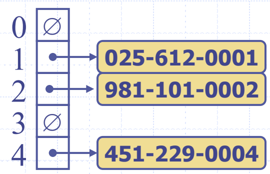
Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Hash Tables

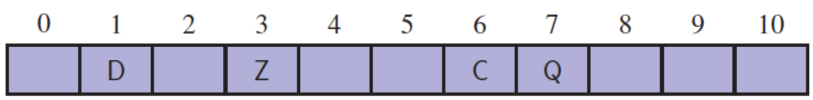


Recall the Map ADT

- get(k): if the map M has an entry with key k, return its associated value; else, return null
- put(k, v): insert entry (k, v) into the map M; if key k is not already in M, then return null; else, return old value associated with k
- remove(k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null
- size(), isEmpty()
- entrySet(): return an iterable collection of the entries in M
- keySet(): return an iterable collection of the keys in M
- values(): return an iterator of the values in M

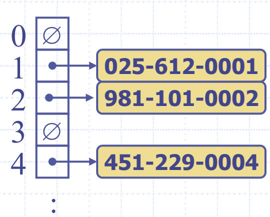
Intuitive Notion of a Map

- 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 $N \geq n$.

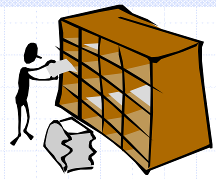


More General Kinds of Keys

- 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.



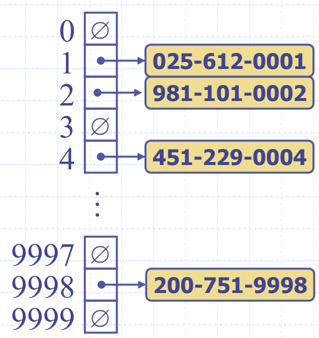
Hash Functions and Hash Tables



- 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 $i = h(k)$

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

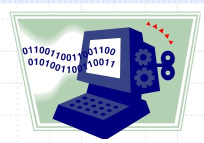
- A hash function is usually specified as the composition of two functions:

  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 Codes



- Memory address:
  - We reinterpret the memory address of the key object as an integer (default hash code of all Java objects)
  - 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 type (e.g., byte, short, int and float in Java)
- 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 (e.g., long and double in Java)

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

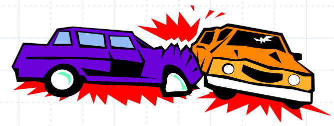
Abstract Hash Map in Java

```
public abstract class AbstractHashMap<K,V> extends AbstractMap<K,V> {
  protected int n = 0;                 // number of entries in the dictionary
  protected int capacity;              // length of the table
  private int prime;                   // prime factor
  private long scale, shift;           // the shift and scaling factors
  public AbstractHashMap(int cap, int p) {
    prime = p;
    capacity = cap;
    Random rand = new Random();
    scale = rand.nextInt(prime−1) + 1;
    shift = rand.nextInt(prime);
    createTable();
  }
  public AbstractHashMap(int cap) { this(cap, 109345121); }  // default prime
  public AbstractHashMap() { this(17); }                      // default capacity
  // public methods
  public int size() { return n; }
  public V get(K key) { return bucketGet(hashValue(key), key); }
  public V remove(K key) { return bucketRemove(hashValue(key), key); }
  public V put(K key, V value) {
    V answer = bucketPut(hashValue(key), key, value);
    if (n > capacity / 2)              // keep load factor <= 0.5
      resize(2 * capacity − 1);        // (or find a nearby prime)
    return answer;
  }
```

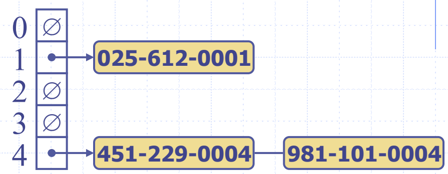

Abstract Hash Map in Java, 2

```
  // private utilities
  private int hashValue(K key) {
    return (int) ((Math.abs(key.hashCode()*scale + shift) % prime) % capacity);
  }
  private void resize(int newCap) {
    ArrayList<Entry<K,V>> buffer = new ArrayList<>(n);
    for (Entry<K,V> e : entrySet())
      buffer.add(e);
    capacity = newCap;
    createTable();                     // based on updated capacity
    n = 0;                             // will be recomputed while reinserting entries
    for (Entry<K,V> e : buffer)
      put(e.getKey(), e.getValue());
  }
  // protected abstract methods to be implemented by subclasses
  protected abstract void createTable();
  protected abstract V bucketGet(int h, K k);
  protected abstract V bucketPut(int h, K k, V v);
  protected abstract V bucketRemove(int h, K k);
}
```

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 Chaining

```
public class ChainHashMap<K,V> extends AbstractHashMap<K,V> {
  // a fixed capacity array of UnsortedTableMap that serve as buckets
  private UnsortedTableMap<K,V>[ ] table;   // initialized within createTable
  public ChainHashMap( ) { super( ); }
  public ChainHashMap(int cap) { super(cap); }
  public ChainHashMap(int cap, int p) { super(cap, p); }
  /** Creates an empty table having length equal to current capacity. */
  protected void createTable( ) {
    table = (UnsortedTableMap<K,V>[ ]) new UnsortedTableMap[capacity];
  }
  /** Returns value associated with key k in bucket with hash value h, or else null. */
  protected V bucketGet(int h, K k) {
    UnsortedTableMap<K,V> bucket = table[h];
    if (bucket == null) return null;
    return bucket.get(k);
  }
  /** Associates key k with value v in bucket with hash value h; returns old value. */
  protected V bucketPut(int h, K k, V v) {
    UnsortedTableMap<K,V> bucket = table[h];
    if (bucket == null)
      bucket = table[h] = new UnsortedTableMap<>( );
    int oldSize = bucket.size( );
    V answer = bucket.put(k,v);
    n += (bucket.size( ) − oldSize);      // size may have increased
    return answer;
  }
```


