8. Symbol tables. Hashing. Hash tables

Agenda

- What is symbol table
- Symbol tables in Python
- Symbol table key requirements
- Hashing
- Collision resolution
- Separate-chaining symbol table implementation

1. Symbol table

Key-value pair abstraction:

- **Insert** a **value** with specified **key**
- **Given** a **key**, **search** for the corresponding **value**

Examples:

- Dictionary: given a term (key), find the definition (value)
- File system: given a file name (key), find the location on disk (value)
- Phone: given a person's name (key), find their phone number (value)

2. Symbol table

- **Also known as:** maps, dictionaries, associative arrays
- Can also be viewed as **generalized arrays** where keys need not be between 0 and N-1 (where N – number of elements)
- Applications in information technology:
  - Caches
  - Search
  - File systems
  - Databases
  - Networks
- Symbol tables are one of the most widely used data structures in programming

3. Symbol tables in Python

- Symbol tables are called dictionaries in Python
- Create a dictionary:
  - `d = {}`
  - `d = dict()`
- Put a key-value mapping (will overwrite value if key is already present):
  - `d['key1'] = 'value1'`
- Get a value by key:
  - `d['key1'] # example: print(d['key1'])`
- Check if key is in the dictionary:
  - `'key' in d.keys() # example: if 'key' in d.keys(): … (can also use not in)`
- Remove key (and corresponding value) from dictionary:
  - `del d['key1']`
  - `d['key1'].pop() # also returns the corresponding value`

3. Symbol tables in Python

- None is allowed both as a key and as a value:
  - `d['key'] = None`
  - `d[None] = 'val'`
- Because of this, looking up a key that is not present in the dictionary throws `KeyError`:

```
d = {}
d['key1'] = 'value1'
print(d['key2']) # throws KeyError
del d['key2'] # also KeyError
d['key2'].pop() # KeyError here as well
```

- Possible to provide the default value when a key is not in the dictionary:

```
d = {'key1': 'value1'}
print(d.get('key2', 'default_value')) # prints 'default_value'
```

- Iterate over all key-value pairs in the dict:
  - `for k, v in d.items(): ...`

3. Symbol tables in Python

- **Value** can be any object
- However, there are requirements for dictionary **keys**:

```
class SomeObject:
    def __init__(self, a, b):
        self.a = a
        self.b = b

d = {}
d[SomeObject(1, 's1')] = 'value'
print(d[SomeObject(1, 's1')]) # throws KeyError
```

- When using user-defined objects as a key, these objects need to be able to return their **hash** (more on this later) and have a way to test **equality**

3. Symbol tables in Python

- Proper implementation could look like this:

```
class SomeObject:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __hash__(self):
        return hash((self.a, self.b))

    def __eq__(self, other):
        return (self.__class__ == other.__class__ and
                self.a == other.a and
                self.b == other.b)

SomeObject(1, 's1') == SomeObject(2, 's2') # __eq__ method is called
```

User-defined types as dictionary key

Requirements:

- **Equality.** For any references **x**, **y** and **z**, `__eq__` must be implemented in a way to be:
  - **Reflexive:** `x == x` is `True`
  - **Symmetric:** `x == y` iff `y == x`
  - **Transitive:** if `x == y` and `y == z`, then `x == z`
- Should be **immutable** (means fields should never change)

Symbol table implementations

- **Linked list or array:**
  - **Search.** Scan through all keys until find a match
  - **Insert.** Scan through all keys until find a match; if no match add to front/back
- **Ordered array of key-value pairs:**
  - **Keys.** Need to use comparable types or implement comparison (for example, using `__lt__` method)
  - **Search.** Use binary search
  - **Insert.** If the given key is present, use binary search to find the place to update the value. Otherwise, use comparison (`__lt__`) to find the place to insert the new key, and shift all greater keys over

Complexity

ST implementation	Average case		Worst case		Key requirement
	Search	Insert	Search	Insert	
Linked list or array	`N / 2`	`N`	`N`	`N`	`__eq__`
Ordered array	`log N`	`N / 2`	`log N`	`N`	`__lt__`

Hashing

- **Plan:** save items in a key-indexed table (index is a function of the key)
- Hash function – method for computing array index from key
- Accepts an arbitrary object, and produces integer value according to some rule
- Python's built-in hash function returns signed integer (size depends on the system architecture – 32 bit for 32-bit systems, 64 bit for 64-bit systems):

```
print(hash('first_key'))  # returns -5801299530476810693

print(hash('second_key'))  # returns -3260146112784830420
```

