Python in Practice

Review

* Big O Notation
* Master Theorem
* Searching and Sorting

Data Application Lab

Overview

* Introduction to Data Structure and Algorithm
* Data Structure
  * Searching and Sorting
  * List
  * Stack and Queue
  * Linked List
  * Dictionary
  * Binary Search Tree
  * Heap
* Algorithm
  * Divide and Conquer
  * Dynamic Programming
  * Greedy

Data Application Lab

Python Dictionary

* {Key:Value} pair
  * dict = {‘Tom’: ‘Cat’, ‘Jerry’: ‘Mouse’ , ‘Simba': ‘Lion'}
  * dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
* Access Values
* Update Dictionary
* Delete Dictionary Element
* Properties of Dictionary Keys
  * More than one entry per key not allowed
  * Keys must be immutable: strings, numbers, tuples
* Application: Word Count

Data Application Lab

Python Dictionary

k in M:	Return True if the map contains an item with key k. In Python, this is implemented with the special __contains__ method.
M.get(k, d=None):	Return M[k] if key k exists in the map; otherwise return default value d. This provides a form to query M[k] without risk of a KeyError.
M.setdefault(k, d):	If key k exists in the map, simply return M[k]; if key k does not exist, set M[k] = d and return that value.
M.pop(k, d=None):	Remove the item associated with key k from the map and return its associated value v. If key k is not in the map, return default value d (or raise KeyError if parameter d is None).
M.popitem():	Remove an arbitrary key-value pair from the map, and return a (k,v) tuple representing the removed pair. If map is empty, raise a KeyError.
M.clear():	Remove all key-value pairs from the map.
M.keys():	Return a set-like view of all keys of M.
M.values():	Return a set-like view of all values of M.
M.items():	Return a set-like view of (k,v) tuples for all entries of M.
M.update(M2):	Assign M[k] = v for every (k,v) pair in map M2.
M == M2:	Return True if maps M and M2 have identical key-value associations.
M != M2:	Return True if maps M and M2 do not have identical key-value associations.

Data Application Lab

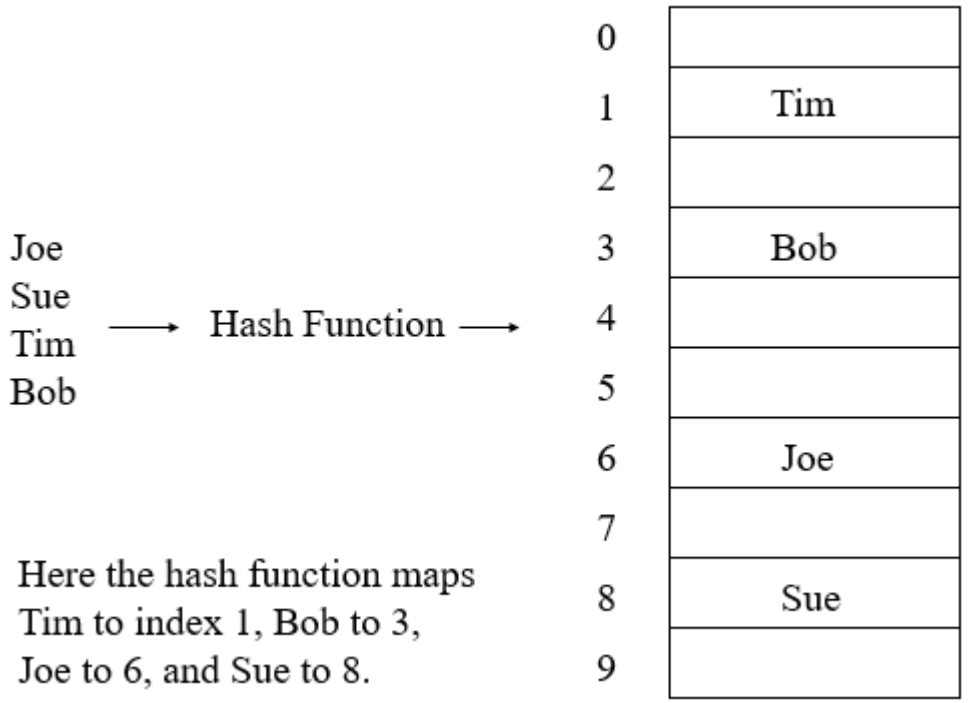
Basic Idea

* Associative Array, Map, Symbol Table, Dictionary
* How Dictionary works?
* Composed of *(Key, Value)* pairs
* Ideally, each key appears just once in the collection
* Operations
  * the addition of pairs to the collection
  * the removal of pairs from the collection
  * the modification of the values of existing pairs
  * the lookup of the value associated with a particular key

