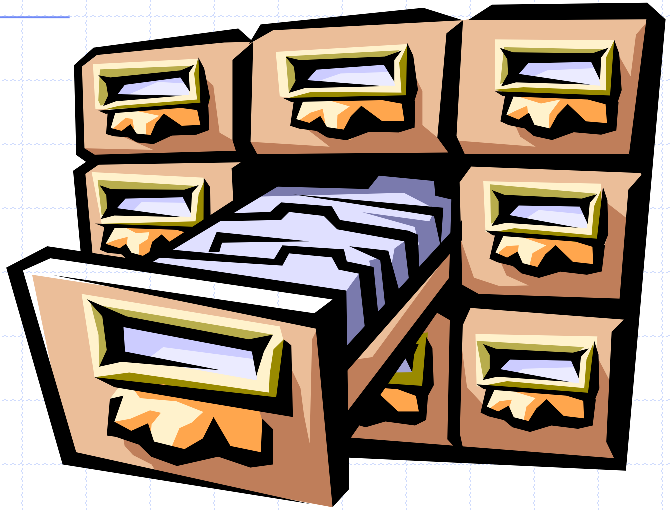
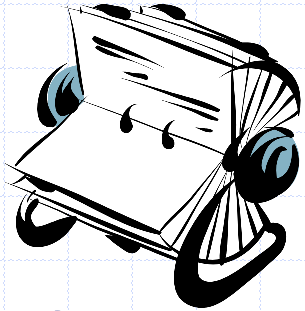
Maps and Dictionaries



Maps



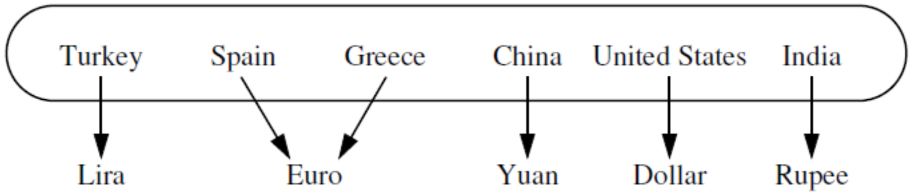
- A **map** is a searchable collection of items that are key-value pairs
- The main operations of a map are for searching, inserting, and deleting items
- Multiple items with the same key are not allowed (Unlike Heap)
- Applications:
  - address book
  - student-record database

Dictionaries

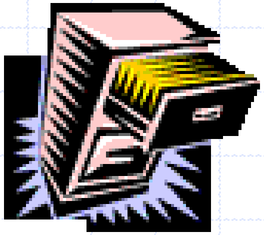
- Python's **dict** class is arguably the most significant data structure in the language.
  - It represents an abstraction known as a **dictionary** in which unique **keys** are mapped to associated **values**.
- Here, we use the term "dictionary" when specifically discussing Python's dict class, and the term "map" when discussing the more general notion of the abstract data type.

Examples

- Natural language dictionary
  - word is key
  - element contains word, definition, pronunciation, etc.
- Web pages
  - URL is key
  - html or other file is element
- Any typical database (e.g. student record)
  - has one or more search keys
  - each key may require own organizational dictionary



The Map ADT (Using **dict** Syntax)



- Get
- Set
- Delete

M[k]: Return the value v associated with key k in map M, if one exists; otherwise raise a KeyError. In Python, this is implemented with the special method __getitem__.

M[k] = v: Associate value v with key k in map M, replacing the existing value if the map already contains an item with key equal to k. In Python, this is implemented with the special method __setitem__.

del M[k]: Remove from map M the item with key equal to k; if M has no such item, then raise a KeyError. In Python, this is implemented with the special method __delitem__.

len(M): Return the number of items in map M. In Python, this is implemented with the special method __len__.

iter(M): The default iteration for a map generates a sequence of *keys* in the map. In Python, this is implemented with the special method __iter__, and it allows loops of the form, **for** k **in** M.

More Map Operations

contains

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).

