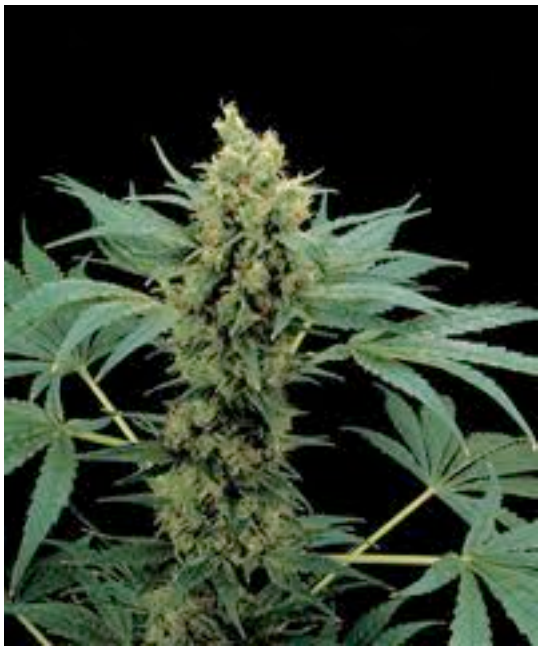
Hash Tables

Lecturer: John Guttag

hashing in CS: Convert the key to an integer and then use that integer to index into a list, which we know can be done in constant time.

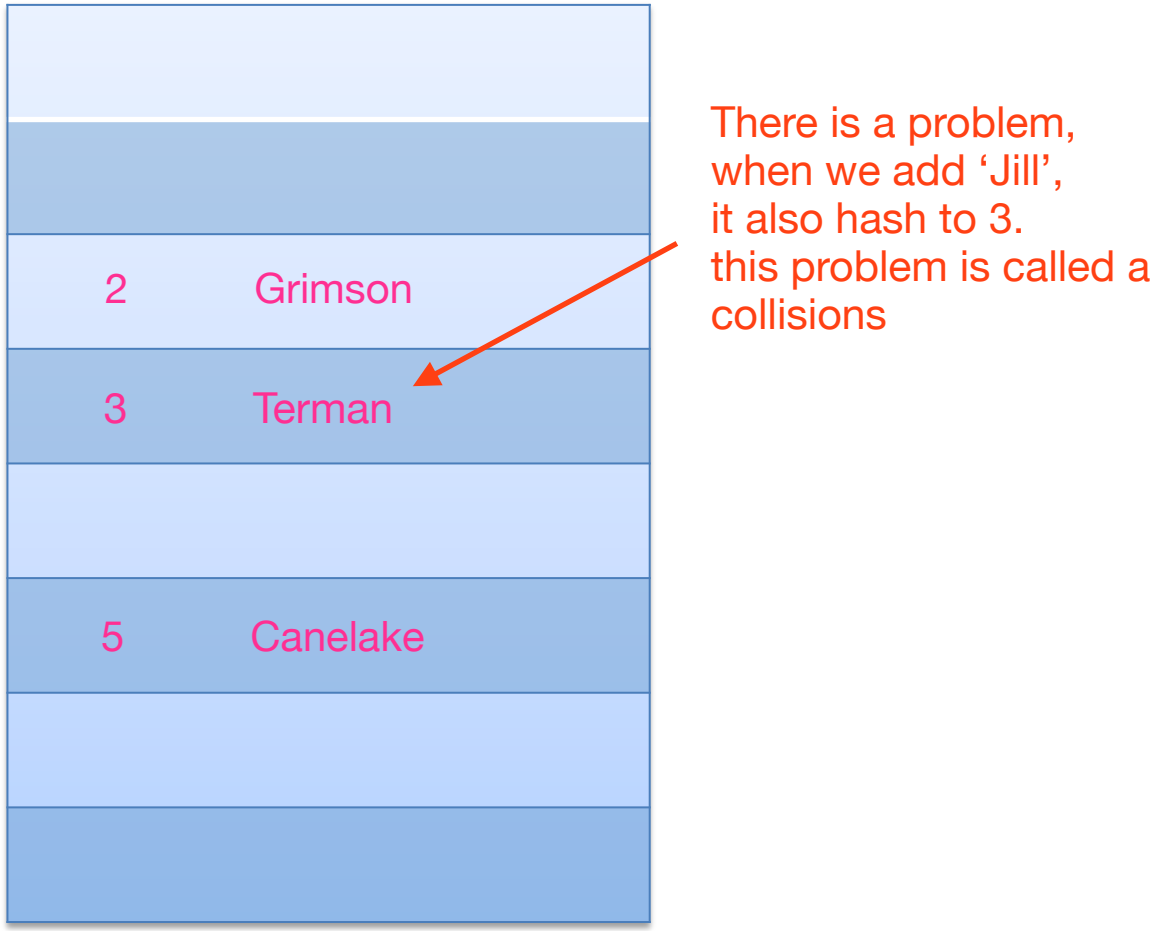


```
def strToInt(s):
    number = ''
    for c in s:
        number = number + str(ord(c))
    index = int(number)
    return index
```

```
def hashStr(s, tableSize = 101):
    number = ''
    for c in s:
        number = number + str(ord(c))
    index = int(number)%tableSize
    return index
```

This is a mapping of first names to last names.
for example,
key=Eric, value=Grimson
so when tableSize = 7, hashStr('Eric', 7) will returns 2

in the same way,
"Chris Terman" → hashStr('Chris', 7) returns 3,
"Sarina Canelake" → 5



There is a problem,
when we add 'Jill',
it also hash to 3.
this problem is called a
collisions

We have collisions because a hash function is a many-to-one mapping. the whole point was to take a very large space and map each element in that space into a much smaller space. Inevitably, if we're doing that, sooner or later we're going to have to map more than one element from this big space to the same point in the smaller space. And that is what produces a collision.

initialize each element of the table to be the empty list

Hash Bucket

[('Eric', 'Grimson')]

[('Chris', 'Terman'), ('Jill', 'Smith')]

time-space trade off:
table large → few collisions and look up fast;
table small → more collisions and look up slow.