Math 260: Python programming in math

Fall 2020

Data structures II:
dictionaries, binary trees...

More data structures

What we have

Given an array-like structure a like a stack, consider the operations:

- **Append** at the 'end' of the structure
- **pop** an element from an 'end'
- **Insert** into the structure (at some non-endpoint index)
- **Look-up** an element (find it in the structure)
- **remove** an element (find, then remove it)

where 'end' and 'largest' are defined in thex structure.

For a stack and queue in python,

- Append: not fast? (Naive: $O(n)$; actual: better)
- Look-up, pop: $O(1)$ (using the list indexing)
- Insert/remove: $O(n)$

(pop for a queue should be $O(1)$, or at least can be made to work this way)

- How do we get good insertion cost? Answer: compromise - balance out look-up and insert costs and organize the data.

Binary search trees

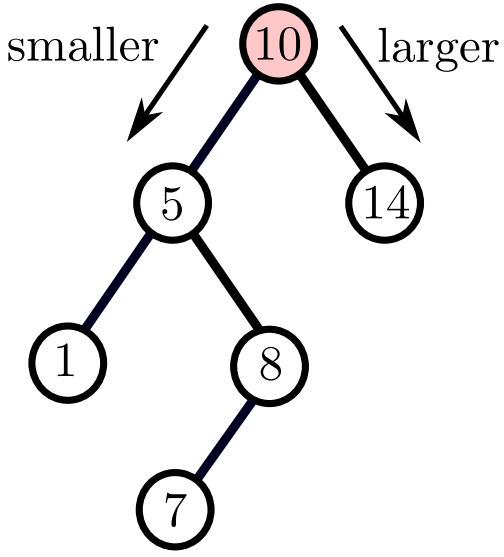
One solution (of several!): a **Binary search tree** (BST)

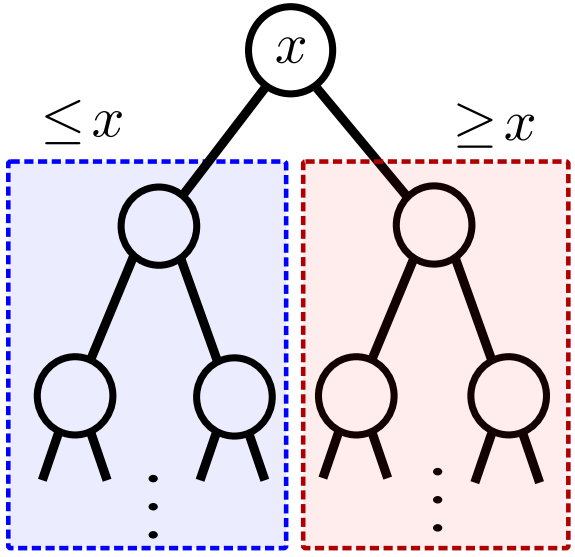
- A good balance for all types of operations
- Heaps, hash tables, etc. may be better if not all operations are needed
- Typical usage: representing a large comparable set of data

  - insert/remove, get a range, combine... (good for data operations)

Defining properties:

- A tree where each node has up to two children
- Key property: For each node with data x, all data at and below its left child is $\leq x$ and all data at and below its right child is $\geq x$.





Binary search trees

- First we need a suitable node class:

```
class BinaryNode:
    def __init__(self, data, left=None, right=None, parent=None);
        self.data = data
        self.children = [left,right]
        self.parent = parent
```

- You can get away with not tracking the parent, but it will make some code a little easier to write

Now let's make a binary search tree class...