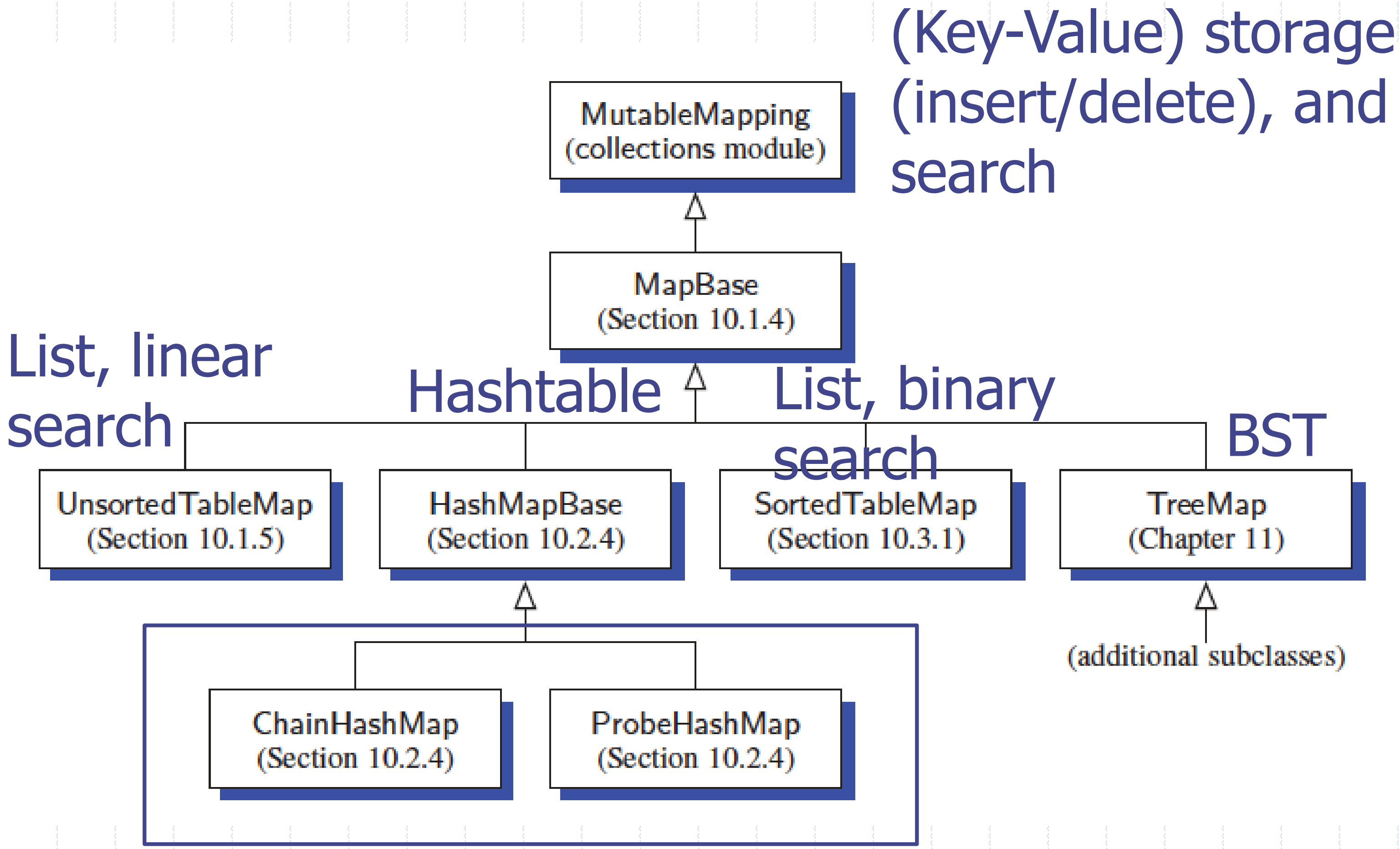
A Few More Map Operations

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.

