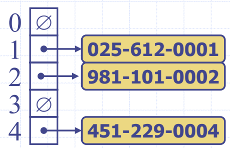
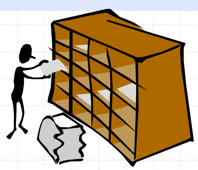
Hash Tables



Recall the Map ADT (§ 8.1)

- Map ADT methods:
  - get(k): if the map M has an entry with key k, return its assoiciated 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()
  - keys(): return an iterator of the keys in M
  - values(): return an iterator of the values in M

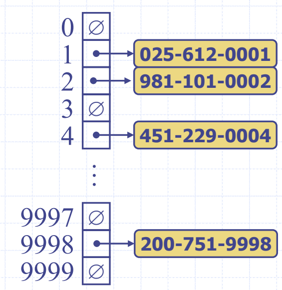
Hash Functions and Hash Tables (§ 8.2)



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



Hash Functions (§ 8.2.2)

- 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 (§ 8.2.3)

- 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

    $p_0(z) = a_{n-1}$

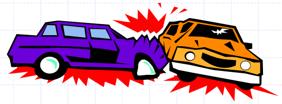
    $p_i(z) = a_{n-i-1} + z p_{i-1}(z)$
    $(i = 1, 2, \ldots, n-1)$
- We have $p(z) = p_{n-1}(z)$

Compression Functions (§ 8.2.4)

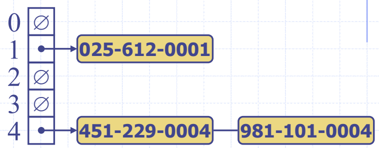
- Division:
  - $h_2(y) = y \bmod N$
  - The size N of the hash table is usually chosen to be a prime
  - The reason has to do with number theory and is beyond the scope of this course
- Multiply, Add and Divide (MAD):
  - $h_2(y) = (ay + b) \bmod N$
  - a and b are nonnegative integers such that

    $a \bmod N \neq 0$
  - Otherwise, every integer would map to the same value b

Collision Handling (§ 8.2.5)



- 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 Methods with Separate Chaining used for Collisions

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

Algorithm get(*k*):
Output: The value associated with the key *k* in the map, or **null** if there is no entry with key equal to *k* in the map
return *A*[*h*(*k*)].get(*k*) {delegate the get to the list-based map at *A*[*h*(*k*)]}

Algorithm put(*k*,*v*):
Output: If there is an existing entry in our map with key equal to *k*, then we return its value (replacing it with *v*); otherwise, we return **null**
t = *A*[*h*(*k*)].put(*k*,*v*) {delegate the put to the list-based map at *A*[*h*(*k*)]}
if *t* = **null then** {*k* is a new key}
  n = *n* + 1
return *t*

Algorithm remove(*k*):
Output: The (removed) value associated with key *k* in the map, or **null** if there is no entry with key equal to *k* in the map
t = *A*[*h*(*k*)].remove(*k*) {delegate the remove to the list-based map at *A*[*h*(*k*)]}
if *t* ≠ **null then** {*k* was found}
  n = *n* - 1
return *t*

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

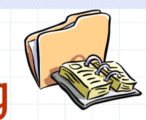
- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

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



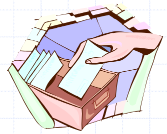
- Consider a hash table A that uses linear probing
- get(k)
  - We start at cell $h(k)$
  - We probe consecutive locations until one of the following occurs
    - An item with key k is found, or
    - An empty cell is found, or
    - N cells have been unsuccessfully probed

```
Algorithm get(k)
  i ← h(k)
  p ← 0
  repeat
    c ← A[i]
    if c = ∅
      return null
    else if c.key () = k
      return c.element()
    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 ***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 entry (k, o) in cell i

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

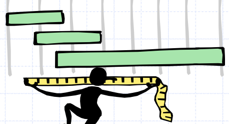
k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		

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

Java Example

