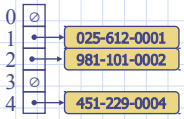
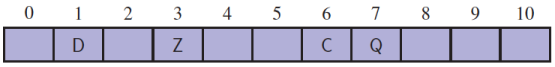
Hash Tables



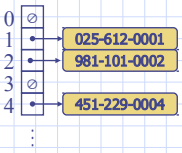
Recall the notion of a Map

- Intuitively, a map M supports the abstraction of using keys as indices with a syntax such as M[k].
- As a mental warm-up, consider a restricted setting in which a map with n items uses keys that are known to be integers in a range from 0 to N – 1, for some N ≥ n.

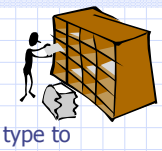


More General Kinds of Keys

- But what should we do if our keys are not integers in the range from 0 to N – 1?
  - Use a **hash function** to map general keys to corresponding indices in a table.
  - For instance, the last four digits of a Social Security number.



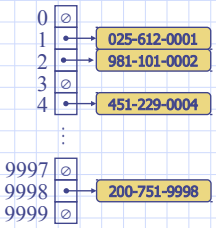
Hash Functions and Hash Tables



- A hash function h maps keys of a given type to integers in a fixed interval $[0, N - 1]$
- Example:
  $h(x) = x \bmod N$
  is a hash function for integer keys
- The integer $h(x)$ is called the hash value of key x
- A hash table for a given key type consists of
  - Hash function h
  - Array (called table) of size N
- When implementing a map with a hash table, the goal is to store item (k, o) at index $i = h(k)$

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 $h(x) =$ last four digits of x



Hash Functions

- A hash function is usually specified as the composition of two functions:
  - Hash code: h_1: keys → integers
  - Compression function: h_2: integers → $[0, N - 1]$
- 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 Codes

- Memory address:
  - We reinterpret the memory address of the key object as an integer Good in general, except for numeric and string keys
- Integer cast:
  - We reinterpret the bits of the key as an integer
  - Suitable for keys of length less than or equal to the number of bits of the integer
- Component sum:
  - We partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and we sum the components (ignoring overflows)
  - Suitable for numeric keys of fixed length greater than or equal to the number of bits of the integer type

Hash Codes (cont.)

- Polynomial accumulation:
  - We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits) $a_0 a_1 \ldots a_{n-1}$
  - 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, ignoring overflows
  - Especially suitable for strings (e.g., the choice $z = 33$ gives at most 6 collisions on a set of 50,000 English words)
- Polynomial $p(z)$ can be evaluated in $O(n)$ time using Horner's rule:
  - The following polynomials are successively computed, each from the previous one in $O(1)$ time
    $p_0(z) = a_{n-1}$
    $p_i(z) = a_{n-i-1} + z p_{i-1}(z)$ $(i = 1, 2, \ldots, n-1)$
- We have $p(z) = p_{n-1}(z)$

Compression Functions

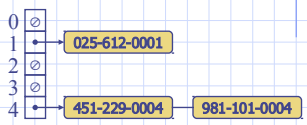
- Division:
  - $h_2(y) = y \bmod N$
  - The size N of the hash table is usually chosen to be a prime
  - The reason has to do with number theory and is beyond the scope of this course
- Multiply, Add and Divide (MAD):
  - $h_2(y) = (ay + b) \bmod N$
  - a and b are nonnegative integers such that $a \bmod N \neq 0$
  - Otherwise, every integer would map to the same value b

Abstract Hash Table Class

```
1  class HashMapBase(MapBase):
2    """Abstract base class for map using hash-table with MAD compression."""
3
4    def __init__(self, cap=11, p=109345121):
5      """Create an empty hash-table map."""
6      self._table = cap * [ None ]
7      self._n = 0                          # number of entries in the map
8      self._prime = p                      # prime for MAD compression
9      self._scale = 1 + randrange(p-1)     # scale from 1 to p-1 for MAD
10     self._shift = randrange(p)           # shift from 0 to p-1 for MAD
11
12   def _hash_function(self, k):
13     return (hash(k)*self._scale + self._shift) % self._prime % len(self._table)
14
15   def __len__(self):
16     return self._n
17
18   def __getitem__(self, k):
19     j = self._hash_function(k)
20     return self._bucket_getitem(j, k)    # may raise KeyError
21
22   def __setitem__(self, k, v):
23     j = self._hash_function(k)
24     self._bucket_setitem(j, k, v)        # subroutine maintains self._n
25     if self._n > len(self._table) // 2:  # keep load factor <= 0.5
26       self._resize(2 * len(self._table) - 1)   # number 2^x - 1 is often prime
27
28   def __delitem__(self, k):
29     j = self._hash_function(k)
30     self._bucket_delitem(j, k)           # may raise KeyError
31     self._n -= 1
32
33   def _resize(self, c):                  # resize bucket array to capacity c
34     old = list(self.items())             # use iteration to record existing items
35     self._table = c * [None]             # then reset table to desired capacity
36     self._n = 0                          # n recomputed during subsequent adds
37     for (k,v) in old:
38       self[k] = v                        # reinsert old key-value pair
```


Collision Handling

- 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(k):
return A[h(k)].get(k)