Example

Operation	Return Value	Map
len(M)	0	{ }
M['K'] = 2	–	{'K': 2}
M['B'] = 4	–	{'K': 2, 'B': 4}
M['U'] = 2	–	{'K': 2, 'B': 4, 'U': 2}
M['V'] = 8	–	{'K': 2, 'B': 4, 'U': 2, 'V': 8}
M['K'] = 9	–	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M['B']	4	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M['X']	KeyError	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M.get('F')	None	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M.get('F', 5)	5	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M.get('K', 5)	9	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
len(M)	4	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
del M['V']	–	{'K': 9, 'B': 4, 'U': 2}
M.pop('K')	9	{'B': 4, 'U': 2}
M.keys()	'B', 'U'	{'B': 4, 'U': 2}
M.values()	4, 2	{'B': 4, 'U': 2}
M.items()	('B', 4), ('U', 2)	{'B': 4, 'U': 2}
M.setdefault('B', 1)	4	{'B': 4, 'U': 2}
M.setdefault('A', 1)	1	{'A': 1, 'B': 4, 'U': 2}
M.popitem()	('B', 4)	{'A': 1, 'U': 2}

Our MapBase Class

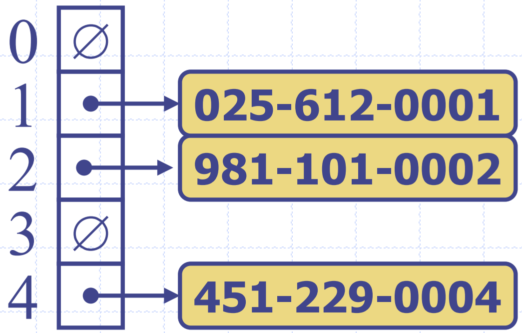


The MapBase Abstract Class

```
class MapBase(MutableMapping):
  """Our own abstract base class that includes a nonpublic _Item class."""

  #------------------------------- nested _Item class -------------------------------
  class _Item:
    """Lightweight composite to store key-value pairs as map items."""
    __slots__ = '_key', '_value'

    def __init__(self, k, v):
      self._key = k
      self._value = v

    def __eq__(self, other):
      return self._key == other._key      # compare items based on their keys

    def __ne__(self, other):
      return not (self == other)          # opposite of __eq__

    def __lt__(self, other):
      return self._key < other._key       # compare items based on their keys
```

Define comparator ==, != , < so that we can compare two items x and y

Hash Tables



Why we need Hashing?

- Think about BST can be used to implement a Map
  - Search time of a key in BST is O(log(N))

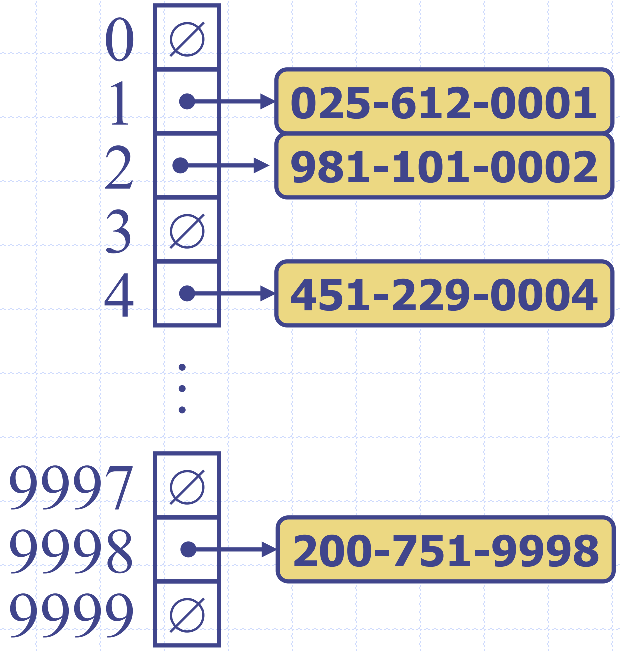
- Ideally a structure ought to allow constant average time for insertions/removals/searches
- Today's topic: Hashing

Introducing Hashing

- Hashing is a technique that determines an entry's index using only its search key
- Hash function h(*x*)
  - Takes a search key *x* and produces an integer index i in a hash table implemented using a dynamic array A of size N
- The goal is to store item (x, v) at index i $= h(x)$; $A[i] = v$ (a very ideal case)
- Example: $h(x) = x \bmod N$ *# if x is an integer*

