# CMSC 132: OBJECT-ORIENTED PROGRAMMING II

## Hashing

Department of Computer Science
University of Maryland, College Park

# Announcements

- Video “What most schools don’t teach”
  - http://www.youtube.com/watch?v=nKIu9yen5nc

# Introduction

- If you need to find a value in a list what is the most efficient way to perform the search?
  - Linear search
  - Binary search
  - Can we have O(1)?

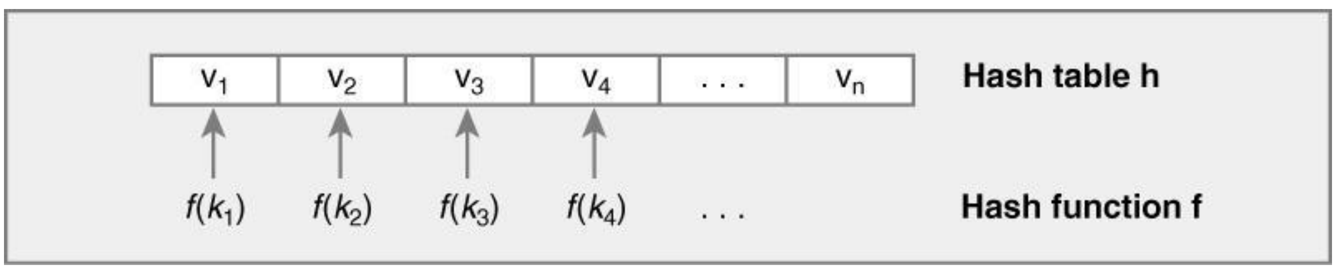
# Hashing

- Remember that modulus allows us to map a number to a range
  - X % N → X mapped to value between 0 and N - 1
- Suppose you have 4 parking spaces and need to assign each resident a space. How can we do it?

  parkingSpace(ssn) = ssn % 4
- Problems??
  - What if two residents are assigned the same spot? Collission!
- What if we want to use name instead of ssn?
  - Generate integer out of the name
- We just described hashing

# Hashing

- **Hashing**
  - Technique for storing key-value entries into an **array**
    - In Java we will have an array of Objects where each Object has a key (e.g., student's name) and a reference to data of interest (e.g., student's grades)
  - The **array** is called the **hash table**
  - **Ideally** can result in O(1) search times
- **Hash Function**
  - Takes a search key ($K_i$) and returns a location in the array (an integer index (**hash index**))
  - A search key maps (hashes) to index i




- **Ideal Hash Function**
  - If every search key corresponds to a unique element in the hash table

# Hashing

- If we have a large range of possible search keys, but a subset of them are used, allocating a large table would a waste of significant space
- **Typical hash function (two steps)**
  1. Transforms a search key to an integer value called the **hash code.** For example, for a string we can add Unicode values to generate a **hash code**
  2. Compress the **hash code** so it lies within the range of indices for the **hash table.** Using the modulus operator (%) we can compress the **hash code** in order to generate the **hash index** (location in the table)
- **Collision**
  - Takes place when two or more search keys map to the hash table entry
- **Good Hash Function**
  - Fast to compute
  - Minimizes Collisions
    - Using a function that distributes values uniformly reduces probability of collisions

# Hash Codes

- **You can generate a hash code for a string**
  - By adding Unicode values
  - Better approach - Multiplying Unicode value of each character by a factor that depends on the character's position in the string
- For primitive types
  - If the key is an **int**, use the key
  - If **char**, **short**, **byte**, cast to **int**
  - If **long**, **float, double** manipulate the internal binary representation
- **Example:**

```
System.out.println("Java".hashCode());  // prints 2301506
How did they get this?
Ascii for J is 74, a is 97, and v is 118
74 *(31)^3 + 97 *(31)^2 + 118 * 31 + 97  = 2301506
```

