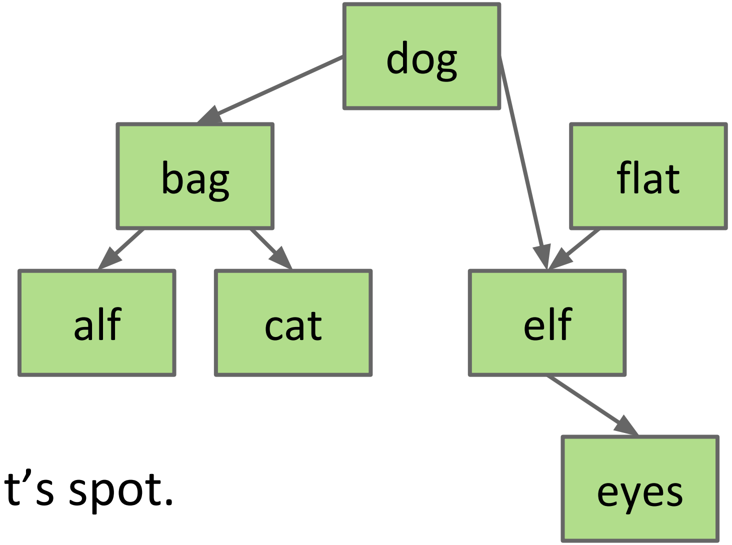
Goal:

- Maintain BST property.
- Flat's child definitely larger than dog.
  - Safe to just move that child into flat's spot.

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

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

Example: delete("flat"):



Goal:

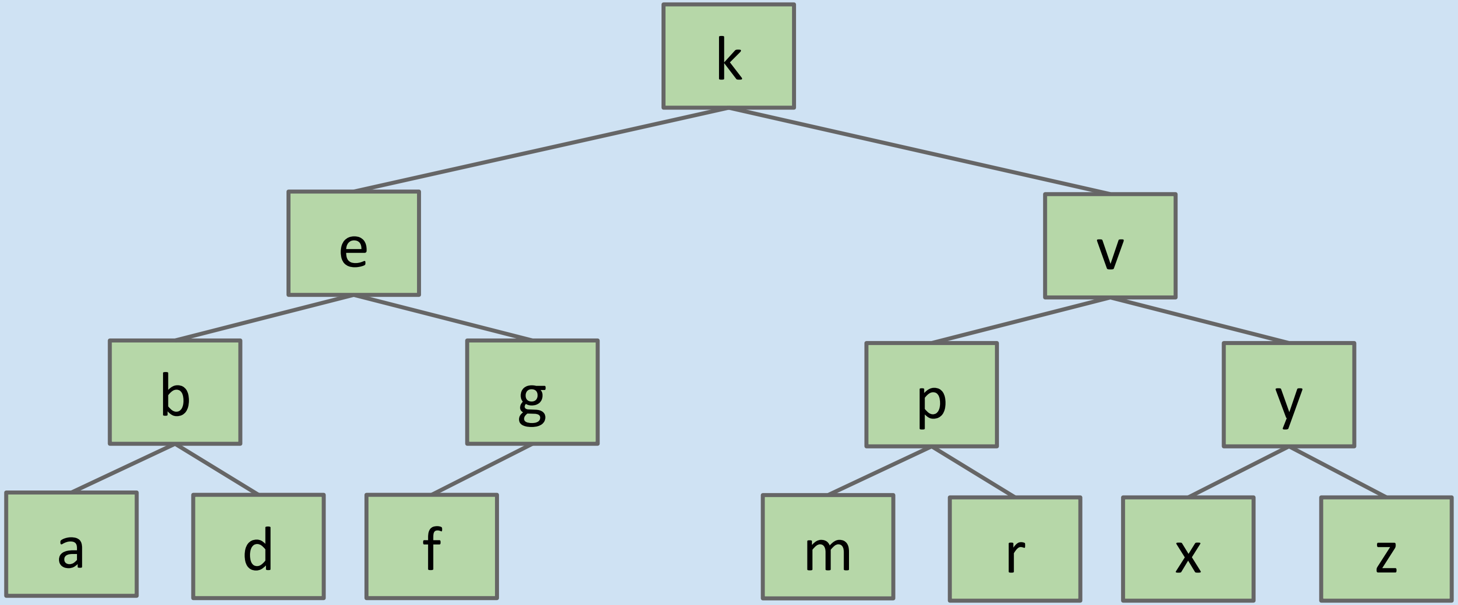
- Maintain BST property.
- Flat's child definitely larger than dog.
  - Safe to just move that child into flat's spot.