```
/** A hash table with linear probing and the MAD hash function */
public class HashTable implements Map {
  protected static class HashEntry implements Entry {
    Object key, value;
    HashEntry () { /* default constructor */ }
    HashEntry(Object k, Object v) { key = k; value = v; }
    public Object key() { return key; }
    public Object value() { return value; }
    protected Object setValue(Object v) {  // set a new value, returning old
      Object temp = value;
      value = v;
      return temp;  // return old value
    }
  }
  /** Nested class for a default equality tester */
  protected static class DefaultEqualityTester implements EqualityTester {
    DefaultEqualityTester() { /* default constructor */ }
    /** Returns whether the two objects are equal.  */
    public boolean isEqualTo(Object a, Object b) { return a.equals(b); }
  }
  protected static Entry AVAILABLE = new HashEntry(null, null); // empty
        marker
  protected int n = 0;                    // number of entries in the dictionary
  protected int N;                        // capacity of the bucket array
  protected Entry[] A;                                  // bucket array
  protected EqualityTester T;             // the equality tester
  protected int scale, shift;   // the shift and scaling factors
  /** Creates a hash table with initial capacity 1023. */
  public HashTable() {
    N = 1023; // default capacity
    A = new Entry[N];
    T = new DefaultEqualityTester(); // use the default equality tester
    java.util.Random rand = new java.util.Random();
    scale = rand.nextInt(N-1) + 1;
    shift = rand.nextInt(N);
  }
/** Creates a hash table with the given capacity and equality tester. */
  public HashTable(int bN, EqualityTester tester) {
    N = bN;
    A = new Entry[N];
    T = tester;
    java.util.Random rand = new java.util.Random();
    scale = rand.nextInt(N-1) + 1;
    shift = rand.nextInt(N);
  }
```


Java Example (cont.)

```
/** Determines whether a key is valid. */
protected void checkKey(Object k) {
  if (k == null) throw new InvalidKeyException("Invalid key: null.");
}
/** Hash function applying MAD method to default hash code. */
public int hashValue(Object key) {
  return Math.abs(key.hashCode()*scale + shift) % N;
}
/** Returns the number of entries in the hash table. */
public int size() { return n; }
/** Returns whether or not the table is empty. */
public boolean isEmpty() { return (n == 0); }
/** Helper search method - returns index of found key or -index-1,
 * where index is the index of an empty or available slot. */
protected int findEntry(Object key) throws InvalidKeyException {
  int avail = 0;
  checkKey(key);
  int i = hashValue(key);
  int j = i;
  do {
    if (A[i] == null)   return -i - 1;  // entry is not found
    if (A[i] == AVAILABLE) {                 // bucket is deactivated
        avail = i;                           // remember that this slot is available
        i = (i + 1) % N;                     // keep looking
    }
    else if (T.isEqualTo(key,A[i].key()))  // we have found our entry
        return i;
    else // this slot is occupied--we must keep looking
        i = (i + 1) % N;
  } while (i != j);
  return -avail - 1;  // entry is not found
}
/** Returns the value associated with a key. */
public Object get (Object key) throws InvalidKeyException {
  int i = findEntry(key);  // helper method for finding a key
  if (i < 0) return null;  // there is no value for this key
  return A[i].value();     // return the found value in this case
}
```

```
/** Put a key-value pair in the map, replacing previous one if it exists. */
public Object put (Object key, Object value) throws InvalidKeyException {
  if (n >= N/2) rehash(); // rehash to keep the load factor <= 0.5
  int i = findEntry(key); //find the appropriate spot for this entry
  if (i < 0) {       // this key does not already have a value
    A[-i-1] = new HashEntry(key, value); // convert to the proper index
    n++;
    return null;    // there was no previous value
  }
  else                                    // this key has a previous value
    return ((HashEntry) A[i]).setValue(value); // set new value & return old
}
/** Doubles the size of the hash table and rehashes all the entries. */
protected void rehash() {
  N = 2*N;
  Entry[] B = A;
  A = new Entry[N]; // allocate a new version of A twice as big as before
  java.util.Random rand = new java.util.Random();
  scale = rand.nextInt(N-1) + 1;                          // new hash scaling factor
  shift = rand.nextInt(N);                                // new hash shifting factor
  for (int i=0; i&ltB.length; i++)
    if ((B[i] != null) && (B[i] != AVAILABLE)) { // if we have a valid entry
      int j = findEntry(B[i].key());  // find the appropriate spot
      A[-j-1] = B[i];              // copy into the new array
    }
}
/** Removes the key-value pair with a specified key. */
public Object remove (Object key) throws InvalidKeyException {
  int i = findEntry(key);               // find this key first
  if (i < 0) return null;               // nothing to remove
  Object toReturn = A[i].value();
  A[i] = AVAILABLE;                                       // mark this slot as
     deactivated
  n--;
  return toReturn;
}
/** Returns an iterator of keys. */
public java.util.Iterator keys() {
  List keys = new NodeList();
  for (int i=0; i&ltN; i++)
    if ((A[i] != null) && (A[i] != AVAILABLE))
      keys.insertLast(A[i].key());
  return keys.elements();
}
} // ... values() is similar to keys() and is omitted here ...
```