# Scaling (Compressing) hash code

- Using the modulus operator, we can compress an integer to lie within a given range of values. If **n** is the table size

  **remainder (hash index/compressed hash code) = hash code % n**

  **remainder lies in the range [0, n – 1]**

- Selecting table size (**n**)
  - If **n** is even, the **compressed hash code** will have the same parity as the **hash code** (if hash code is odd, result is odd; if even, even)
  - **Many indices of the table will be left out if n is even**
    - **Size of the hash table should be odd**
  - When **n** is a prime number, **hash code % n** provides values that are distributed throughout the range [0, **n** – 1]
  - **Size of a hash table should be a prime number n greater than 2**

# Hash Function

- Example (generating **hash indices**)

  hash("apple") = 5
  hash("watermelon") = 3
  hash("kiwi") = 0
  hash("mango") = 6
  hash("banana") = 2

- Perfect hash function
  - Unique values for each key

| | |
|---|---|
| 0 | kiwi |
| 1 | |
| 2 | banana |
| 3 | watermelon |
| 4 | |
| 5 | apple |
| 6 | mango |
| 7 | |

# Hash Function

- Suppose now

  hash("apple") = 5
  hash("watermelon") = 3
  hash("kiwi") = 0
  hash("mango") = 6
  hash("banana") = 2
  hash(“orange") = 3

- Collision
  - Same hash index for multiple keys

| | |
|---|---|
| 0 | kiwi |
| 1 | |
| 2 | banana |
| 3 | watermelon |
| 4 | |
| 5 | apple |
| 6 | mango |
| 7 | |

# Resolving Collisions

- **Choice #1**
  - Look for an unused entry in the table
  - This technique is referred to as **open addressing**
- **Choice #2**
  - Each element in the table can be associated with more than one search key
    - Each element now becomes a bucket (e.g., a list)
    - This technique is referred to as **separate chaining**

# Resolving Collisions (Open Addressing)

- **Probing** → locating an open element/position in the hash table
- **Open addressing** has several variations depending on the next position (increment) to use to resolve the collision
  - **Linear probing** →When a collision occurs at index position **k,** we see whether position **k + 1** is available (not in use). If it is in use, we look at **k + 2** and so on, wrapping around to the beginning of the table if necessary
    - **Probe sequence** → table elements considered in a search
  - **Quadratic probing →** Considers elements at indices $k + j^2$ (e.g., k + 1, k + 4, k + 9, etc.) wrapping around if necessary
  - **Double Hashing** →The increment of 1 for linear probing and $j^2$ for quadratic, is replaced with the result of a second hash function that determines the increment

# Open Addressing Summary

- **Search →** searches the probe sequence for the key, examining elements that are present and ignoring ***Removed*** entries. Search stops when element is found or ***NeverUsed*** is reached
- **Remove →** performs a search and if it finds the key it marks the element as ***Removed***
- **Insertion →** searches the probe sequence, keeping track of the first element that is in the ***Removed*** or ***NeverUsed*** state. If the key is not found, it is placed in the first element that was in the ***Removed*** or ***NeverUsed*** state

# Insertion: Open Addressing (Linear Probing)

- Table states: Occupied, NeverUsed, Removed
- Suppose now

  hash("apple") = 5
  hash("watermelon") = 3
  hash("kiwi") = 0
  hash("mango") = 6
  hash("banana") = 2
  hash("orange") = 3
  hash("pear") = 3

- **Insertion of orange and pear**
  - Same hash index for multiple keys (orange and pear)
  - Using linear probing we find next available position and insert element
- **Searching after insertion (watermelon, orange and pear)**
  - Hash search key. If element found at hash index, stop; otherwise, search forward until element found or ***NeverUsed*** seen (element not found)

| | |
|---|---|
| 0 | kiwi |
| 1 | *NeverUsed* |
| 2 | banana |
| 3 | watermelon |
| 4 | ~~NeverUsed~~ orange |
| 5 | apple |
| 6 | mango |
| 7 | ~~NeverUsed~~ pear |