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

- Flat will be garbage collected (along with its instance variables).

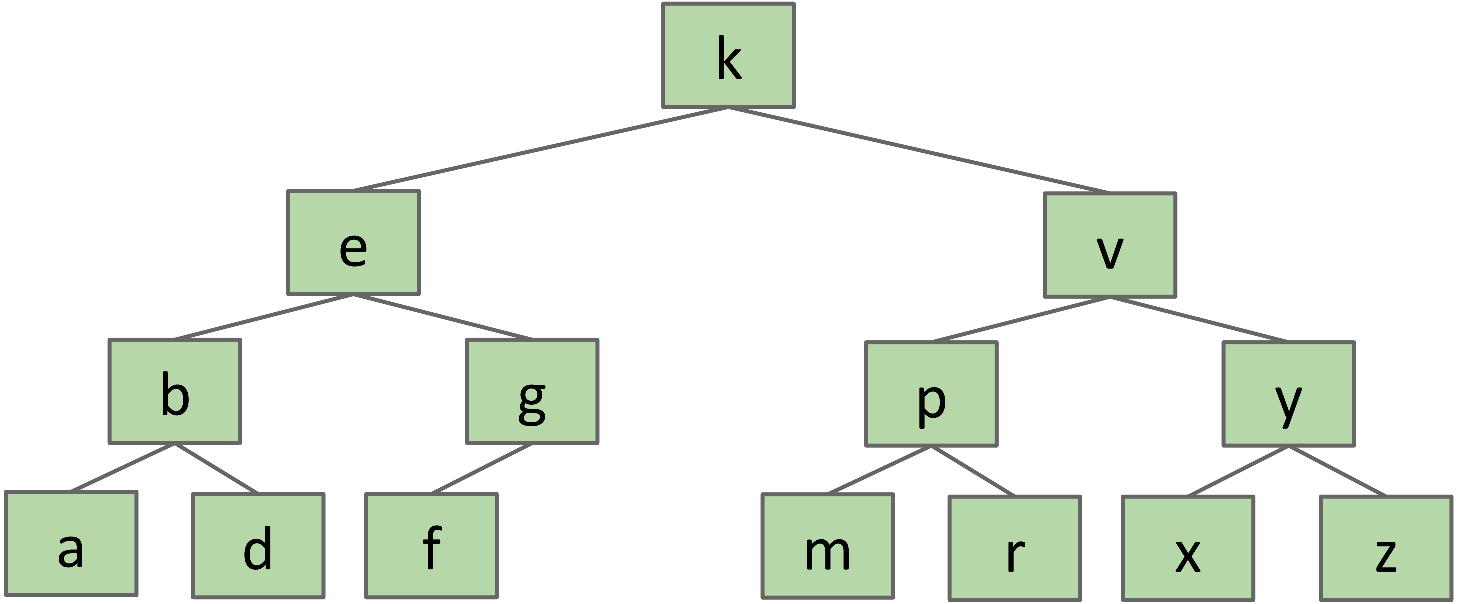
Hard Challenge

Delete k.



