Hash Tables

```c
int getRandomNumber()
{
    return 4; // chosen by fair dice roll.
    // guaranteed to be random.
}
```

xkcd. http://xkcd.com/221/. “Random Number.” Used with permission under Creative Commons 2.5 License.
Recall the Map Operations

- **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\()**
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 $N \geq n$. 

```
0 1 2 3 4 5 6 7 8 9 10
D Z C Q
```
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.

```
0  Ø
1  025-612-0001
2  981-101-0002
3  Ø
4  451-229-0004
...```

© 2015 Goodrich and Tamassia
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:
  \[ h(x) = x \mod N \]
  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 $(k, o)$ 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 $h(x) = \text{last four digits of } x$.
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. 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).

- **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
    \[ p_0(z) = a_{n-1} \]
    \[ p_i(z) = a_{n-i-1} + zp_{i-1}(z) \]  
    \( (i = 1, 2, \ldots, n-1) \)

- We have \( p(z) = p_{n-1}(z) \)
**Tabulation-Based Hashing**

- Suppose each key can be viewed as a tuple, \( k = (x_1, x_2, \ldots, x_d) \), for a fixed \( d \), where each \( x_i \) is in the range \([0, M - 1]\).

- There is a class of hash functions we can use, which involve simple table lookups, known as **tabulation-based hashing**.

- We can initialize \( d \) tables, \( T_1, T_2, \ldots, T_d \), of size \( M \) each, so that each \( T_i[j] \) is a uniformly chosen independent random number in the range \([0, N - 1]\).

- We then can compute the hash function, \( h(k) \), as

\[
h(k) = T_1[x_1] \oplus T_2[x_2] \oplus \ldots \oplus T_d[x_d],
\]

where “\( \oplus \)” denotes the bitwise exclusive-or function.

- Because the values in the tables are themselves chosen at random, such a function is itself fairly random. For instance, it can be shown that such a function will cause two distinct keys to collide at the same hash value with probability \( 1/N \), which is what we would get from a perfectly random function.
Compression Functions

- **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

- **Random linear hash function:**
  - \( h_2(y) = (ay + b) \mod N \)
  - \( a \) and \( b \) are random nonnegative integers such that \( a \mod N \neq 0 \)
  - Otherwise, every integer would map to the same value \( b \)
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 $t = \text{null}$ then  
---  
$n = n + 1$  
---  
return $t$

**Algorithm** `remove(k)`:  
$t = A[h(k)].remove(k)$  
if $t \neq \text{null}$ then  
---  
$n = n - 1$  
---  
return $t$  

{k is a new key}  

{k was found}
Performance of Separate Chaining

- Let us assume that our hash function, $h$, maps keys to independent uniform random values in the range $[0,N-1]$.

- Thus, if we let $X$ be a random variable representing the number of items that map to a bucket, $i$, in the array $A$, then the expected value of $X$, $E(X) = n/N$, where $n$ is the number of items in the map, since each of the $N$ locations in $A$ is equally likely for each item to be placed.

- This parameter, $n/N$, which is the ratio of the number of items in a hash table, $n$, and the capacity of the table, $N$, is called the **load factor** of the hash table.

- If it is $O(1)$, then the above analysis says that the expected time for hash table operations is $O(1)$ when collisions are handled with separate chaining.
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($k$)

```java
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.

- `put(k, v)`
  - We throw an exception if the table is full.
  - We start at cell `h(k)`.
  - We probe consecutive cells until a cell `i` is found that is empty.
    - We store `(k, v)` in cell `i`.

- `remove(k)`
  - We search for an entry with key `k`.
  - If such an entry, `(k, v)`, is found, we move elements to fill the “hole” created by its removal.
Pseudo-code for get and put

- get(k):
  
i \leftarrow h(k)
  
  while \( A[i] \neq \text{NULL} \) do
    if \( A[i].\text{key} = k \) then
      return \( A[i] \)
      i \leftarrow (i + 1) \text{ mod } N
  
  return \text{NULL}

- put(k, v):
  
i \leftarrow h(k)
  
  while \( A[i] \neq \text{NULL} \) do
    if \( A[i].\text{key} = k \) then
      \( A[i] \leftarrow (k, v) \)  // replace the old \((k, v')\)
      i \leftarrow (i + 1) \text{ mod } N
  
  \( A[i] \leftarrow (k, v) \)
Pseudo-code for remove

- remove($k$):
  
  ```
  i ← h($k$)
  while $A[i] \neq$ NULL do
    if $A[i].key = k$ then
      temp ← $A[i]$
      $A[i] ←$ NULL
      Call Shift($i$) to restore $A$ to a stable state without $k$
    
    return temp
  
  $i ← (i + 1) \mod N$

  return NULL
  ```

- Shift($i$):

  ```
  $s ← 1$ // the current shift amount
  while $A[(i + s) \mod N] \neq$ NULL do
    $j ← h(A[(i + s) \mod N].key)$ // preferred index for this item
    if $j \notin (i, i + s] \mod N$ then
      $A[i] ← A[(i + s) \mod N]$ // fill in the “hole”
      $A[(i + s) \mod N] ←$ NULL // move the “hole”
      $i ← (i + s) \mod N$
      $s ← 1$
    else
      $s ← s + 1$
  ```
Performance of Linear Probing

- 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 $O(1)$ with constant load $< 1$.
- In practice, hashing is very fast provided the load factor is not close to 100%.
- Applications of hash tables:
  - small databases
  - compilers
  - browser caches
A More Careful Analysis of Linear Probing

- Recall that, in the linear-probing scheme for handling collisions, whenever an insertion at a cell $i$ would cause a collision, then we instead insert the new item in the first cell of $i+1$, $i+2$, and so on, until we find an empty cell.