SSN Example

- We design a hash table for a map storing entries as (SSN, Name), where SSN (social security number) is a nine-digit positive integer
- Our hash table uses an array of size $N = 10{,}000$ and the hash function $h(x) = \text{last four digits of } x$



Hash Functions

- A hash function is usually specified as the composition of two functions:

  Hash code:

  h_1: keys $\rightarrow$ integers

  Compression function:

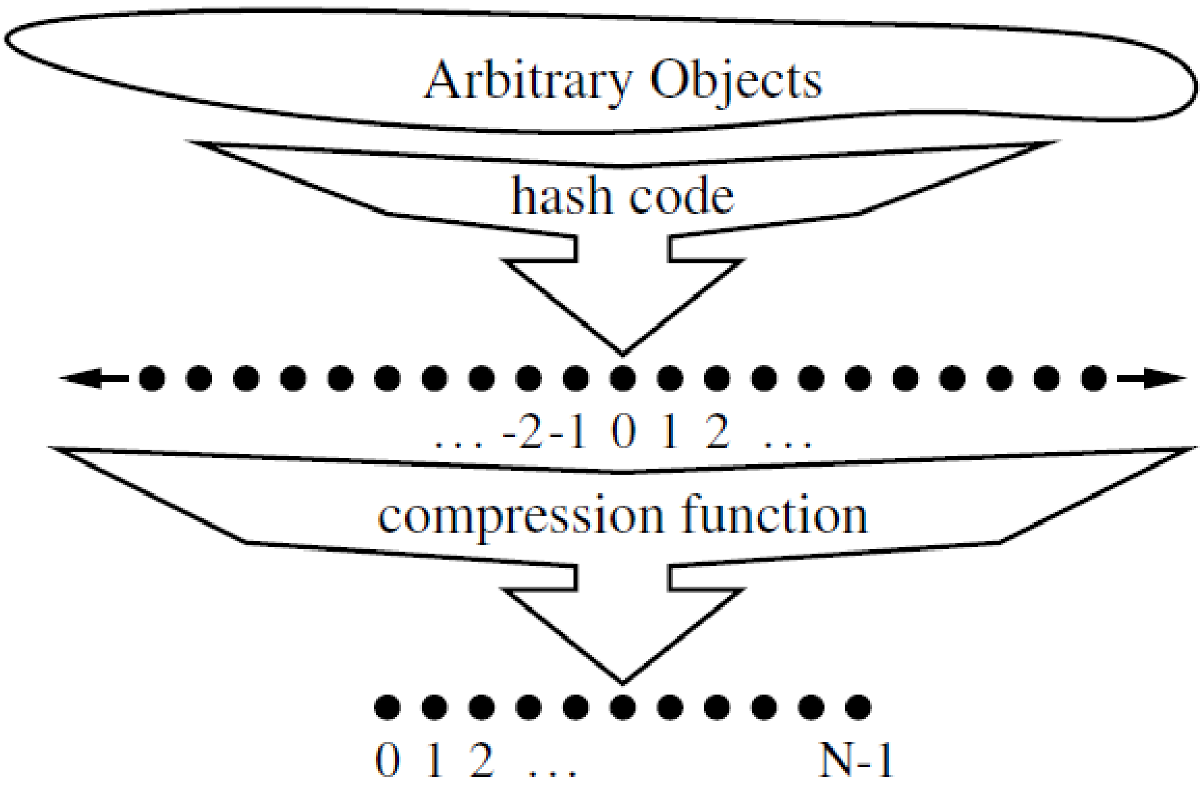
  h_2: integers $\rightarrow [0, N-1]$

- The hash code is applied first, and the compression function is applied next on the result, i.e.,

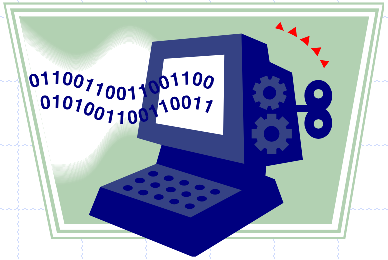
$$h(x) = h_2(h_1(x))$$

- The goal of the hash function is to "disperse" the keys in an apparently random way

Hash Functions (two parts)