Data Application Lab

Simplify Words

* Consider an array of items each of which contains a key, such as name for a bank record.
* Each key is mapped into some value in the range of 0 to the array size – 1.
* This mapping function is called a hash function.
* The hashing function should be a simple algorithm that distributes the keys evenly among the cells.



Basic Hashing

* The direct indexing approaches above work by
  * setting aside a big enough block for all possible data items
  * spacing these so that the address of any item can be found by a simple calculation from its ordinarily
* A hash function maps any key to a valid array position
* For integer keys, (*key mod N*) is the simplest hash function
  * SSN, Phone Number, Zip code
* In general, any function that maps from the space of keys to the space of array indices is valid
* But a good hash function spreads the data out evenly in the array
* *Hash function h:* Mapping from U to the slots of a hash table *T[0..m–1]*.

$$h : U \rightarrow \{0, 1, \ldots, m-1\}$$

  * With arrays, key *k* maps to slot *A[k]*.
  * With hash tables, key *k* maps or "hashes" to slot *T[h[k]]*.
  * *h[k]* is the hash value of key *k*.

Computing the Hash Function

* Idealistic goal. Scramble the keys uniformly to produce a table index.
  * Efficiently computable
  * Each table index equally likely for each key
  * Thoroughly researched problem, still problematic in practical applications
* Example One: Phone Numbers
* Example Two: Social Security Numbers
  * Bad: first three digits
  * Better: last three digits
* Example Three: Person
  * First Name, Last Name
  * Birthday
* Practical challenge. Need different approach for each key type

Hash Code Conventions

* Key: hashable values
* __hash__
* __eq__
* Requirement
  * If x equals y, then they have same hash values
  * Highly desirable. If x not equals y, then they have different hash values
* Legal (but poor) implementation. Always return 17.
* User-defined types. Users are on their own.

Hash Code Design

* "Standard" recipe for user-defined types
  * Combine each significant field using the 31x + y rule (Horner Rule)
  * If field is a primitive type, use wrapper type hashCode()
  * If field is null, return 0
  * If field is a reference type, use hashCode() ← applies rule recursively
  * If field is an array, apply to each entry ← or use Arrays.deepHashCode()
* Horner's Rule gives us a simple way to compute a polynomial using multiplication and addition:

$$a_0 + a_1x + a_2x^2 \ldots + a_nx^n = a_0 + x(a_1 + x(a_2 + \ldots + xa_n)\ldots)$$

* In practice. Recipe works reasonably well; used in Java libraries
* Basic rule. Need to use the whole key to compute hash code; consult an expert for state-of-the-art hash codes.

Data Application Lab

Collision

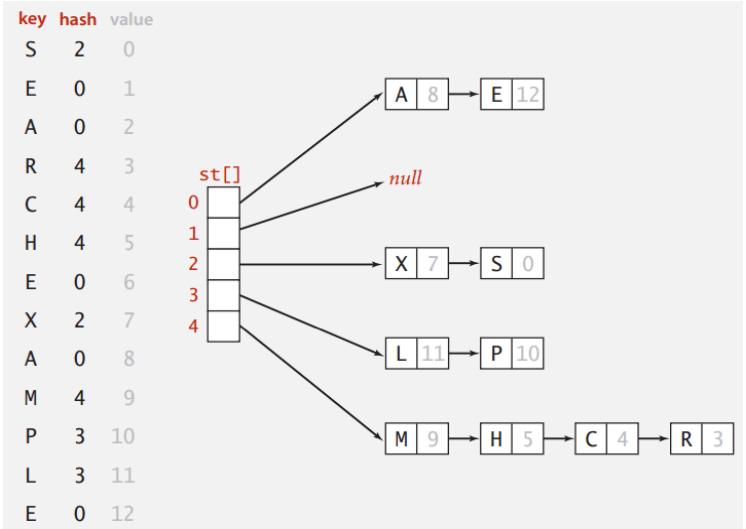
* What if we use a block which is not big enough for all possible items?
  * Addressing function must map all items into this space.
  * Some items may get mapped to the same position ⇒ called a *collision*.
* A collision occurs when h(x) maps two keys to the same location.
* Challenge: Deal with collisions efficiently
* Collision Resolution
  * Bucketing, Separate Chaining
  * Probing, Open Addressing