Hash Table with Chaining, 2

```
  /** Removes entry having key k from bucket with hash value h (if any). */
  protected V bucketRemove(int h, K k) {
    UnsortedTableMap<K,V> bucket = table[h];
    if (bucket == null) return null;
    int oldSize = bucket.size( );
    V answer = bucket.remove(k);
    n −= (oldSize − bucket.size( ));      // size may have decreased
    return answer;
  }
  /** Returns an iterable collection of all key-value entries of the map. */
  public Iterable<Entry<K,V>> entrySet( ) {
    ArrayList<Entry<K,V>> buffer = new ArrayList<>( );
    for (int h=0; h < capacity; h++)
      if (table[h] != null)
        for (Entry<K,V> entry : table[h].entrySet( ))
          buffer.add(entry);
    return buffer;
  }
}
```


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

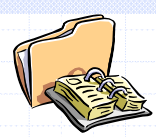
- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

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

⇩

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

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 ← h(k)
  p ← 0
  repeat
    c ← A[i]
    if c = ∅
      return null
    else if c.getKey () = k
      return c.getValue()
    else
      i ← (i + 1) mod N
      p ← p + 1
  until  p = N
  return null
```


Updates with Linear Probing

- To handle insertions and deletions, we introduce a special object, called ***DEFUNCT***, 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 ***DEFUNCT*** 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 ***DEFUNCT***, or
    - N cells have been unsuccessfully probed
  - We store (k, o) in cell i

Probe Hash Map in Java

```
public class ProbeHashMap<K,V> extends AbstractHashMap<K,V> {
  private MapEntry<K,V>[ ] table;        // a fixed array of entries (all initially null)
  private MapEntry<K,V> DEFUNCT = new MapEntry<>(null, null);   //sentinel
  public ProbeHashMap( ) { super( ); }
  public ProbeHashMap(int cap) { super(cap); }
  public ProbeHashMap(int cap, int p) { super(cap, p); }
  /** Creates an empty table having length equal to current capacity. */
  protected void createTable( ) {
    table = (MapEntry<K,V>[ ]) new MapEntry[capacity];   // safe cast
  }
  /** Returns true if location is either empty or the "defunct" sentinel. */
  private boolean isAvailable(int j) {
    return (table[j] == null || table[j] == DEFUNCT);
  }
```

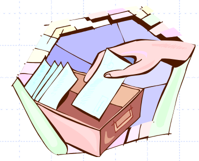

Probe Hash Map in Java, 2

```java
/** Returns index with key k, or −(a+1) such that k could be added at index a. */
private int findSlot(int h, K k) {
  int avail = −1;                           // no slot available (thus far)
  int j = h;                                // index while scanning table
  do {
    if (isAvailable(j)) {                   // may be either empty or defunct
      if (avail == −1) avail = j;           // this is the first available slot!
      if (table[j] == null) break;          // if empty, search fails immediately
    } else if (table[j].getKey().equals(k))
      return j;                             // successful match
    j = (j+1) % capacity;                   // keep looking (cyclically)
  } while (j != h);                         // stop if we return to the start
  return −(avail + 1);                      // search has failed
}
/** Returns value associated with key k in bucket with hash value h, or else null. */
protected V bucketGet(int h, K k) {
  int j = findSlot(h, k);
  if (j < 0) return null;                   // no match found
  return table[j].getValue();
}
```


Probe Hash Map in Java, 3

```java
/** Associates key k with value v in bucket with hash value h; returns old value. */
protected V bucketPut(int h, K k, V v) {
  int j = findSlot(h, k);
  if (j >= 0)                               // this key has an existing entry
    return table[j].setValue(v);
  table[−(j+1)] = new MapEntry<>(k, v);     // convert to proper index
  n++;
  return null;
}
/** Removes entry having key k from bucket with hash value h (if any). */
protected V bucketRemove(int h, K k) {
  int j = findSlot(h, k);
  if (j < 0) return null;                   // nothing to remove
  V answer = table[j].getValue();
  table[j] = DEFUNCT;                       // mark this slot as deactivated
  n−−;
  return answer;
}
/** Returns an iterable collection of all key-value entries of the map. */
public Iterable<Entry<K,V>> entrySet() {
  ArrayList<Entry<K,V>> buffer = new ArrayList<>();
  for (int h=0; h < capacity; h++)
    if (!isAvailable(h)) buffer.add(table[h]);
  return buffer;
}
}
```


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 $j = 0, 1, \dots, N-1$
- 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
  $$1, 2, \dots, q$$

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

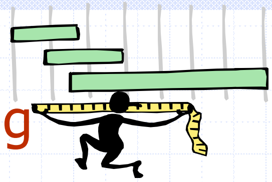
k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		

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

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