Features:

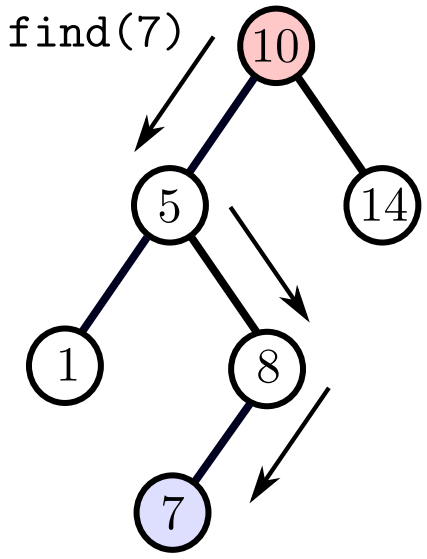
- find: finds a node with the given value (returns the node)
- delete: deletes a node from the tree
- insert: adds a node to the tree

```
class BinaryTree:
    def __init__(self, root);
        self.root = root
    def insert(self, val):
        # inserts a value into the tree
    def find(self, val):
        # finds a node with the given value
    def delete(self, node):
        # remove a node from the tree
```

Finding an element

Starting at the root node...

- if `val` is $>$ than the node's, go right
- Otherwise ($\leq$), go left
- If you can no longer go anywhere, stop and return `None` (failure case)
- Concise to write with a while loop...



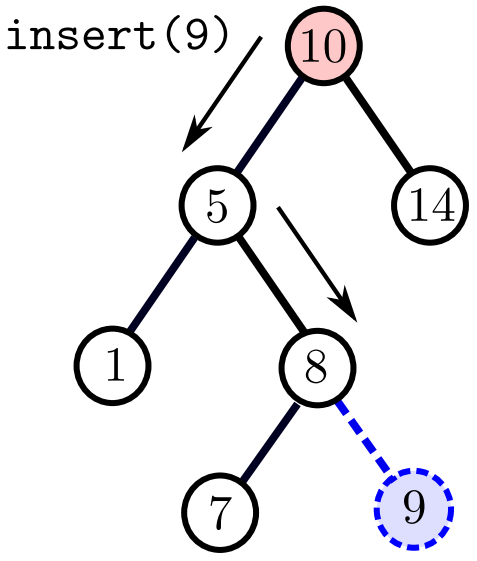

```
def find(self, val):
    n = self.root
    while n and n.data != val: #(Note: None is False)
        n = n.children[val > n.data]

    return n
```

- The BST property guarantees that this works!
- Note that `val > n.data` is zero (False) if `val` $\leq$ `n.data`
- Note that an empty child of a node is set as `None`

Inserting an element

Insert is almost the same as find!
To insert a value `val` into the tree:

- Start at the root node
- if `val` is $>$ than the node's, go right
- Otherwise ($\leq$), go left
- Stop when the next step would go to `None`
- Add a new node with `val` in this open spot




```
def insert(self, val):
    node = self.root
    while node:
        p = node # new node will attach to p
        node = node.children[val > node.data]
    n = BinaryNode(val)
    p.children[val > p.data] = n
    n.parent = p
```

- Searches until it finds an empty spot (`node`); then `p` is the node to attach to.
- New node is always a 'leaf' (no children)