Bucketing

* Allow more than one item to be stored at each position in the hash table
  * ➔ associate a List with each hash table cell. . .
* Bucketing
  * Each list is represented by a fixed size block
* Advantages
  * Simple to implement: hash to address, then search list
* Disadvantages
  * Searching the list slows down Table access
  * Fixed size ➔ may waste a lot of space
  * Buckets may overflow → back where we started, a collision is just an overflow with a bucket size of 1

Separate Chaining

* Allow more than one item to be stored at each position in the hash table
  * ➔ each list is represented by linked list or chain
* Use an array of M < N linked lists
  * Hash: map key to integer i between 0 and M – 1
  * Insert: put at front of i th chain (if not already there).
  * Search: need to search only i th chain.
* Advantages
  * Simple to implement: hash to address, then search list
  * No overflow
* Disadvantages
  * Searching the list slows down Table access
  * Extra space for pointers (if we are storing records of information the space used by pointers will generally be small compared to the total space used)
  * Performance deteriorates as chain lengths increase



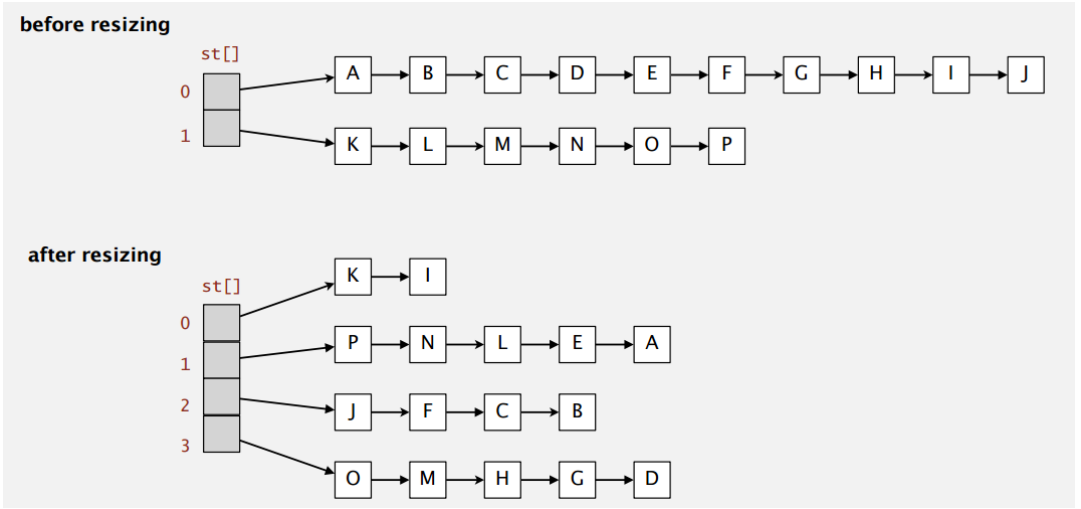
Data Application Lab

Performance

* Worst Case
  * all items stored in a single chain
  * $O(n)$ → same as List, no gain
* But expected case performance is much better. . .
* Load factor λ: the number of items N in the table divided by the size M of the table.
* Consequence:
  * λ too large ⇒ almost full, size of list maybe large as well
  * λ too small ⇒ many empty chains

Rehashing

* Worst Case
  * all items stored in a single chain
  * *O(n)* → same as List, no gain
* But expected case performance is much better. . .
* Load factor λ: the number of items N in the table divided by the size M of the table.
* Goal.
  * re-hashing when hashtable is filles beyond *λ*
  * increase the size of hashtable
  * Need to rehash all keys when resizing
  * *x.hashCode()* does not change
  * But *hash(x)* can change
  * Java: 0.75



Open Addressing

* When a new key collides, find next empty slot, and put it there
* If h(x) is occupied, try h(x)+f(i) mod N for i = 1 until an empty slot is found
* Many ways to choose a good f(i)
* Simplest method: Linear Probing
  * $f(i) = i$
* Linear-probing Hash Table
  * Hash. Map key to integer i between 0 and $M-1$.
  * Insert. Put at table index i if free; if not try $i+1$, $i+2$, etc.