1. Hashing function

- **Properties:**
  a. **Required:** if `x == y`, the `x.__hash__() == y.__hash__()`
  b. **Highly desirable, but not always possible** due to the limited range of returned values (integers): if `x != y`, then `x.__hash__() != y.__hash__()`
- When the property (b) does not hold, this is called **hash collision**
- **Goal:** scramble the keys uniformly to produce a table index
  a. Efficiently computable
  b. Each table index equally likely for each key (thoroughly researched, but a hard problem)

2. Hashing function

- **Example of a hashing function.** Suppose we want to use phone numbers as keys, and therefore compute the hash:
  a. Bad implementation: use first three digits (bad because the first 1-3 digits of a phone number are the country code)
  b. Better implementation: use last three digits
- Widely used string hashing function:

```
def string_hash(string):
    hash = 0
    for character in string:
        hash = ord(character) + (31 * hash)
    return hash
```

- Equivalent to (Horner's method):

$$h = s[0] \cdot 31^{L-1} + \ldots + s[L-3] \cdot 31^2 + s[L-2] \cdot 31^1 + s[L-1] \cdot 31^0$$

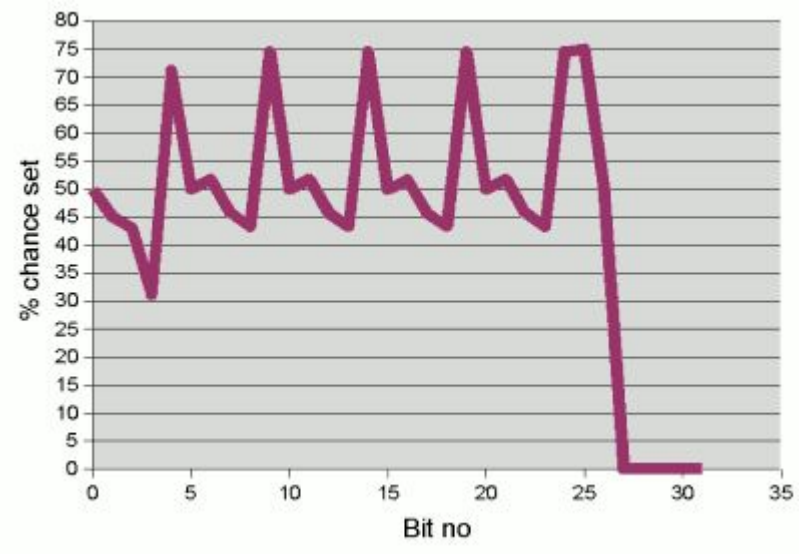
- Complexity of hashing string of length L: L multiplies/adds

1. Why 31?

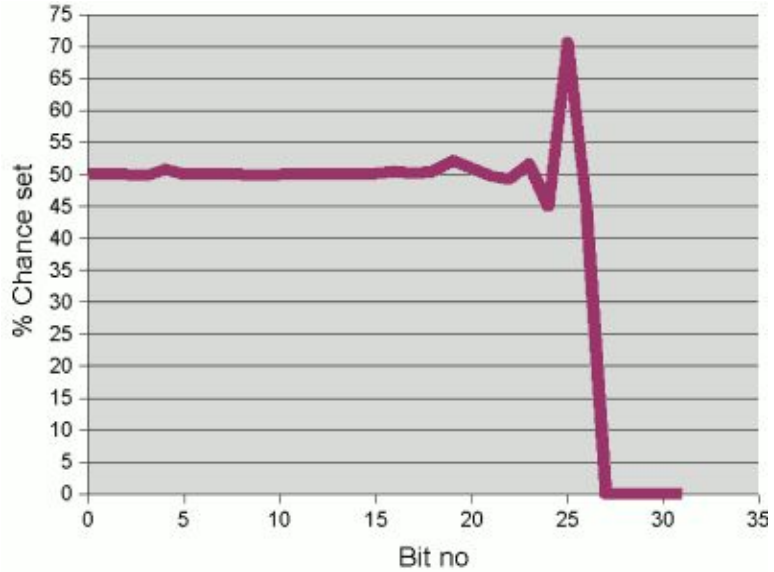
“The value 31 was chosen because it is an odd prime. If it were even and the multiplication overflowed, information would be lost, as multiplication by 2 is equivalent to shifting. The advantage of using a prime is less clear, but it is traditional. A nice property of 31 is that the multiplication can be replaced by a shift and a subtraction for better performance: `31 * i == (i << 5) - i`. Modern VMs do this sort of optimization automatically.” (Joshua Bloch “Effective Java”)

