Recall the Map ADT (§ 8.1)

Map ADT methods:
- \textbf{get}(k): if the map M has an entry with key k, return its associated value; else, return null
- \textbf{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
- \textbf{remove}(k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null
- \textbf{size()}, \textbf{isEmpty}()
- \textbf{keys}(): return an iterator of the keys in M
- \textbf{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:
\(h(x) = x \text{ mod } N\)
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 \((k, a)\) 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
\[h(x) = \text{last four digits of } x\]
A hash function is usually specified as the composition of two functions:

**Hash code:**

- $h_1: \text{keys} \rightarrow \text{integers}$

**Compression function:**

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

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

- **Component sum:**
  - We partition the bits of the key into components of fixed length (e.g., byte, short, int and float in Java)
  - 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)

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

- **Division:**
  - $h_2(y) = y \mod N$
  - 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):**
  - $h_2(y) = (ay + b) \mod N$
  - $a$ and $b$ are nonnegative integers such that $a \mod N \neq 0$
  - 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`
    - **Return:** `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
    - **Return:** `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 mod 13**
- Insert keys `18, 41, 22, 44, 59, 32, 31, 73`, in this order

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 c.element()
  else if c.key = k
    return c
  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.
- To search for an entry with key \( k \),
- 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 \( i \).
- Else, we return \textit{null}.

\[ \text{put}(k, o) \]

- We throw an exception if the table is full.
- We search for an entry with key \( k \).
- 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 \( i \).
- Else, we return \textit{null}.

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 \((i + jd(k)) \mod N\) 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.

Example of Double Hashing

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

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 \( 1 / (1 - \alpha) \).
- In practice, hashing is very fast provided the load factor is not close to 100%.
- Applications of hash tables:
  - small databases
  - compilers
  - browser caches
**A hash table with linear probing and the MAD hash function.**

```java
/** A hash table with linear probing and the MAD hash function. */
public class HashTable implements Map {
    //... (code continues)
}
```

**Nested class for a default equality tester.**

```java
/** Nested class for a default equality tester. */
protected static class DefaultEqualityTester implements EqualityTester {
    //... (code continues)
}
```

**Determines whether a key is valid.**

```java
/** Determines whether a key is valid. */
protected void checkKey(Object k) {
    if (k == null) throw new InvalidKeyException("Invalid key: null.");
    //... (code continues)
}
```

**Inserts a key-value pair in the map, replacing previous one if it exists.**

```java
/** Insert an arbitrary Key/Value pair into the hash table. */
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
    }
    //... (code continues)
}
```

**Creates a hash table with the given capacity and equality tester.**

```java
/** Creates a hash table with the given capacity and equality tester. */
public HashTable(int bN, EqualityTester tester) {
    //... (code continues)
}
```

**Returns an iterator of keys.**

```java
/** Returns an iterator of keys. */
public java.util.Iterator keys() {
    List keys = new NodeList();
    for (int i=0; i&ltN; i++)
        if ((B[i] != null) && (B[i] != AVAILABLE)) { // if we have a valid entry
            if (T.isEqualTo(key,B[i].key()))  // we have found our entry
                keys.insertLast(A[i].key());
            else if (B[i].key() != null) { // if the entry is not null
                //... (code continues)
        }
    //... (code continues)
}
```

© 2004 Goodrich, Tamassia

---

**Java Example**

**Java Example (cont.)**