Hash Codes



- Memory address:
  - We reinterpret the memory address of the key object as an integer; Good in general, except for numeric and string keys

```python
x = 0.15
y = 0.15

print(id(x))
print(id(y))
```

```
139975315702768
139975316104560
```

- Integer cast:
  - We reinterpret the bits of the key as an integer (i.e. Type casting of a key, e.g. float32)
  - Suitable for keys of length less than or equal to the number of bits of an integer (32-bit)

Hash Codes

- Polynomial Hash Codes:
  - We partition the bits of a key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits)

    key = $a_0\, a_1\, \ldots\, a_{n-1}$
  - We evaluate the polynomial

    $p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_{n-1} z^{n-1}$

    at a fixed value z *(e.g. z=33)*, ignoring overflows

e.g. h_1('the') = ord('t') + ord('h') * z + ord('e') * z^2

= 127 + 104 * 33 + 101 * 33^2

= 113515

  - Especially suitable for strings (e.g., the choice $z = 33$ gives at most 6 collisions on a set of 50,000 English words)

Hash Codes in Python

- Python has a built-in function with signature hash(x) that returns an integer value that serves as the hash code for object x.
- Only *immutable* data types are deemed hashable in Python.
- Function that computes hash codes can be implemented in the form of a special method named __hash__() within a class.
- You may override __hash__() if needed

Compression Functions

- Division:
  - $h_2(y) = y \bmod N$
  - The size N of the hash table is usually chosen to be a prime
  - The reason has to do with number theory and is beyond the scope of this course
- Multiply, Add and Divide (MAD):
  - $\mathbf{h_2(y) = ((ay + b) \bmod p) \bmod N}$
  - p is a prime number larger than N
  - a, b are chosen at random from [0,p-1] with a > 0

Hash Functions

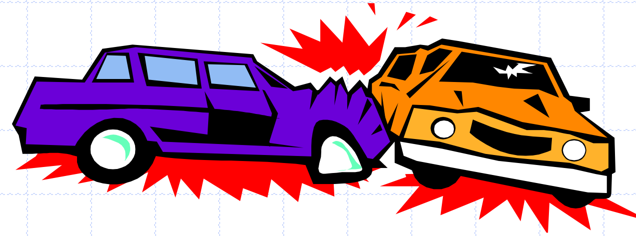
- A good hash function should
  - Minimize collisions
  - Be fast to compute

- To reduce the chance of a collision
  - Choose a hash function that distributes entries uniformly throughout the hash table.

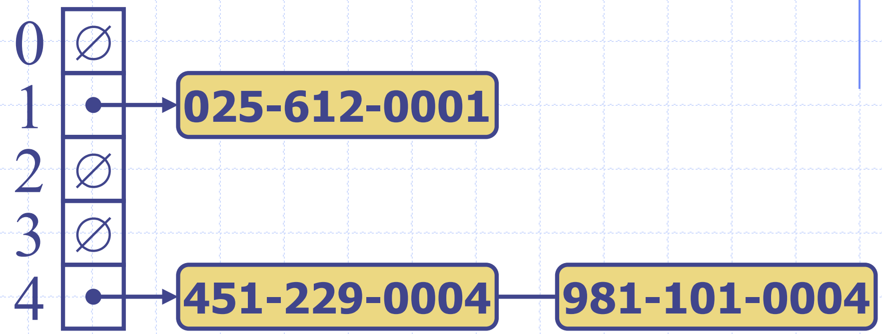
Resolving Collisions

- Definition: hash function maps search key into a location already in use in the hash table
- Two choices:
  - Use another location in the hash table
  - Change the structure of the hash table so that each array location can represent more than one value

Collision Handling (Separate Chaining)



- Collisions occur when different elements are mapped to the same cell
- Separate Chaining: let each cell in the table point to a linked list of entries that map there



- Separate chaining is simple, but requires additional memory outside the table

Map with Separate Chaining

- Delegate operations to a list-based map at each cell:

Algorithm get(x):
  j = h(x) // obtain a bucket index j via hashing
  return A[j].get(x)

Algorithm put(x,v):
  j = h(x) // obtain a bucket index j via hashing
  t = A[j].put(x,v)
  if t = **null then** {x is a new key}
    n = n + 1
  return t

Algorithm remove(x):
  j = h(x) // obtain a bucket index j via hashing
  t = A[j].remove(x)
  if t ≠ **null then** {x was found}
    n = n - 1
  return t

Separate Chaining (Cont.)