Let $X_1, X_2, \ldots, X_n$ be a set of mutually independent indicator random variables, such that each $X_i$ is 1 with some probability $p_i > 0$ and 0 otherwise. Let $X = \sum_{i=1}^{n} X_i$ be the sum of these random variables, and let $\mu$ denote the mean of $X$, that is, $\mu = E(X) = \sum_{i=1}^{n} p_i$. The following bound, which is due to Chernoff (and which we derive in Section 19.5), establishes that, for $\delta > 0$,

$$P_r(X > (1 + \delta)\mu) < \left[\frac{e^{\delta}}{(1 + \delta)(1+\delta)}\right]^\mu.$$  

- For this analysis, let us assume that we are storing $n$ items in a hash table of size $N = 2n$, that is, our hash table has a load factor of $1/2$. 

© 2015 Goodrich and Tamassia

Hash Tables
A More Careful Analysis of Linear Probing, 2

Let $X$ denote a random variable equal to the number of probes that we would perform in doing a search or update operation in our hash table for some key, $k$. Furthermore, let $X_i$ be a 0/1 indicator random variable that is 1 if and only if $i = h(k)$, and let $Y_i$ be a random variable that is equal to the length of a run of contiguous nonempty cells that begins at position $i$, wrapping around the end of the table if necessary. By the way that linear probing works, and because we assume that our hash function $h(k)$ is random,

$$X = \sum_{i=0}^{N-1} X_i(Y_i + 1),$$

which implies that

$$E(X) = \sum_{i=0}^{N-1} \frac{1}{2n} E(Y_i + 1) = 1 + (1/2n)E\left(\sum_{i=0}^{N-1} Y_i\right).$$

- Thus, if we can bound the expected value of the sum of $Y_i$’s, then we can bound the expected time for a search or update operation in a linear-probing hashing scheme.
A More Careful Analysis of Linear Probing, 2

Let $X$ denote a random variable equal to the number of probes that we would perform in doing a search or update operation in our hash table for some key, $k$. Furthermore, let $X_i$ be a 0/1 indicator random variable that is 1 if and only if $i = h(k)$, and let $Y_i$ be a random variable that is equal to the length of a run of contiguous nonempty cells that begins at position $i$, wrapping around the end of the table if necessary. By the way that linear probing works, and because we assume that our hash function $h(k)$ is random,

$$X = \sum_{i=0}^{N-1} X_i(Y_i + 1),$$

which implies that

$$E(X) = \sum_{i=0}^{N-1} \frac{1}{2n} E(Y_i + 1)$$

$$= 1 + (1/2n)E \left( \sum_{i=0}^{N-1} Y_i \right).$$

- Thus, if we can bound the expected value of the sum of $Y_i$'s, then we can bound the expected time for a search or update operation in a linear-probing hashing scheme.
Consider, then, a maximal contiguous sequence, $S$, of $k$ nonempty table cells, that is, a contiguous group of occupied cells that has empty cells next to its opposite ends. Any search or update operation that lands in $S$ will, in the worst case, march all the way to the end of $S$. That is, if a search lands in the first cell of $S$, it would make $k$ wasted probes, if it lands in the second cell of $S$, it would make $k - 1$ wasted probes, and so on. So the total cost of all the searches that land in $S$ can be at most $k^2$. Thus, if we let $Z_{i,k}$ be a 0/1 indicator random variable for the existence of a maximal sequence of nonempty cells of length $k$, then

$$
\sum_{i=0}^{N-1} Y_i \leq \sum_{i=0}^{N-1} \sum_{k=1}^{2n} k^2 Z_{i,k}.
$$

Put another way, it is as if we are “charging” each maximal sequence of nonempty cells for all the searches that land in that sequence.
A More Careful Analysis of Linear Probing, 4

So, to bound the expected value of the sum of the \( Y_i \)'s, we need to bound the probability that \( Z_{i,k} \) is 1, which is something we can do using the Chernoff bound given above. Let \( Z_k \) denote the number of items that are mapped to a given sequence of \( k \) cells in our table. Then,

\[
\Pr(Z_{i,k} = 1) \leq \Pr(Z_k \geq k).
\]

Because the load factor of our table is 1/2, \( E(Z_k) = k/2 \). Thus, by the above Chernoff bound,

\[
\Pr(Z_k \geq k) = \Pr(Z_k \geq 2(k/2)) \leq (e/4)^{k/2} < 2^{-k/4}.
\]

Therefore, putting all the above pieces together,

\[
E(X) = 1 + \frac{1}{2n} E \left( \sum_{i=0}^{N-1} Y_i \right)
\]

\[
\leq 1 + \frac{1}{2n} \sum_{i=0}^{N-1} \sum_{k=1}^{2n} k^2 2^{-k/4}
\]

\[
\leq 1 + \sum_{k=1}^{\infty} k^2 2^{-k/4} = O(1).
\]

That is, the expected running time for doing a search or update operation with linear probing is \( O(1) \), so long as the load factor in our hash table is at most 1/2.
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

- Common choice of compression function for the secondary hash function:
  \[
  d_2(k) = q - k \mod q
  \]
  where
  - \( q < N \)
  - \( q \) is a prime

- The possible values for \( d_2(k) \) are
  \[
  1, 2, \ldots, q \]
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.