2. Why 31?

- Probability of successive bits being set in the hash code of a randomly-generated 5 character string, when X is chosen as the multiplier in the hash function:



X = 32



X = 31

Hash code design

- General guidelines:
  - Combine each field (or array element) using 31x + y rule
  - If field implements hash code (for example, built-in type), use the available implementation
  - If field is None, return 0
  - If field is of user-defined / compound type, apply rule recursively
- This method is fast and works reasonably well in practice
- However, there are better hashing functions

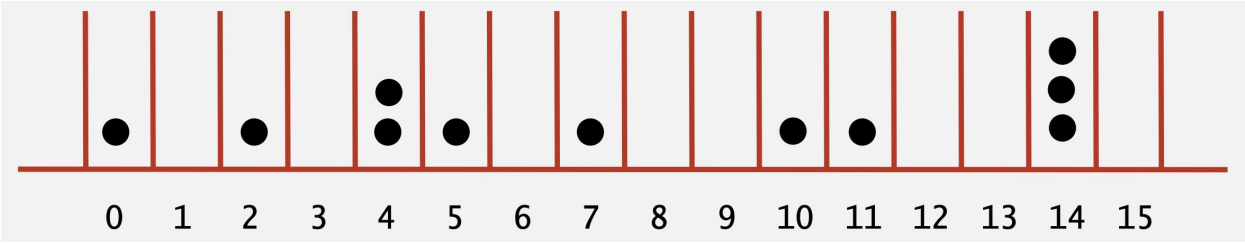
Modular hashing

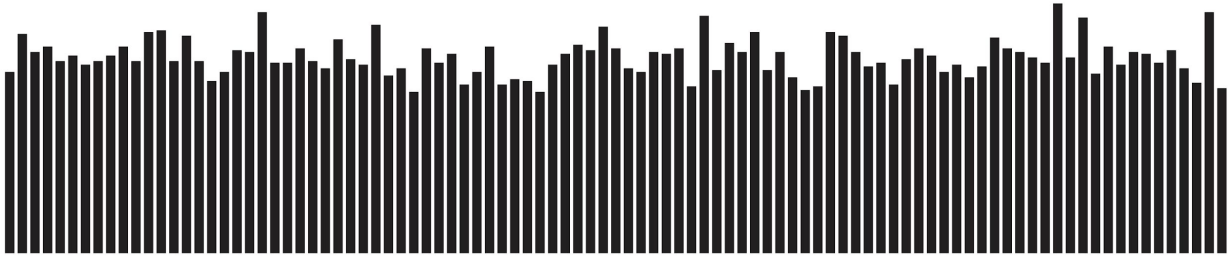
- We need to use hash code to find index in an array to place the key
- Therefore, we need to limit the range of returned values to array bounds
- We use modulo operation to achieve this:

```
def find_index(string, array_length):
    return string_hash(string) % array_length


string = 'hello world'
print(string_hash(string))  # returns 8800692682095891 6
print(find_index(string, array_length=20))  # returns 16
```

Uniform hashing assumption

- **Uniform hashing assumption.** Each key is equally likely to hash to an integer between 0 and M – 1
- **Bins and balls.** Throw balls uniformly at random into M bins





Hash value frequencies for words in Tale of Two Cities (M = 97)