- In worst case, operations on an individual bucket take time proportional to the size of the bucket.
- Assuming we use a good hash function to index the n items of our map in a bucket array of capacity N, the expected size of a bucket is n/N.
  - A good hash function should disperse keys uniformly
- If given a good hash function, the core map operations run in $O(\mathrm{Ceil}(n/N))$.
- The ratio $\lambda = n/N$, called the ***load factor*** of the hash table.

Resolving Collisions (Linear Probing)

Linear probing

- A different technique to avoid separate chaining
- Resolves a collision during hashing by examining consecutive locations in hash table
- Beginning at original hash index

  Find the next available one (*probe sequence*)

Linear Probing

- Open addressing: the colliding item is placed in a different cell of the table
- Linear probing: handles collisions by placing the colliding item in the next (circularly) available table cell
- Each table cell inspected is referred to as a "probe"

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

0	1	2	3	4	5	6	7	8	9	10	11	12

⇩

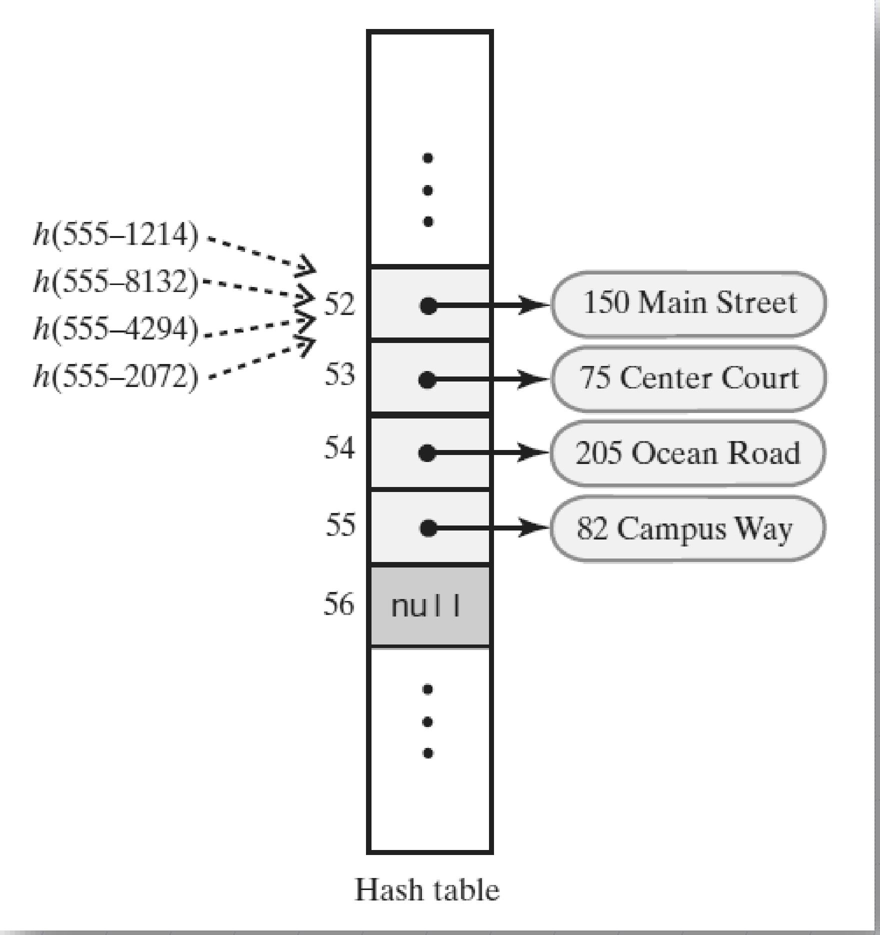
		41			18	44	59	32	22	31	73	
0	1	2	3	4	5	6	7	8	9	10	11	12

Linear Probing Issues

- Colliding items lump together, causing future collisions to cause a longer sequence of probes

Issue #1 Clustering:

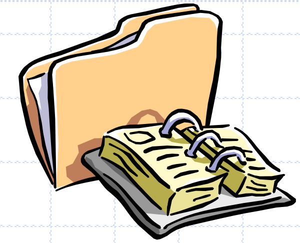
All keys hash into index 52!



