Ordered Dictionaries via Binary Search Trees

CMPUT 275 - Winter 2018

University of Alberta

Dictionaries / Maps

Recall a dictionary in Python associates items to keys. It maintains a collection of pairs

```
(key, item)
```

where the key values are unique.

It supports **indexing** by keys.

```
>>> msg = {"arrive":"hello", "depart":"goodbye" }
>>> print(msg["arrive"])
hello
```

Dictionaries / Maps

The underlying data structure supporting this is a **hash table**.
`https://github.com/python/cpython`

While there are no certain guarantees on the performance, we typically observe dictionary lookups/updates to run in $O(1)$ time[1].

In `C++`, the standard template library has these as the `unordered_map` type.

Major Drawback (in some cases)
No ordering of the entries is maintained.

[1] Well, really $O()$ of the hash function run time, like $O(len(str))$ for strings.

Dictionaries / Maps

This topic: **Ordered Dictionaries / Maps**

We can store items in a way that maintains the ordering of their keys.

This is already present in `C++` as simply the `map` type. Let's build one in Python.

Our Goal

We want to maintain a dictionary that allows us to answer the following queries quickly:

- How many keys are $\leq$ some given key?
- What is the i'th key (in terms of their sorted order via $\leq$)?
- What is the smallest key in the dictionary that is $>$ some given key? etc.

We still need to be able to add/update/remove entries quickly too.

End Result: We will discuss and implement a dictionary that supports all of these operations in $O(\log n)$ time[2]!

[2]Minor detail: using $O(\log n)$ comparisons, which may take more than constant time for some data types like string comparison.

Prerequisite

To store items in a hash table, we needed a hash function.

This is no longer needed for our keys, but we need to be able to compare them via <, or using the `__lt__` method if the keys are some custom class.

The < operator must define a total ordering of the keys.

Suppose x, y, z are keys. The following must hold for the concept of an ordering to even make sense.

- **Antisymmetry**: Exacly one of $x < y$ or $y < x$ holds if x and y are not equal.
- **Transitivity**: If $x < y$ and $y < z$ then $x < z$.

Total Ordering Examples

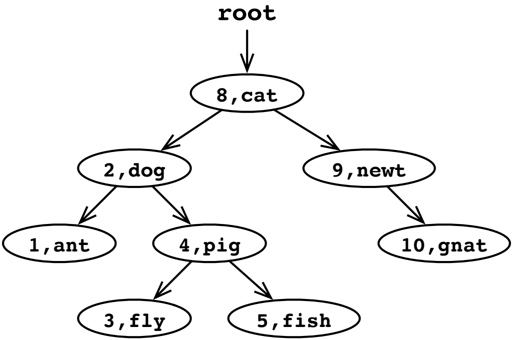
Common

- Normal ordering of numbers.
- Lexicographic ordering of tuples or strings.

A Weird One

- For a tuple of integers $(s_1, \ldots, s_k)$, order first by $\sum_i s_i$ and break ties lexicographically.
  Used by some algebraic algorithms that generalize the notion of greatest common divisors to multivariate polynomials.

Underlying Representation: Binary Search Tree

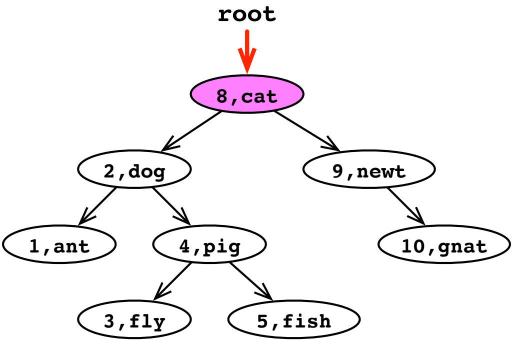


A binary search tree is a rooted tree with the following properties.

- Each node has ≤ 2 children, labelled `left` and `right`.
- For each node v, any node in the left subtree has a smaller key than v and any node in the right subtree has a greater key than v.