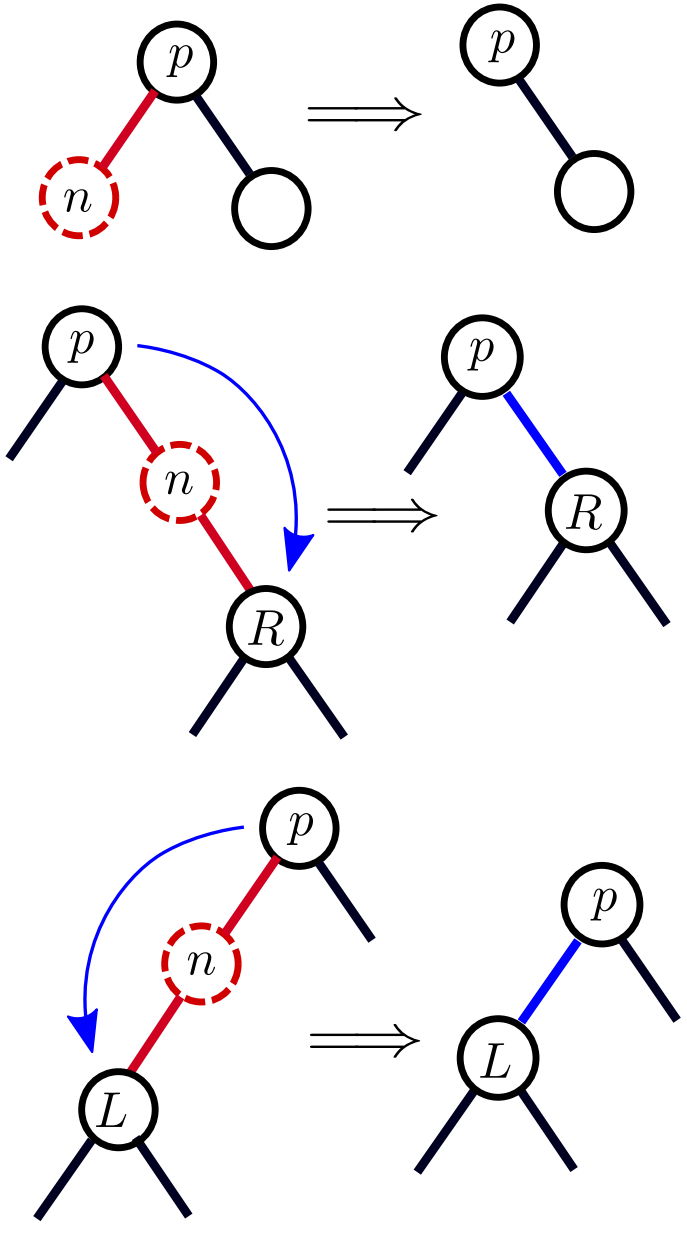
Deleting an element

Delete is more subtle than insert:

- We need to remove the node **and** maintain the BST property
- Cut, then re-attach branches of the tree...
- ...avoiding copying of data and nodes
- Easy cases: the node has at most one child

```
def delete(self, n):
    left = n.children[0]
    right = n.children[1]

    if not(left or right): # top picture
        side = node.data > n.parent.data
        n.parent.children[side] = None
    elif not left: #middle picture
        # connect parent to child of deleted node
        n.parent.children[1] = right
        right.parent = n.parent
    elif not right:
        ...
```

(Also: special cases for a root node...)

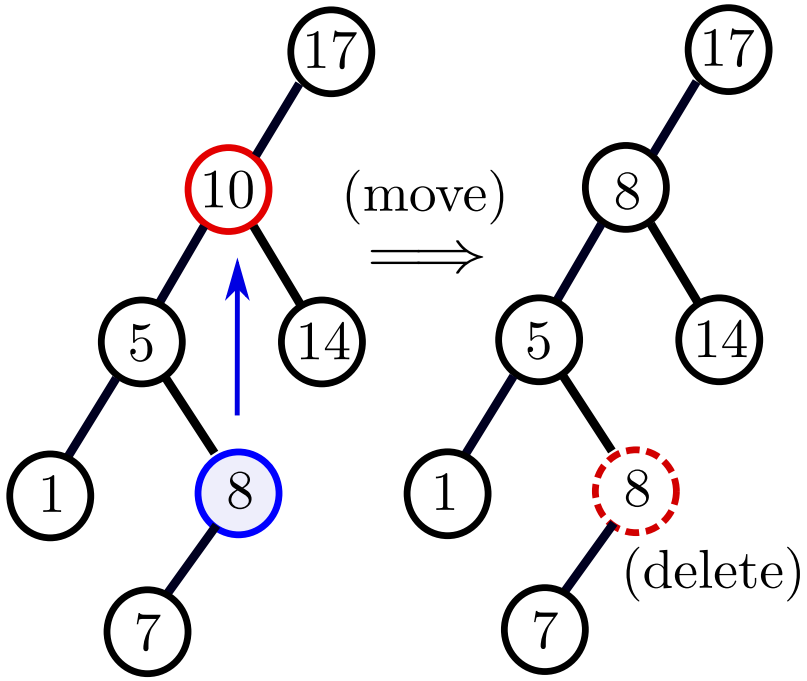


Deleting an element

The last case is recursive.
To delete `node` with two children:

- Find the largest 'predecessor' `pre` of `node`: the largest entry below it with a smaller value
- move the data from `pre` to `node`
- Then delete `pre` (recursively)

```
def delete(self,n):
    left = n.children[0]
    right = n.children[1]
    # ...other cases...
    else:
        # find pre by going right
        while left.children[1]:
            left = left.children[1]

        n.data = left.data  # move data
        self.delete(left)  # delete pre
```



The second call will be one of the easy cases - why?