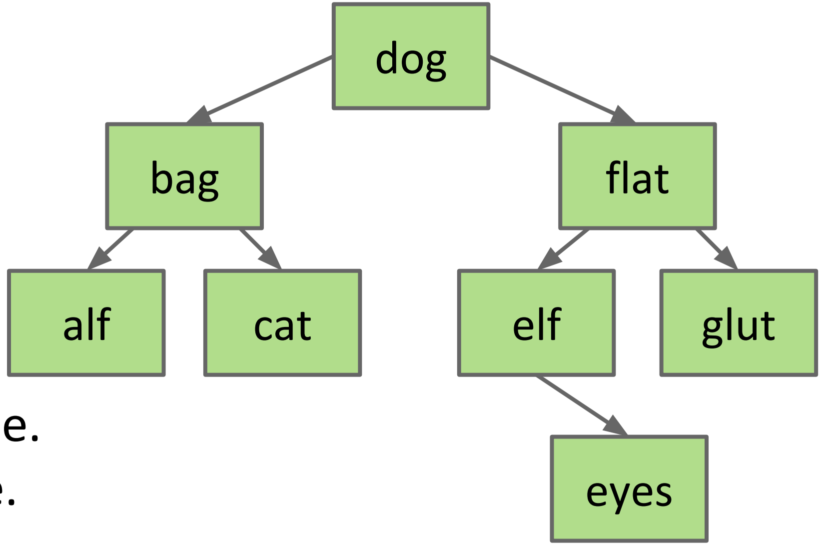
Hard Challenge

Delete k.



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

Example: delete(“dog”)



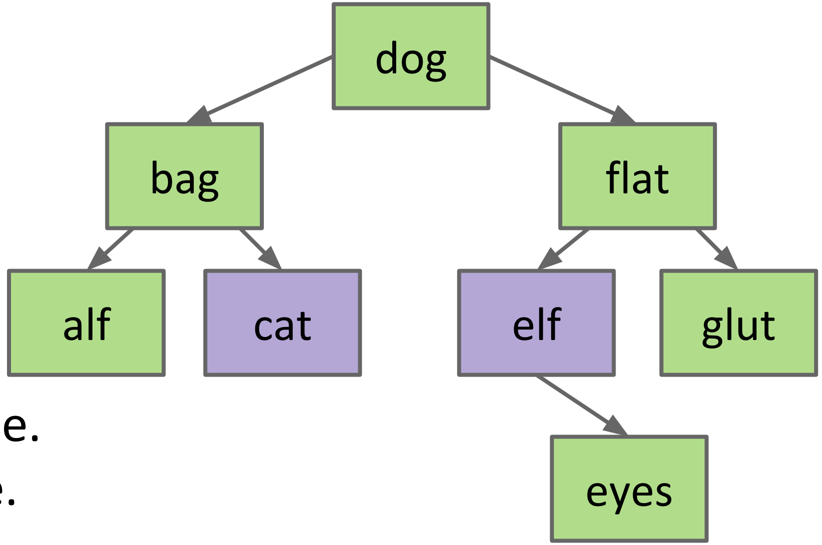
Goal:

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

Would bag work?

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

Example: delete(“dog”)



Goal:

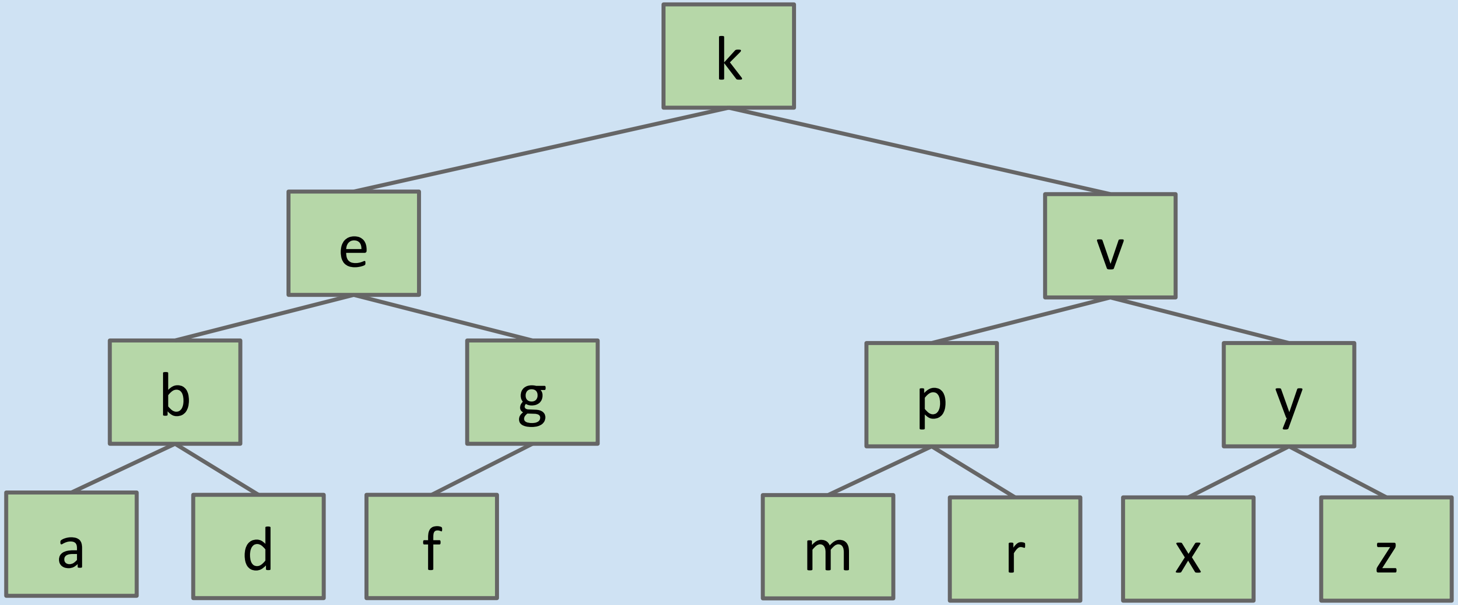
- Find a new root node.
- Must be > than everything in left subtree.
- Must be < than everything right subtree.

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

- Delete “cat” or “elf”, and stick new copy in the root position:
  - This deletion guaranteed to be either case 1 or 2. Why?
- This strategy is sometimes known as “Hibbard deletion”.

Hard Challenge (Hopefully Now Easy)

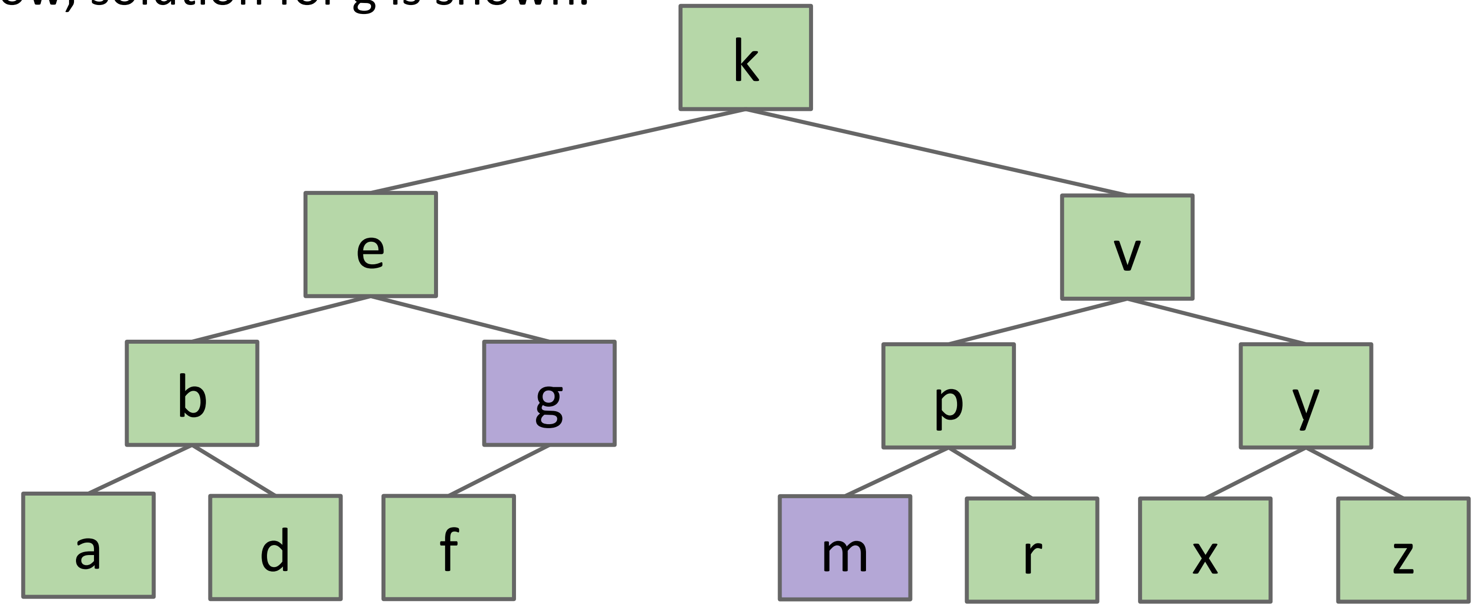
Delete k.