Underlying Representation: Binary Search Tree

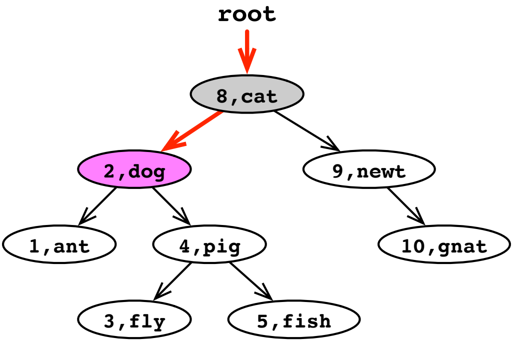
To find a node with a given key, start at the root and chase down the appropriate subtree by comparing the key to find with the key of the current node.



Picture: Finding the node with key 3.

Underlying Representation: Binary Search Tree

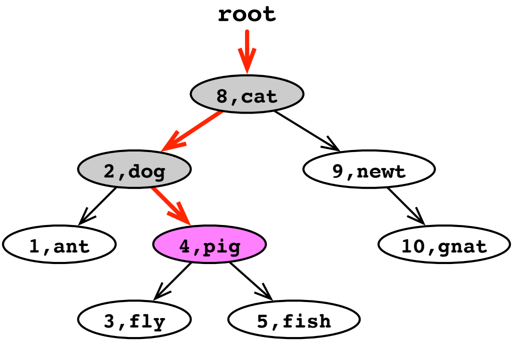
To find a node with a given key, start at the root and chase down the appropriate subtree by comparing the key to find with the key of the current node.



Picture: Finding the node with key 3.

Underlying Representation: Binary Search Tree

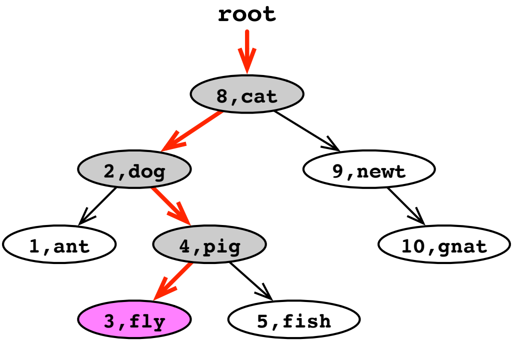
To find a node with a given key, start at the root and chase down the appropriate subtree by comparing the key to find with the key of the current node.



Picture: Finding the node with key 3.

Underlying Representation: Binary Search Tree

To find a node with a given key, start at the root and chase down the appropriate subtree by comparing the key to find with the key of the current node.



Picture: Finding the node with key 3.

Algorithm 1 Finding a node with key x.

```
node ← root
while node ≠ null do
    if x == node.key then
        return node
    else if x < node.key then
        node ← node.left
    else
        node ← node.right
return null                                  # no node has this key
```

Running Time: $O(\textit{height})$.

Later, we will ensure the height is $O(\log n)$.

Updating

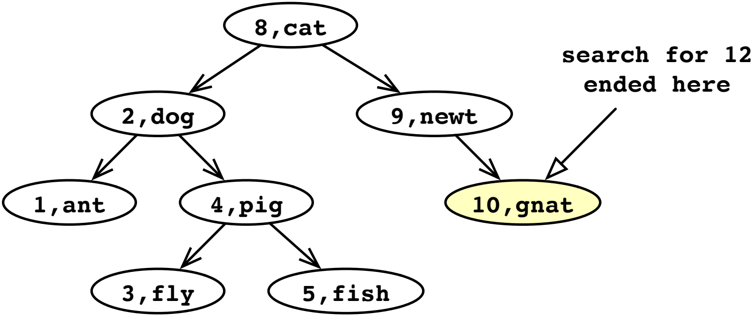
Suppose we want to update a key k to store item x.

If k is already in the tree then just update its associated item to x (no tree restructuring).

Otherwise, we have to create a new node. But where?