Data Application Lab

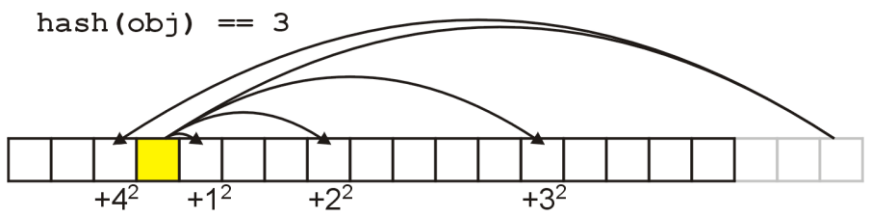
Linear Probing

* Increment hash index by one (with wrap-around) until the item, or null, is found
* Removals
  * How do we delete when probing?
  * Lazy deletion: mark as deleted
  * We can overwrite it if inserting
  * But we know to keep looking if searching
* Primary Clustering
  * If there are many collisions, blocks of occupied cells form: primary clustering
  * Any hash value inside the cluster adds to the end of that cluster
  * it becomes more likely that the next hash value will collide with the cluster
* Instead of searching forward in a linear fashion, try to jump far enough out of the current (unknown) cluster

Quadratic Probing

* Suppose that an element should appear in position h:
  * if h is occupied, then check the following sequence of the array:

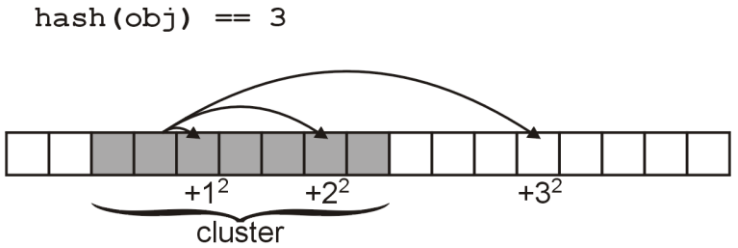
$$h + 1^2,\ h + 2^2,\ h + 3^2,\ h + 4^2,\ h + 5^2, \ldots$$
$$h + 1,\ h + 4,\ h + 9,\ h + 16,\ h + 25, \ldots$$



Data Application Lab

Quadratic Probing

* If one of $h + i^2$ falls into a cluster, this does not imply the next one will



* Even if two bins are initially close, the sequence in which subsequent bins are checked varies greatly
* Thus, quadratic probing solves the problem of primary clustering
* Unfortunately, there is a second problem which must be dealt with
* Suppose we have M = 8 cells
  * $1^2 \equiv 1, 2^2 \equiv 4, 3^2 \equiv 1$
* In this case, we are checking cell h + 1 twice having checked only one other cell

Quadratic Probing

* Unfortunately, there is no guarantee that

$$h + i^2 \bmod M$$

will cycle through 0, 1, ..., M – 1

* Solution:
  * require that M be prime
  * in this case, $h + i^2 \bmod M$ for $i = 0, \ldots, (M - 1)/2$ will cycle through exactly $(M + 1)/2$ values before repeating

Quadratic Probing

* Example with M = 11:

  $0, 1, 4, 9, 16 \equiv 5, 25 \equiv 3, 36 \equiv 3$

* With M = 13:

  $0, 1, 4, 9, 16 \equiv 3, 25 \equiv 12, 36 \equiv 10, 49 \equiv 10$

* With M = 17:

  $0, 1, 4, 9, 16, 25 \equiv 8, 36 \equiv 2, 49 \equiv 15, 64 \equiv 13, 81 \equiv 13$

Data Application Lab

Quadratic Probing

* Thus, quadratic probing avoids primary clustering
* Unfortunately, we are not guaranteed that we will use all the cells
* In reality, if the hash function is reasonable, this is not a significant problem until λ approaches 1
* Secondary Clustering
  * The phenomenon of primary clustering will not occur with quadratic probing
  * However, if multiple items all hash to the same initial cell, the same sequence of numbers will be followed
  * This is termed secondary clustering
  * The effect is less significant than that of primary clustering