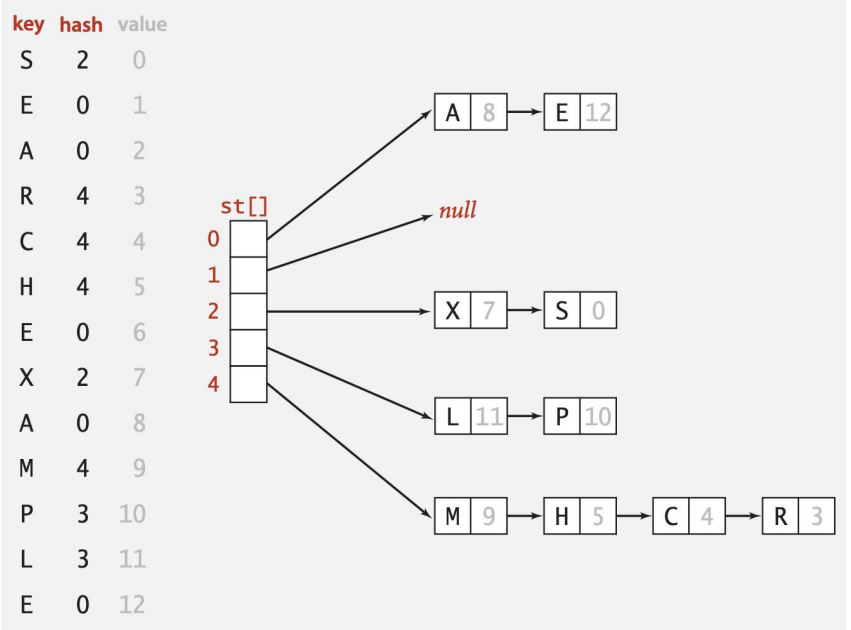
Collisions

- Collision happens when two distinct keys hash to the same index
- Birthday paradox:
  - Concerns the probability that, in a set of N randomly chosen people, some pair of them will have the same birthday:
    - 99.9% probability is reached with just 70 people
    - 50% probability with 23 people
  - Means we can't avoid collisions unless we have a huge (quadratic) amount of memory
- Therefore, we need to deal with collisions efficiently

Separate-chaining symbol table

- **Design:** have a chain (linked list) to handle collisions: if multiple keys hash to the same bucket (array index), store them in the chain

- **Hash:** map key to integer **i** between **0** and **M - 1**
- **Insert:** put at front of **i**th chain
- **Search:** hash key and find the bucket (**i**), and then do linear search over all elements in the chain



1. Separate-chaining symbol table implementation

- Create basic structure: class for our hash table, and inside it the class for internal node
- Initialize by creating an empty numpy array that will contains elements of type Node
- Size of the internal array is configurable

```
class SeparateChainingHashST:
    class Node:
        def __init__(self, key, value, next):
            self.key = key
            self.value = value
            self.next = next

    def __init__(self, bucket_count):
        self.arr = np.empty(bucket_count,
dtype=self.Node)
        self.bucket_count = bucket_count
```

2. Separate-chaining symbol table implementation

- Here we define the function that performs modular hashing
- We also define put function:
  - Compute array index based on hash
  - If there is no Node at this index, create one and put the given key-value pair there
  - If there are multiple nodes at this index, do sequential search to find the given key. If found, replace value. Otherwise, insert new node at the front of the chain

```
def find_index(self, key):
    return key.__hash__() % self.bucket_count

def put(self, key, value):
    index = self.find_index(key)

    node = self.arr[index]
    while node is not None:
        if key == node.key:
            node.value = value
            return
        node = node.next

    self.arr[index] = self.Node(key, value,
next=self.arr[index])
```

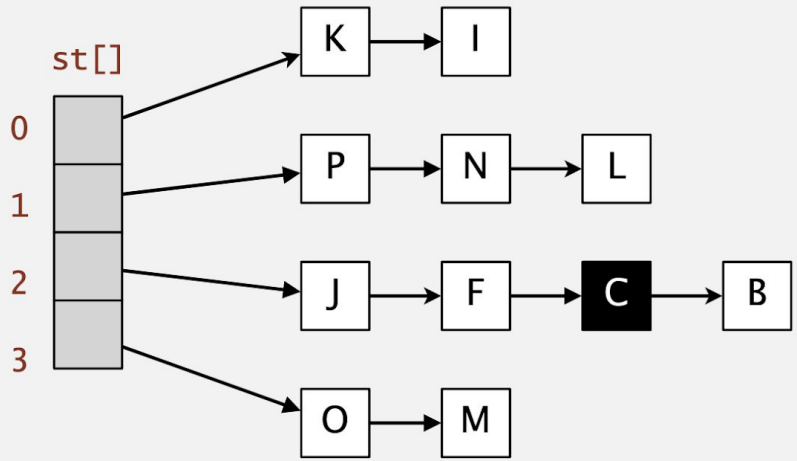
3. Separate-chaining symbol table implementation