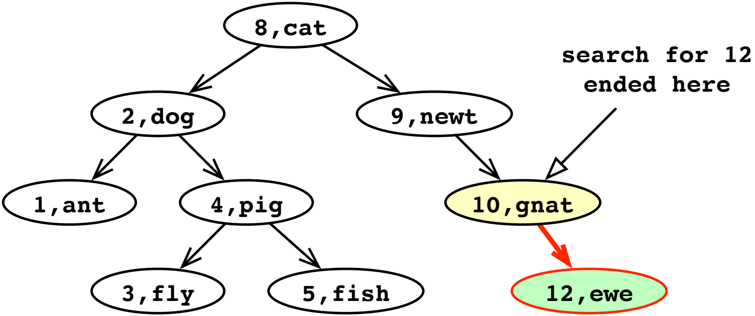
Idea: Just under the last node we saw in the search for the key.

Updating



Picture: Adding ewe with key 12.

Updating



Picture: Adding ewe with key 12.

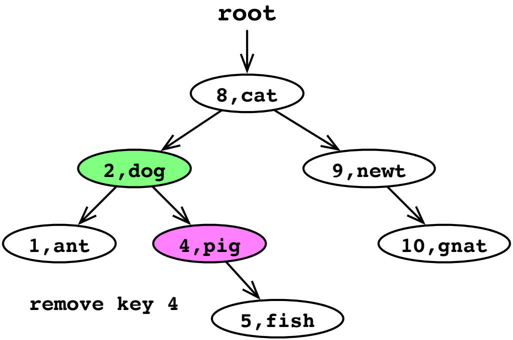
Algorithm 2 Updating key k with item x.

```
if root is null then
    root ← new node with k, x and null children
    return
node, parent ← root, null
while node ≠ null and key ≠ node.key do
    if k < node.key then
        node, parent ← node.left, node
    else
        node, parent ← node.right, node
if node is not null then
    update the item at node to x                    # just replace the item
else if k < parent.key then
    parent.left ← new node with k, x and null children
else
    parent.right ← new node with k, x and null children
```

Removing a Key

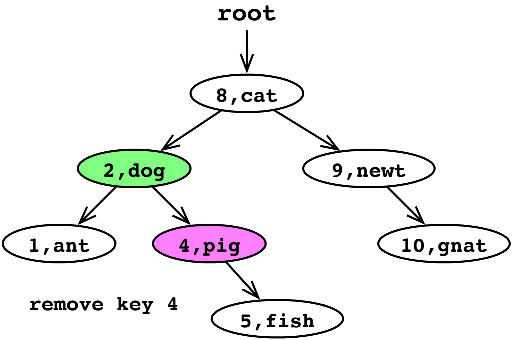
Suppose we want to remove an entry corresponding to a certain key (similar to `dict.remove`).

We already know how to find the the key and its parent.



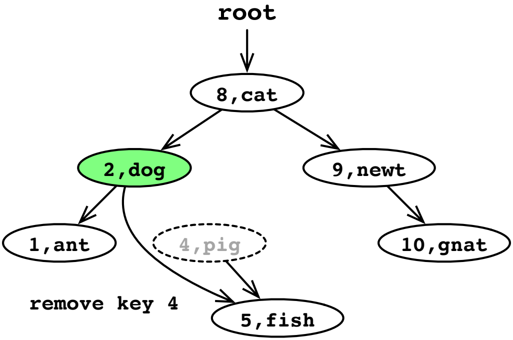
Removing a Key

Easy Case: *node.left* is `null`



Removing a Key

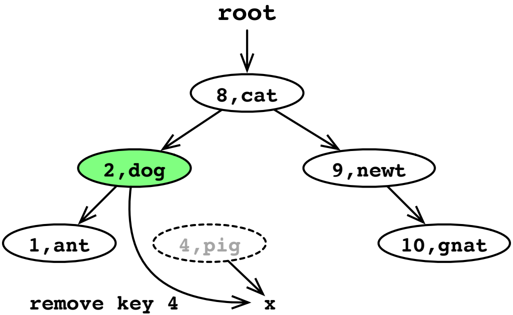
Just replace the appropriate child pointer of the parent to the right child of the node to delete.



Easy to check BST properties still hold.