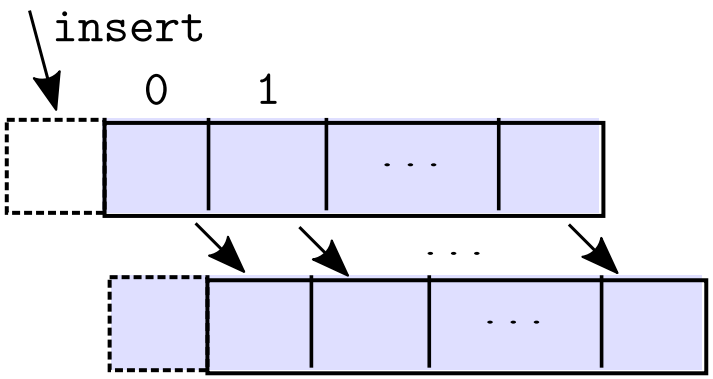
Binary search trees: efficiency

Let's now compare a sorted list to a tree...

Suppose I have n integers (e.g. a roster of student IDs). I need t be able to:

- check if a given integer is in the list
- add/remove them from the list.

Let's also assume the binary search trees are 'balanced'. That is, they take up as much space as possible at each depth.



Sorted list:

- Insert/remove: $O(n)$
- Find: $O(\log n)$
- Creation: $O(n \log n)$ (by sorting)

'Worst case' of $O(n)$ for an unbalanced tree can be avoided...

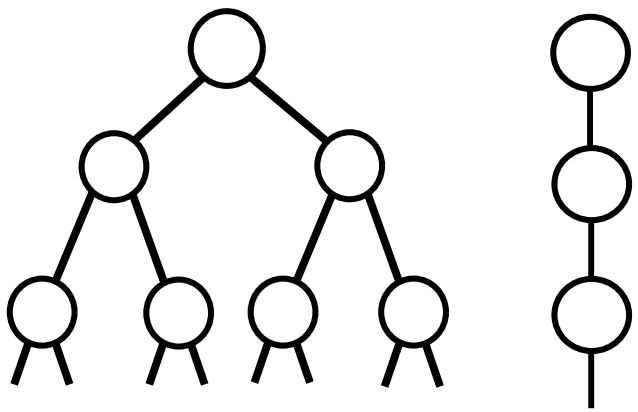
Binary search trees: efficiency

Let's now compare a sorted list to a tree...

Suppose I have n integers (e.g. a roster of student IDs). I need t be able to:

- check if a given integer is in the list
- add/remove them from the list.

Let's also assume the binary search trees are 'balanced'. That is, they take up as much space as possible at each depth.

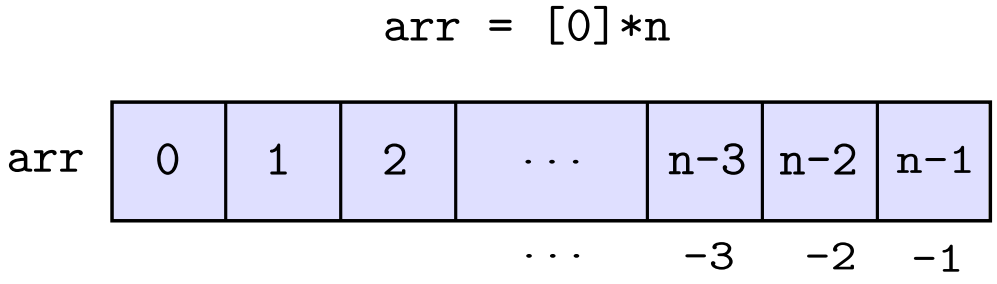


Binary tree

- Insert: $O(\text{depth})$
- Find: $O(\text{depth})$
- Both $O(\log n)$ if balanced

'Worst case' of $O(n)$ for an unbalanced tree can be avoided...