Hard Challenge (Hopefully Now Easy)

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

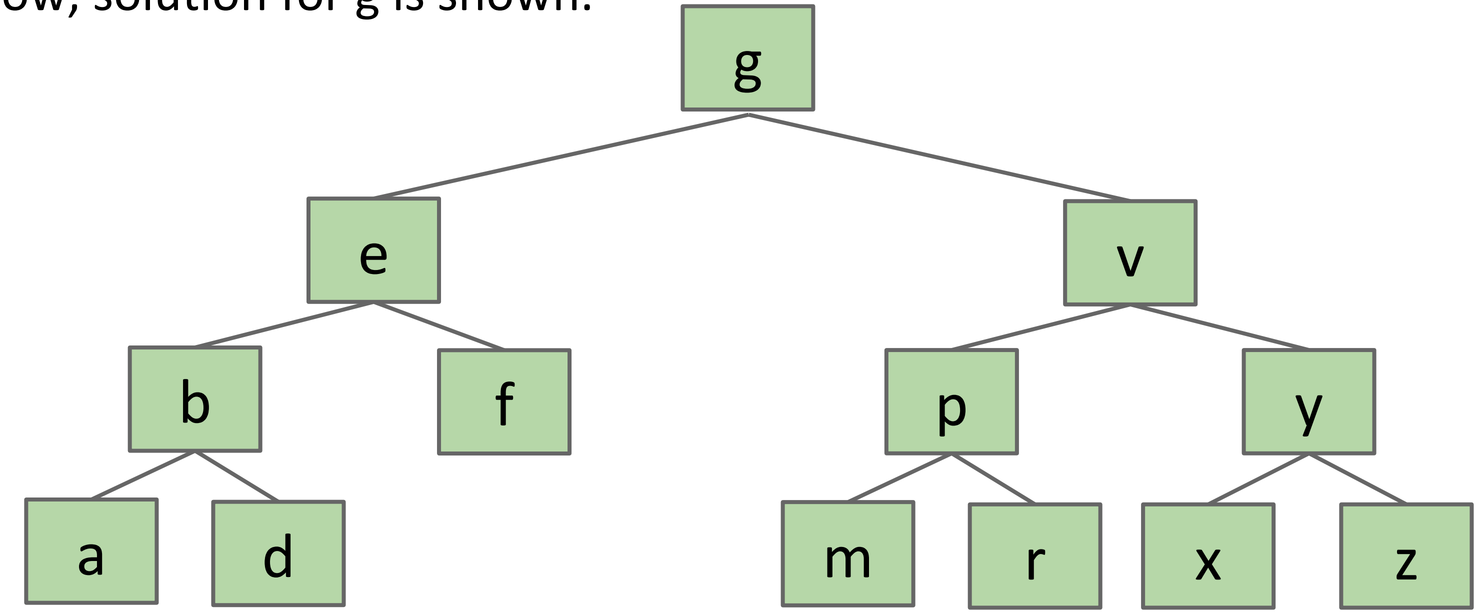
- Below, solution for g is shown.