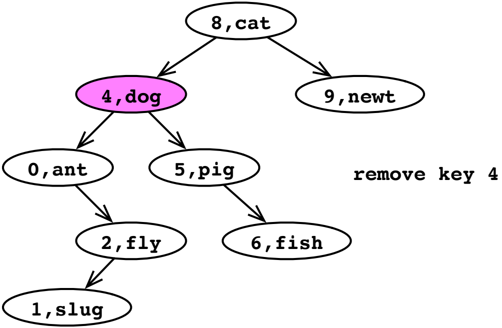
Removing a Key

Works even if the right child of the node to be deleted was also `null` (i.e. the node to delete was a leaf).



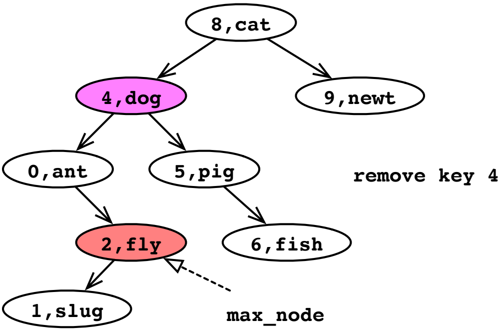
Removing a Key

Slightly Harder Case: *node.left* is not `null`



Removing a Key

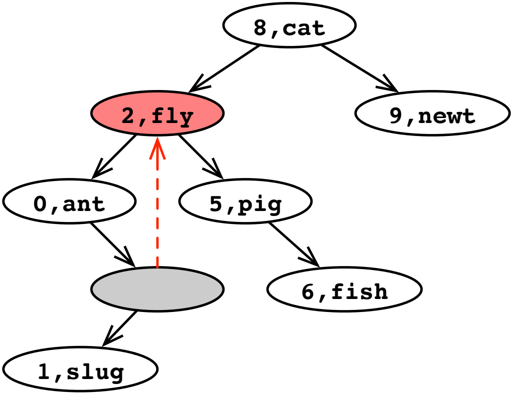
Find the maximum-key node in the subtree rooted at *node.left*, call this *max_node*.



Start at *node.left* and then follow the *.right* pointer until you hit `null`

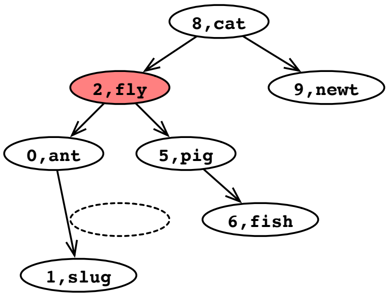
Removing a Key

Move the key and item of *max_node* to *node*.



Removing a Key

Finally, remove *max_node* and have its parent point to *max_node.left*.



Note, *max_node.right* is always `null` because this is the maximum-key node lying under *node.left*.

Caution: *max_node* might not be the right child of its parent.

Considerations When Removing a Key

If you are deleting the root node (not just updating its key), there is no "parent" so make sure to update the root pointer itself.

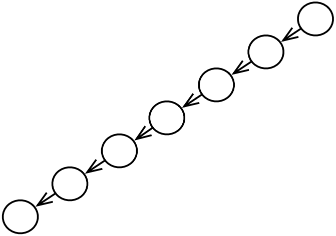
In Python, you do not actually **delete** the node like you would in `C++`. Simply removing all references to it suffices. It will eventually be garbage collected.

Test thoroughly! This is the most delicate function we have discussed so far in the class.

Algorithm 3 Removing the key k and its associated item.

```
node, parent ← find the node with key k, like in insert
if node is null then
   FAILURE, key not found
else if node.left is null then
   if parent is null then
      root ← node.right
   else
      change the appropriate child pointer of parent to node.right
else
   max_node, max_parent ← node.left, node
   while max_node.right is not null do
      max_node, max_parent ← max_node.right, max_node
   node.key, node.item ← max_node.key, max_node.item
   change the appropriate pointer of max_parent to max_node.left
```

All operations take $O(height)$ time. Can be $O(n)$ in the worst case!



So what advantage does this have?

Neat Theory: If the keys were inserted in random order then the expected height is $O(\log n)$.

But we want **guarantees**!

First: Let's do a worksheet and then implement this.