- Define get method:
  - Again, compute array index based on the hash
  - If there is no Node at this index, return and raise KeyError (consistent with built-in dict interface)
  - If the bucket is not empty, traverse the chain, looking for the given key. If found, return the value. Otherwise, raise KeyError

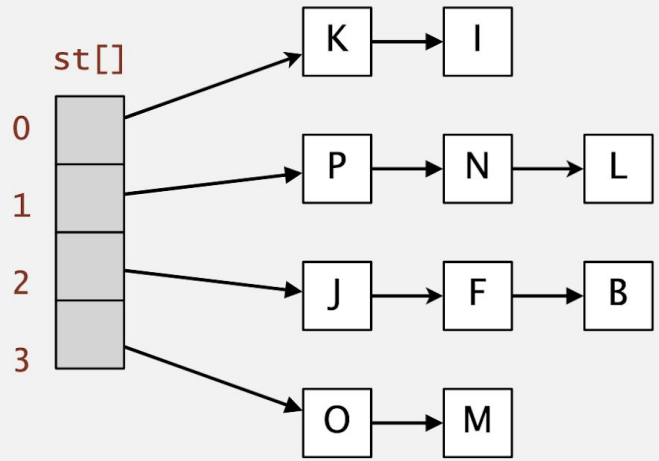
```
def get(self, key):
    index = self.find_index(key)

    node = self.arr[index]
    while node is not None:
        if key == node.key:
            return node.value
        node = node.next

    raise KeyError(key)
```

Deletion in separate-chaining symbol table

before deleting C



after deleting C



4. Separate-chaining symbol table implementation

```
def delete(self, key):
    index = self.find_index(key)

    node = self.arr[index]
    previous_node = None
    while node is not None:
        if key == node.key:
            if previous_node:
                previous_node.next = node.next
            else:
                self.arr[index] = None
            return node.value
        previous_node = node
        node = node.next

    raise KeyError(key)
```

4. Separate-chaining symbol table implementation

```
def __str__(self):
    key_value_pairs = []
    for bucket in self.arr:
        node = bucket
        while node is not None:
            key_value_pairs.append((node.key, node.value))
            node = node.next
    return str(key_value_pairs)
```

Using this symbol table

```
hash_dict = SeparateChainingHashST(8)
hash_dict.put('key1', 'value1')
hash_dict.put('key2', 'value2')
hash_dict.put('key3', 'value3')
hash_dict.put('key4', 'value4')
hash_dict.put('key5', 'value5')
print(hash_dict.get('key1')) # prints 'value1'
print(hash_dict.get('key4')) # prints 'value4'
print(hash_dict) # prints '[('key2', 'value2'), ('key5', 'value5'), ('key1',
'value1'), ('key4', 'value4'), ('key3', 'value3')]'
print(hash_dict.delete('key2')) # prints 'value2'
print(hash_dict) # prints '[('key5', 'value5'), ('key1', 'value1'), ('key4',
'value4'), ('key3', 'value3')]'

# if we rename 'put' to '__setitem__', and 'get' to '__getitem__', we can use
this syntax:
hash_dict['key1'] = 'value1'
print(hash_dict['key1'])
```

Analysis of separate chaining

- **Property.** Under uniform hashing assumption, probability that the number of keys in a list is within a constant factor of N / M is extremely close to 1
- **Consequence.** Number of probes for search/insert is proportional to N / M (M times faster than sequential search)
  - M too large ⇒ too many empty chains
  - M too small ⇒ chains too long
  - Typical choice: M <= N / 0.75 (standard Java implementation, offers a good tradeoff between time and space costs)
- Average complexity for search, insertion and deletion is constant (under uniform hashing assumption)
- Worst-case complexity is linear: with a bad choice of hashing function (for example, using the one that returns a constant value for every input), the hash table degrades to a single chain, and search becomes sequential

Complexity. Summary

ST implementation	Average case		Worst case		Key requirement
	Search	Insert	Search	Insert	
Linked list or array	`N / 2`	`N`	`N`	`N`	`__eq__`
Ordered array	`log N`	`N / 2`	`log N`	`N`	`__lt__`
Separate-chaining hash table	`constant`	`constant`	`N`	`N`	`__eq__ + __hash__`