Hash tables



An **array** has the property that:

- Look-up for the element at index k is $O(1)$ (optimally fast)
- Setting the element at index k is $O(1)$
- The data can be 'contiguous' (one block of memory)

However, sometimes we have data in **key-value** pairs, where each 'key' is associated with a 'value', such as,,,

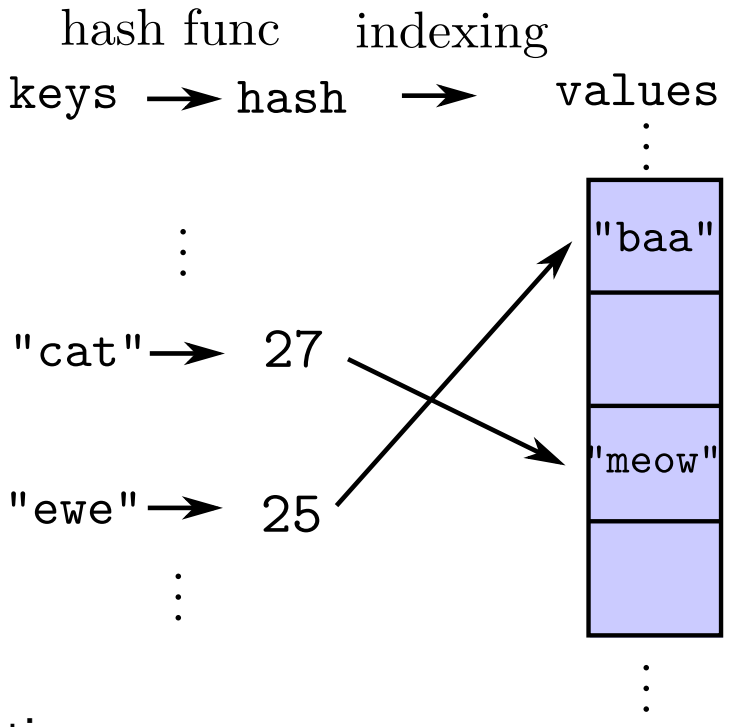
- Lists of properties and values
- words and definitions

An array is this, with positive integer keys. What if they are not integers?

Hash tables

Solution: use a **hash table**

- Create an array with enough space
- Define a 'hash function' that assigns a number (a 'hash') to any key
- Store the value for each key at that index
- If too many keys exist, resize the array



With a good hash function, we get nice properties:

- Array like look-up/insert: `table[key] = val` and `v = table[key]`
- Look up **and insert** in $\approx O(1)$ time (indexing, hash both cost $O(1)$)
- Any hashable object can be inserted

Hash tables

Subtleties:

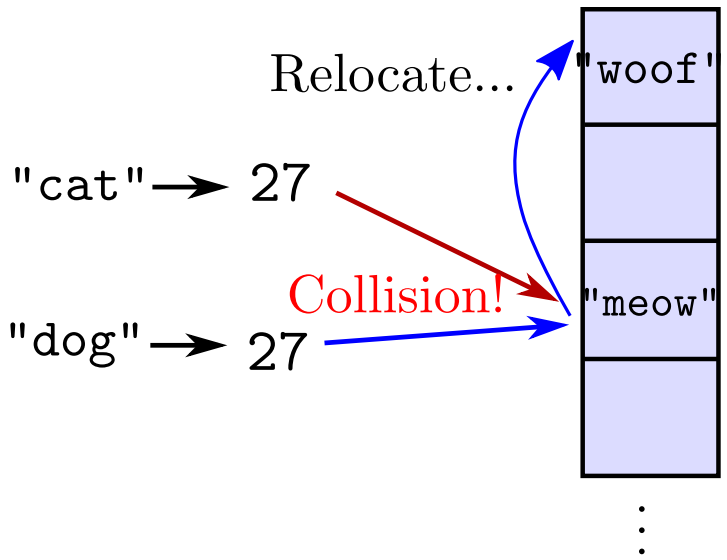
- Good hash functions are not trivial to create
- (you can use, e.g. functions of the object id)
- The table must handle **collisions** correctly