Double Hashing

* Double hashing uses a secondary hash function d(k) and handles collisions by placing an item in the first available cell of the series

  $(i + jd(k)) \bmod N$

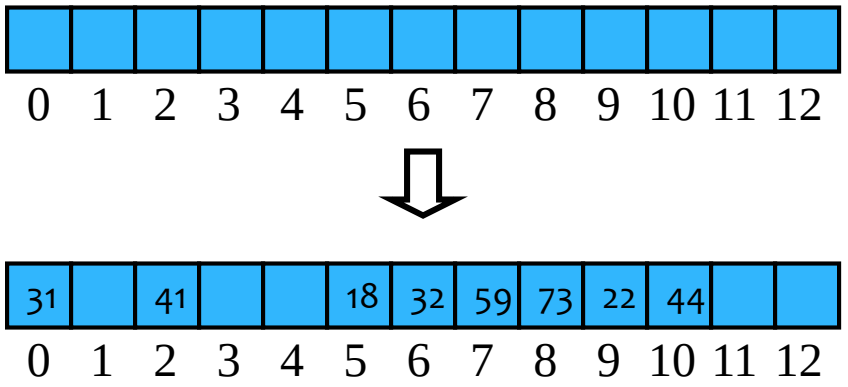
  for $j = 0, 1, \ldots, N - 1$
* The secondary hash function $d(k)$ cannot have zero values
* The table size N must be a prime to allow probing of all the cells
* Common choice of compression function for the secondary hash function:
  * $d_2(k) = q - (k \bmod q)$
  * where
    * q < N
    * q is a prime

Data Application Lab

Example of Double Hashing

* Consider a hash table storing integer keys that handles collision with double hashing
  * $N = 13$
  * $h(k) = k \bmod 13$
  * $d(k) = 7 - k \bmod 7$
* Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		



Data Application Lab

Summary

* Hash tables can be used to:
  * improve the space requirements of some ADTs for which bounded representations are suitable
  * improve the time efficiency of some ADTs, such as Table, which require unbounded representations
  * In the worst case, searches, insertions and removals on a hash table take O(n) time
  * The worst case occurs when all the keys inserted into the map collide
  * The expected running time of all the dictionary ADT operations in a hash table is O(1)
  * In practice, hashing is very fast provided the load factor is not close to 100%
  * When the load gets too high, we can rehash.…
* We have seen a number of methods for collision resolution in hash tables:
  * bucketing and separate chaining
  * open addressing, including linear probing and quadratic probing
  * double hashing

Caution

* DO NOT “Hash first, ask question later”
* ALWAYS “Ask question first, hash later”

* (Hash can be additive ! ! !)

* For many applications, such as those naturally represented by trees, hashing would lose the structure.

Data Application Lab

Practice

* Two Sum (1E), Three Sum (1E), Four Sum (15M)
  * Given an array of integers, find two numbers such that they add up to a specific target, where *index1* must be less than *index2*.
* Contains Duplicates (217E)
  * Given an array of integers, find if the array contains any duplicates. Your function should return true if any value appears at least twice in the array, and it should return false if every element is distinct.
* Contains Duplicates II (219E)
  * Given an array of integers and an integer k, find out whether there are two distinct indices i and j in the array such that nums[i] = nums[j] and the difference between i and j is at most k.

Practice

* Word Pattern (290E)
  * Given a pattern and a string str, find if str follows the same pattern.
  * Here follow means a full match, such that there is a bijection between a letter in pattern and a non-empty word in str.
  * Examples:
    * pattern = "abba", str = "dog cat cat dog" should return true.
    * pattern = "abba", str = "dog cat cat fish" should return false.
    * pattern = "aaaa", str = "dog cat cat dog" should return false.
    * pattern = "abba", str = "dog dog dog dog" should return false.

Practice

* Anagrams (49M)
  * Given an array of strings, return all groups of strings that are anagrams.
* Max Points on a Line (149H)
  * Given n points on a 2D plane, find the maximum number of points that lie on the same straight line.

Data Application Lab

Python in Practice
The End