Hard Challenge (Hopefully Now Easy)

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

- Below, solution for g is shown.

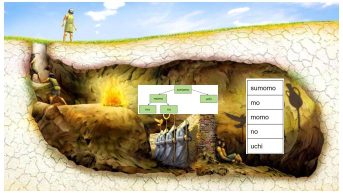


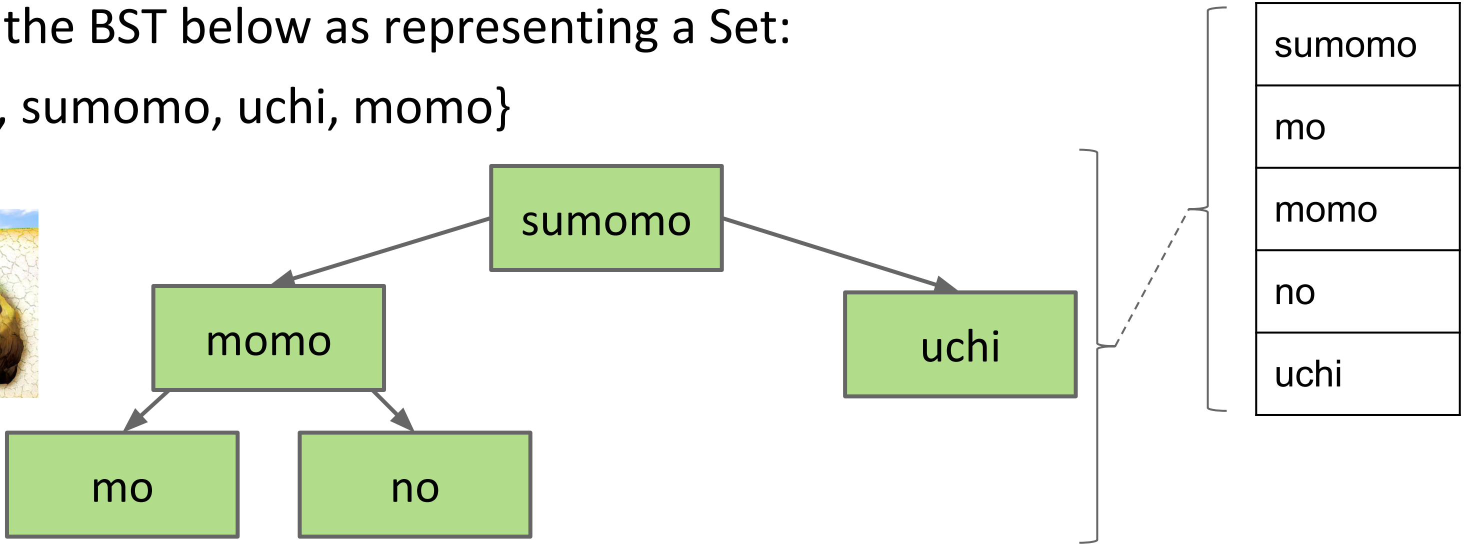
Sets vs. Maps, Summary

Sets vs. Maps

Can think of the BST below as representing a Set:

- {mo, no, sumomo, uchi, momo}

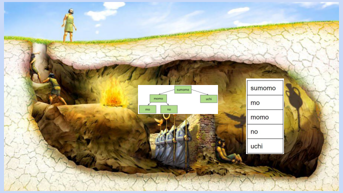


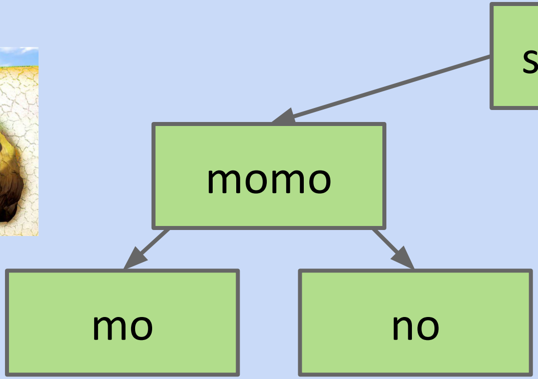


Sets vs. Maps

Can think of the BST below as representing a Set:

- {mo, no, sumomo, uchi, momo}





sumomo
mo
momo
no
uchi

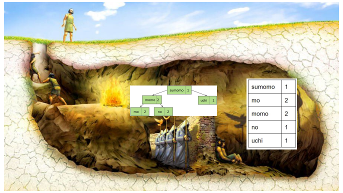
But what if we wanted to represent a mapping of word counts?

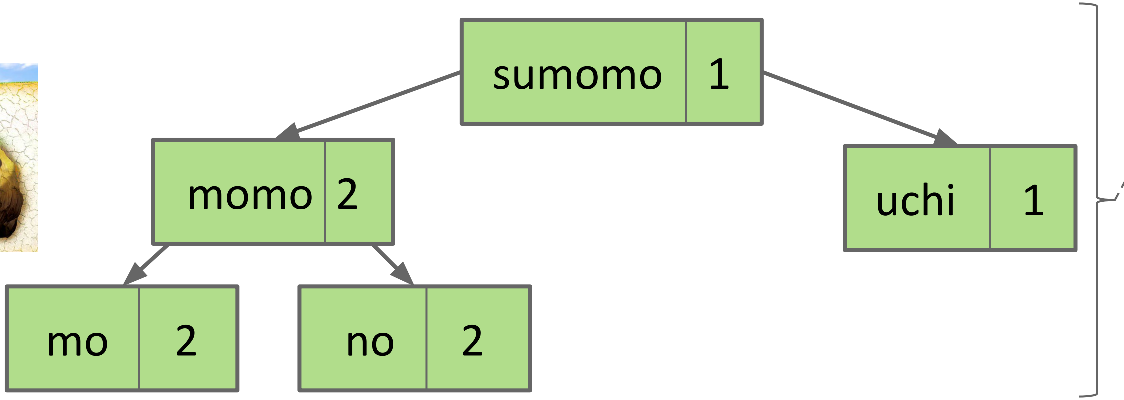
????

sumomo	1
mo	2
momo	2
no	1
uchi	1

Sets vs. Maps

To represent maps, just have each BST node store key/value pairs.





sumomo	1
mo	2
momo	2
no	1
uchi	1

Note: No efficient way to look up by value.

- Example: Cannot find all the keys with value = 1 without iterating over ALL nodes. This is fine.

Summary

Abstract data types (ADTs) are defined in terms of operations, not implementation.

Several useful ADTs: Disjoint Sets, Map, Set, List.

- Java provides Map, Set, List interfaces, along with several implementations.

We've seen two ways to implement a Set (or Map): ArraySet and using a BST.

- ArraySet: $\Theta(N)$ operations in the worst case.
- BST: $\Theta(\log N)$ operations if tree is balanced.

BST Implementations:

- Search and insert are straightforward (but insert is a little tricky).
- Deletion is more challenging. Typical approach is "Hibbard deletion".

BST Implementation Tips

Tips for BST Lab

- Code from class was "naked recursion". Your BSTMap will not be.
- For each public method, e.g. `put(K key, V value)`, create a private recursive method, e.g. `put(K key, V value, Node n)`
- When inserting, always set left/right pointers, even if nothing is actually changing.
- Avoid "arms length base cases". Don't check if left or right is null!

```
static BST insert(BST T, Key ik) {
  if (T == null)
    return new BST(ik);
  if (ik ≺ T.label()))
    T.left = insert(T.left, ik);
  else if (ik ≻ T.label())
    T.right = insert(T.right, ik);
  return T;
}
```

Always set, even if nothing changes!

Avoid "arms length base cases".

```
  if (T.left == null)
    T.left = new BST(ik);
  else if (T.right == null)
    T.right = new BST(ik);
```

Citations

Probably photoshopped binary tree: http://cs.au.dk/~danvy/binaries.html