# Removal: Open Addressing (Linear Probing)

- Suppose now

  hash("apple") = 5
  hash("watermelon") = 3
  hash("kiwi") = 0
  hash("mango") = 6
  hash("banana") = 2
  hash(“orange") = 3
  hash(“pear") = 3

- **Deleting orange (incorrect, using NeverUsed)**
- Assume we delete orange by replacing the entry with ***NeverUsed***. This will not allow us to find pear as we will stop searching when we find ***NeverUsed***
- We need three states for a table entry
  - **Occupied**, **NeverUsed**, **Removed**
- Removing an element will change the element to ***Removed*** rather than ***NeverUsed***

| | |
|---|---|
| 0 | kiwi |
| 1 | *NeverUsed* |
| 2 | banana |
| 3 | watermelon |
| 4 | ~~orange~~ *NeverUsed* |
| 5 | apple |
| 6 | mango |
| 7 | pear |

# Removal: Open Addressing (Linear Probing)

- Suppose now

  hash("apple") = 5
  hash("watermelon") = 3
  hash("kiwi") = 0
  hash("mango") = 6
  hash("banana") = 2
  hash(“orange") = 3
  hash(“pear") = 3

| | |
|---|---|
| 0 | kiwi |
| 1 | *NeverUsed* |
| 2 | banana |
| 3 | watermelon |
| 4 | ~~orange~~ *Removed* |
| 5 | apple |
| 6 | mango |
| 7 | pear |

- **Deleting orange** (correct, using **Removed**)
- Deleting orange by replacing the entry with ***Removed***
- When we search, we do not stop when we find ***Removed***; only when we find ***NeverUsed***
- Now we can find pear after removing orange

# Insertion: Revisited

- Suppose now

  hash("apple") = 5
  hash("watermelon") = 3
  hash("kiwi") = 0
  hash("mango") = 6
  hash("banana") = 2
  hash(“orange") = 3
  hash(“pear") = 3
  hash(“grape") = 2

| | |
|---|---|
| 0 | kiwi |
| 1 | *NeverUsed* |
| 2 | banana |
| 3 | watermelon |
| 4 | ~~*Removed*~~ grape |
| 5 | apple |
| 6 | mango |
| 7 | pear |

- **Inserting grape**
- To insert grape we first need to determine whether it is in the table (we search until we find it or find ***NeverUsed***). In this traversal we make a note about the first ***Removed (4)*** and ***NeverUsed* (1)** found
- To complete the insertion, we should use the first ***Remove*** found instead of ***NeverUsed***. Using ***NeverUsed*** will lead to longer search times for grape. Also using ***NeverUsed*** would fill the hash table faster (something we want to avoid)

# Clustering

- Collisions resolved with linear probing generate groups of consecutive elements in the hash table. Each group is called a cluster and the phenomenon is known as **primary clustering**
  - Each cluster is a probe sequence you must search when adding, removing, retrieving
  - Bigger clusters mean longer search times
- Linear probing can cause primary clustering
- Quadratic probing avoids primary clustering, but can lead to secondary clustering

# Separate Chaining

- **Separate Chaining** - Second approach to resolve collisions where each element of the table represents more than one value. Each element is called a bucket
  - Elements that hash to the same entry are stored in the same bucket
- **Bucket** – Can be represented with a list, sorted list, linked nodes, etc.
- Operations
  - **Search** – Determine the bucket by hashing the search key; look through the list to find the element or determine it does not exist
  - **Insert** – Look for the item; insert it in the found bucket if not found
  - **Remove** – Look for the item and remove it from the bucket
- You can add entries to a bucket in sorted search-key order, although it is usually unnecessary as typical buckets are short
- You can add entries at the beginning of the bucket if duplicates are allowed or at the end if not