Once working, hash tables are a powerful tool.
Advantages:

- Flexible key types, $O(1)$ look-up/insert

Disadvantages:

- Data does not have locality
  (not consecutive in memory like arrays)
- Not good for iterating over all keys



(collisions are resolved by looking for an open slot to relocate to).

Aside: dynamic arrays

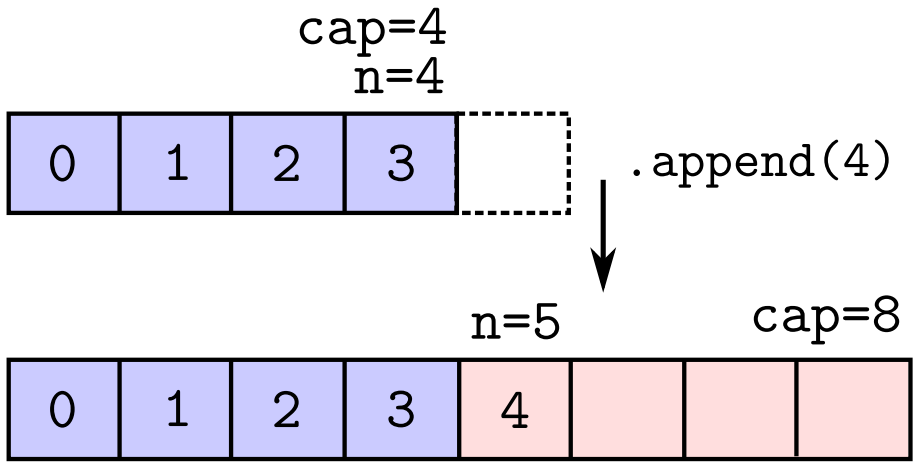
Now let's see how to resize an array, and how `append` actually works in python.

- We want an `Array` class that can 'resize' if needed...
- ...but does not do so every time an element is appended

A solution is to reserve a bit of extra space (a **buffer**) when resizing.

A sketch of an example ('alloc' represents allocating memory):

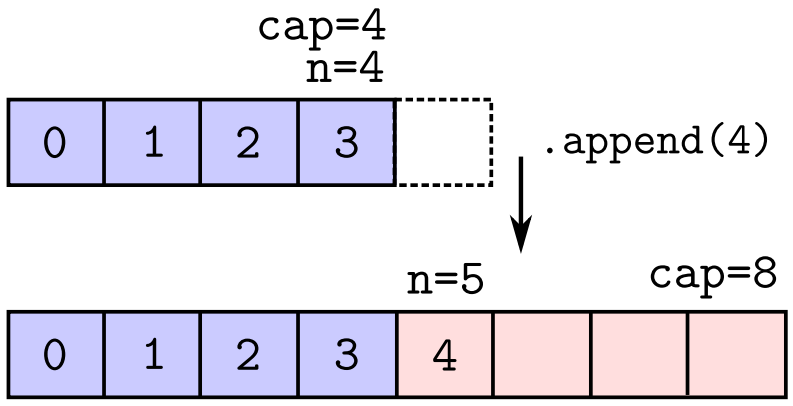
```
def alloc(k): # `allocate' memory
    return [0]*k

class Array: # just for example
    def __init__(self, n):
        self.n = n
        self.cap = n + 5
        self.data = alloc(self.cap)

    def resize(self, newsize):
        while self.cap < newsize:
            self.cap *=2
        space = alloc(self.cap)
        space[:n] = self.data[:n]
        self.data = space
        self.n = newsize
```



Aside: dynamic arrays

So what's the cost of appending?