Clustering

- Collisions resolved with linear probing cause groups of consecutive locations in hash table to be occupied
  - Each group is called a *cluster*
- Bigger clusters mean longer probing times following collision

Search with Linear Probing



- Consider a hash table A that uses linear probing
- get(k)
  - We start at cell $h(k)$
  - We probe consecutive locations until one of the following occurs
    - An item with key k is found, or
    - An empty cell (None) is found, or
    - N cells have been unsuccessfully probed

Algorithm *get*(k)
    $i \leftarrow h(k)$ // hashed index i
    $p \leftarrow 0$ // number of probs
    repeat
        $c \leftarrow A[i]$
        if $c = \varnothing$
            return *null*
        else if $c.getKey\,() = k$
            return $c.getValue()$
        else
            $i \leftarrow (i + 1) \bmod N$ // *probe*
            $p \leftarrow p + 1$ // incr # prob
    until $p = N$
    return *null*

Search with linear probing

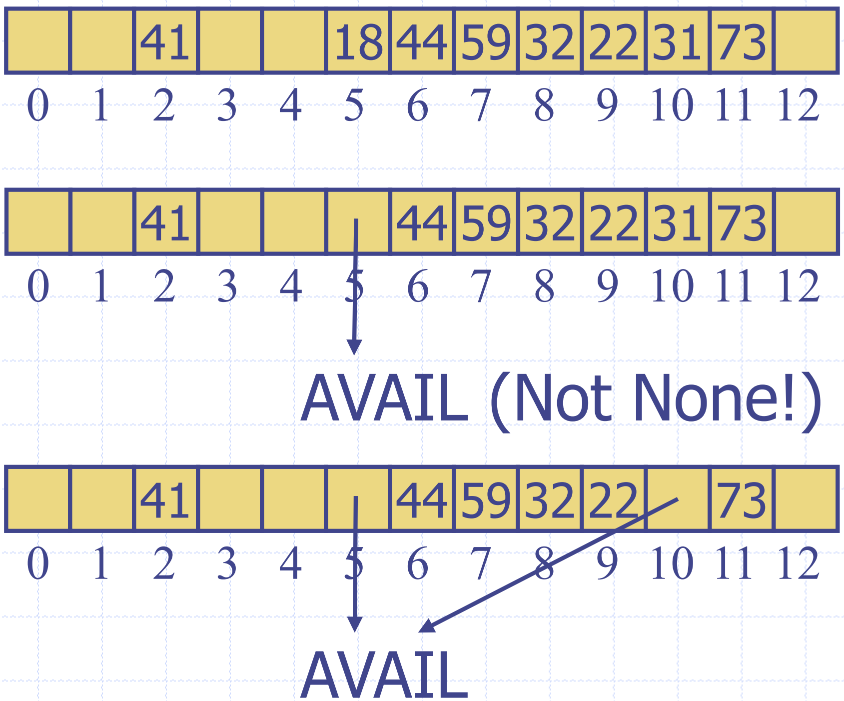
0	1	2	3	4	5	6	7	8	9	10	11	12
		41			18	44	59	32	22	31	73	

- $h(x) = x \bmod 13$

e.g.