# Load Factor

- **Load Factor (λ)** - measure of the cost of collision resolution

$$\lambda = \frac{\textbf{Number of entries in the hash table}}{\textbf{Size of the table}}$$

- For Open Addressing – λ does not exceed 1
- For Separate Chaining – λ has no maximum value
- As **λ** increases, number of comparisons increases
- Performance of linear probing degrades as the load factor increases
  - To main reasonable efficiency, keep **λ < 0.5** (i.e., hash table should be less than half full)
- For reasonable efficiency of separate chaining keep **λ < 1**
- **Rehashing** - When the load factor becomes large, resize the hash table and compute a new hash index for each key

# Hashing in Java

- **hashCode() method**
  - Returns hash code **(not hash index)**
  - Part of the **Object** class
  - Provides hashing support by returning a hash code for any object
  - 32-bit **signed** int – Can be a negative value!
- **Default hashCode( ) implementation**
  - Usually just address of object in memory
- How **hashCode()** could be used:

```
int getHashIndex(K key) {
   int hashIndex = key.hashCode() % hashTableLength;

   return Math.abs(hashIndex);
}
```

# Java Hash Code Contract

- **If you override equals you need to make sure the** "Java Hash Code Contract" is satisfied
- **Java Hash Code Contract**

  if a.equals(b) == true, then we must guarantee

  a.hashCode( ) == b.hashCode( )
- **Inverse is not true**

  !a.equals(b) does not imply a.hashCode( ) != b.hashCode( )

  (Though Java libraries may be more efficient)
- **Converse is also not true**

  a.hashCode( ) == b.hashCode( ) does not imply a.equals(b) == true
- **hashCode()**
  - Must return same value for object in each execution, provided information used in equals( ) comparisons on the object is not modified
  - Easiest (and worst) hashCode implementation – return a constant (e.g., 10, 20, etc)

# When to Override hashCode

- **You must write classes that satisfy the Java Hash Code Contract**
- You will run into problems if you don't satisfy the Java Hash Code Contract and use classes that rely on hashing (e.g., HashMap)
  - Possible problem
    - You add an element to a set but cannot find it during a lookup
  - **Example:** See code distribution
- Does the default **equals** and **hashCode** satisfy the contract? Yes!
- If you implement the **Comparable** interface, you should provide the appropriate **equals** method which leads to the appropriate **hashCode** method
- **Implementing hashCode( )**
  - IMPORTANT: include only information used by equals( )
    - Otherwise two "equal" objects → different hash values
  - Using all/more of information used by equals( )
    - Helps avoid same hash value for unequal objects

# Beware of % (Modulo Operator)

- The % operator is integer remainder

  x % y == x – y * ( x / y )

- Result may be negative

  –|y| < x % y < +|y|

- **x % y has same sign as x**
  - -3 % 2 = -1
  - -3 % -2 = -1
- About absolute value in Java
  - Math.abs(Integer.MIN_VALUE) == Integer.MIN_VALUE !
    - Absolute value is a negative value
  - Will happen 1 in $2^{32}$ times (on average) for random int values
  - **Example:** Absolute.java
- **You must use Math.abs( x % N ) and not Math.abs( x ) % N,** otherwise you will get a negative hash index. By doing % first, you get a value larger than Integer.MIN_VALUE. This will avoid computing the absolute value of Integer.MIN_VALUE which generates a negative value

# Art and Magic of hashCode( )

- There is no “right” hashCode function
  - Art involved in finding good hashCode function
  - Also for finding hashCode to hashBucket function (hashBucket returns a hash index)
- From java.util.HashMap

```
static int hashBucket(Object x, int N) {
    int h = x.hashCode();
    h += ~(h << 9);
    h ^=  (h >>> 14);
    h +=  (h << 4);
    h ^=  (h >>> 10);
    return Math.abs(h % N);
}
```

# References

Data Structures & Abstractions with Java, 5th Edition

ISBN – 9780134831695