- It's mostly cheap ($O(1)$)...
- but occasionally expensive ($O(n)$)



Suppose we start with an empty array, then append 1 to $n = 2^k$.

Total cost:

- n $O(1)$ operations to add elements
- k resizes, each taking $C \cdot 2^k$ operations

$$\begin{aligned} \text{total cost} &= O(n) + C(1 + 2 + \cdots + 2^k) \\ &= O(n) + C(2^{k+1} - 1) \end{aligned}$$

The **average cost** per append is

$$\text{avg. cost.} \approx O(n)`/n = O(1).$$

So we get $O(1)$ append time 'on average' ('**amortized constant**').
Generally: doing expensive operations only occasionally is still efficient.

Dictionaries

Key point: you don't need to implement this; Python has it built-in.
The type is called a **dictionary** (type `dict`). The syntax:

```
d = {'cat': 'meow', 'dog': 'woof', 13 : [1,2]}
d['cat'] # 'meow'
d[13] # [1,2]

d = dict() # empty dict
e = d # both point to the same dict (mutable)

d['sheep'] = 'baa' # add a new key/val pair
all_keys = d.keys() # get a list of the keys
keys, vals = d.items() # get the keys and values
```

You can check if a key is in a dictionary in $O(1)$ time
(Python only has to check one location - where the hash points)

```
d = {'cat': 'meow', 'dog': 'woof', 13 : [1,2]}
if 'cat' in d:
    print(d['cat'])

d['axolotl'] # raises KeyError
```

Note: `in` works for other collections (e.g. lists) too!

Dictionaries

Aside: `zip` creates an 'iterator' for iterating over a set of lists:

```
keys, vals = english_words_and_definitions()
for j in range(len(keys)):
    d[keys[j]] = vals[j]

d = dict(zip(keys, vals)) # fancy short version of above
```

```
for k, v in zip(keys, vals):
    print(k,v)
```

equiv. to

```
for j in range(len(keys)):
    print(keys[j],vals[j])
```

An iterator defines how for loops, etc. iterate ('starting' and 'next' element).

`zip((a,b,...))` says to start with `(a[0],b[0],,..)`,
then 'next' goes from `(a[k],b[k],,..)` to `(a[k+1],b[k+1],,..)`

Aside: zip and iterators

Python uses dictionaries whenever key-value pairs are needed

Every class has a dictionary mapping instance variables names to values

```
class Foo:
    def __init__(self):
        self.bar = 1
        self.c = 2
a = Foo()
for var, value in a.__dict__.items():
    print('a has {}={}'.format(var, value))
```

You can pass a dict. of 'keyword args' with the special 'kwargs syntax':

```
def func(**kwargs):
    for k in kwargs.keys():
        print("Input: {}={}".format(k, kwargs[k]))
func(b=1,cat='meow')
# prints...
#  Input: b=1
#  Input: cat='meow'
```

Aside: enumerate

Often you want to iterate over a set, but also need the index:

```
names = ["alpha", "beta", "gamma"]
for k in range(len(names)):
    print{"{} has index {}".format(name[k], k))
```

When you really want to iterate *over the set*, use enumerate:

```
for k, name,in enumerate(names):
    print("{} has index {}".format(name, k))
```

This trick cleans up code where the index is not needed much.

(`enumerate` creates pairs of indices and values.)

Comparing data structures

Cost of common operations (bad in red, good boxed)

- Array (fixed length):
  - one continuous block, integer indexing
  - append: not supported
  - Insert/look-up: $O(n)$ and O(1)
- Array (dynamic):
  - Array, but with a resize scheme
  - $O(1)$ (amortized) append, insert still $O(n)$
- Hash table:
  - no continuous block of data; not fast to iterate
  - Insert/look-up: O(1) (amortized) and O(1)
- Binary search tree:
  - great when keeping 'order' of data is important
  - Insert/look-up: typically $O(\log n)$, worst case $O(n)$