- search 18
- search 31
- search 33

Deletion/Update with linear probing

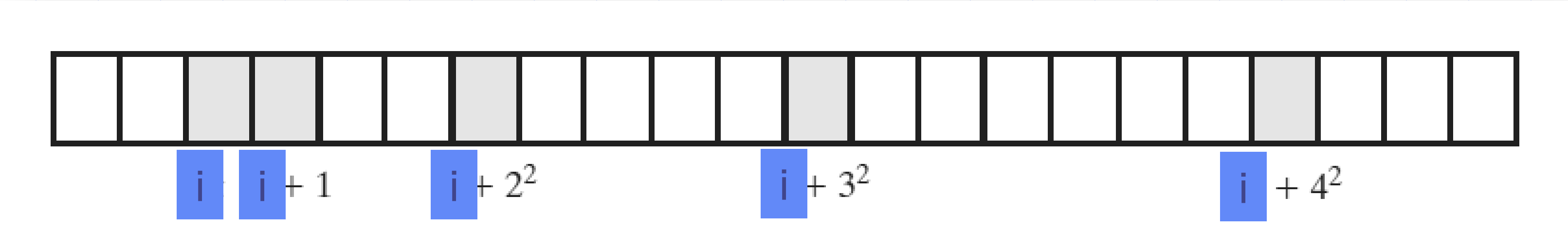


- $h(x) = x \bmod 13$
- delete 18:
  - i = 5 = h(18)
  - Found 18 at i, so set it to special token AVAIL (i.e. an empty slot but was occupied)
- delete 31
- insert 31 (where should 31 be inserted?)

Quadratic Probing

- Linear probing looks at consecutive locations beginning at index $i = h(x)$
- Quadratic probing, considers the locations at indices $i + j^2$
  - Uses the indices i, $i + 1$, $i + 4$, $i + 9$, …

Quadratic Probing



Quadratic Probing Pros and Cons

- Advantages
  - Avoids clustering
- Disadvantages
  - Creates secondary clustering

    A different pattern of filled array locations
  - An empty location may not be found even if one exists

    If the load factor is 0.5 or more

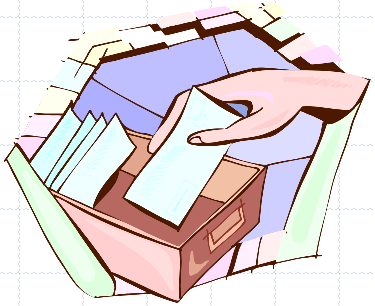
Double Hashing

- Linear and quadratic probing
  - Add increments to i to define a probe sequence
  - Both are <u>independent</u> of the search key
- Double hashing

  Computes increments via a second hash function

  => <u>key-dependent</u> method

Double Hashing



- Double hashing uses a secondary hash function $d_2(k)$ and handles collisions by placing an item in the first available cell of the series
  $$(i + j * d_2(k)) \bmod N$$
  for $j = 0, 1, \ldots, N-1$
- The secondary hash function $d_2(k)$ 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
- The possible values for $d_2(k)$ are
  $$1, 2, \ldots, q$$

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
- Use AVAIL to mark deleted entry

k	$h(k)$	$d(k)$	Probes (locations)		
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		

0	1	2	3	4	5	6	7	8	9	10	11	12

⇩

31		41			18	32	59	73	22	44		
0	1	2	3	4	5	6	7	8	9	10	11	12

Rehashing

- Maximum load factor, based on experimental data:
  - 0.5 for open addressing schemes
  - 0.9 for separate chaining
- If the load factor is above the threshold, then the table should be resized
  - New table should be at least double the old table so that the time cost can be amortized
  - Hash function should be modified
  - Rehash the data – take each item out of the old array and insert it into the new one using the new hash function

Comparison of Hash Table & BST

	BST	HashTable
Average Speed	$O(\log_2 N)$	$O(1)$
Find Min/Max	Yes	No
Items in a range	Yes	No
Sorted Input	Very Bad	No problem

Use HashTable if

1. there is any suspicion of SORTED input, and
2. NO ordering is required.

Performance of Hashing

Operation	List	Hash Table	
		expected	worst case
__getitem__	$O(n)$	$O(1)$	$O(n)$
__setitem__	$O(n)$	$O(1)$	$O(n)$
__delitem__	$O(n)$	$O(1)$	$O(n)$
__len__	$O(1)$	$O(1)$	$O(1)$
__iter__	$O(n)$	$O(n)$	$O(n)$

Coding: count word frequency

- Consider the problem of counting the number of occurrences of words in a document.
- Output the highest frequency word and number of times it appeared in the document.
- Download word_frequency_students.py and inputtext2.txt file from Brightspace
- Write your code in word_frequency_students.py file and submit to Gradescope