Algorithm put(k,v):
t = A[h(k)].put(k,v)
if t = **null then** {k is a new key}
  n = n + 1
return t

Algorithm remove(k):
t = A[h(k)].remove(k)
if t ≠ **null then** {k was found}
  n = n - 1
return t

Hash Table with Separate Chaining

```
class ChainHashMap(HashMapBase):
  """Hash map implemented with separate chaining for collision resolution."""

  def _bucket_getitem(self, j, k):
    bucket = self._table[j]
    if bucket is None:
      raise KeyError('Key Error: ' + repr(k))          # no match found
    return bucket[k]                                   # may raise KeyError

  def _bucket_setitem(self, j, k, v):
    if self._table[j] is None:
      self._table[j] = UnsortedTableMap( )             # bucket is new to the table
    oldsize = len(self._table[j])
    self._table[j][k] = v
    if len(self._table[j]) > oldsize:                  # key was new to the table
      self._n += 1                                     # increase overall map size

  def _bucket_delitem(self, j, k):
    bucket = self._table[j]
    if bucket is None:
      raise KeyError('Key Error: ' + repr(k))          # no match found
    del bucket[k]                                      # may raise KeyError

  def __iter__(self):
    for bucket in self._table:
      if bucket is not None:                           # a nonempty slot
        for key in bucket:
          yield key
```


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

- 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	

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 is found, or
    - N cells have been unsuccessfully probed

Algorithm *get*(k)
  $i \leftarrow h(k)$
  $p \leftarrow 0$
  repeat
    $c \leftarrow A[i]$
    if $c = \varnothing$
      return *null*
    else if $c.getKey() = k$
      return $c.getValue()$
    else
      $i \leftarrow (i + 1) \bmod N$
      $p \leftarrow p + 1$
  until $p = N$
  return *null*

Updates with Linear Probing

- To handle insertions and deletions, we introduce a special object, called *AVAILABLE*, which replaces deleted elements
- remove(k)
  - We search for an entry with key k
  - If such an entry (k, o) is found, we replace it with the special item *AVAILABLE* and we return element o
  - Else, we return *null*

- put(k, o)
  - We throw an exception if the table is full
  - We start at cell $h(k)$
  - We probe consecutive cells until one of the following occurs
    - A cell i is found that is either empty or stores *AVAILABLE*, or
    - N cells have been unsuccessfully probed
  - We store (k, o) in cell i

Hash Table with Linear Probing

```
class ProbeHashMap(HashMapBase):
  """Hash map implemented with linear probing for collision resolution."""
  _AVAIL = object( )        # sentinal marks locations of previous deletions

  def _is_available(self, j):
    """Return True if index j is available in table."""
    return self._table[j] is None or self._table[j] is ProbeHashMap._AVAIL

  def _find_slot(self, j, k):
    """Search for key k in bucket at index j.

    Return (success, index) tuple, described as follows:
    If match was found, success is True and index denotes its location.
    If no match found, success is False and index denotes first available slot.
    """
    firstAvail = None
    while True:
      if self._is_available(j):
        if firstAvail is None:
          firstAvail = j                    # mark this as first avail
        if self._table[j] is None:
          return (False, firstAvail)        # search has failed
      elif k == self._table[j]._key:
        return (True, j)                    # found a match
      j = (j + 1) % len(self._table)        # keep looking (cyclically)
```

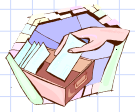
```
  def _bucket_getitem(self, j, k):
    found, s = self._find_slot(j, k)
    if not found:
      raise KeyError('Key Error: ' + repr(k))          # no match found
    return self._table[s]._value

  def _bucket_setitem(self, j, k, v):
    found, s = self._find_slot(j, k)
    if not found:
      self._table[s] = self._item(k,v)                 # insert new item
      self._n += 1                                     # size has increased
    else:
      self._table[s]._value = v                        # overwrite existing

  def _bucket_delitem(self, j, k):
    found, s = self._find_slot(j, k)
    if not found:
      raise KeyError('Key Error: ' + repr(k))          # no match found
    self._table[s] = ProbeHashMap._AVAIL               # mark as vacated

  def __iter__(self):
    for j in range(len(self._table)):                  # scan entire table
      if not self._is_available(j):
        yield self._table[j]._key
```


Double Hashing

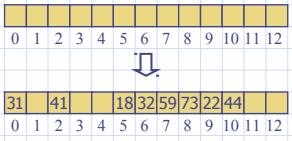


- Double hashing uses a secondary hash function $d(k)$ and handles collisions by placing an item in the first available cell of the series
  $(i + jd(k)) \bmod N$
  for $j = 0, 1, \ldots, N - 1$
- The secondary hash function $d(k)$ cannot have zero values
- The table size N must be a prime to allow probing 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
- The possible values for $d_2(k)$ are
  $1, 2, \ldots, q$

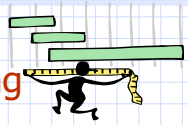
Example of Double Hashing

- 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

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		



Performance of Hashing



- 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 load factor $\alpha = n/N$ affects the performance of a hash table
- Assuming that the hash values are like random numbers, it can be shown that the expected number of probes for an insertion with open addressing is
  $1 / (1 - \alpha)$
- 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%
- Applications of hash tables:
  - small databases
  - compilers
  - browser caches