Hashing

Introduction

Tasks of the Week

- Introduce hashing
- Case Study: Experimenting with hashing using the Java Collection API

Outline

1. Hashing
   - Introduction
   - Hash Functions
   - Collision Resolution
   - Chaining
   - Open Addressing
   - Hashing in Java

Hashing

Each data record has a key. A hash function computes an index from a key.

- If \( \text{hash}(\text{key}) = i \), the record should be stored in bucket \( i \)

Loading Factor: fraction of table that contains data.

- Optimal loading factor is around 70%
- Table size should be about 1.5 times the size of the data set.

\[ h(E) = ? \]
Hashing (cont.)

- Hashing is typically implemented using an array, and each bucket holds one data record.
- Pros: hashing is efficient.
  - The average complexity of insertion and retrieval is $O(1)$
- Cons: collision may occur.
  - Different keys may be hashed into the same bucket.
  - Solution
    - Design hash function to spread data as widely as possible
    - Develop strategies to resolve collision

Simple Hash Functions

- Extraction. Select parts of a key.
  - Example
    - Choose the first letter of a character string

- Division. Compute modulo
  - Example
    - \( \text{hashcode}(k) = \text{Math.abs}(k) \mod N \)
    - where \( N \) is the size of the hash table and should be a prime number

Hash Function Based on Folding

- Folding.
  - Divide key into parts, combine parts, then extract or divide
  - Example
    - Divide a key 987654321 into three parts: 987, 654, and 321.
      - Shift Folding: Shift to align parts
        - \( \text{hash}(987654321) = (987+654+321) \mod N \)
      - Boundary Folding: Reverse every other part
        - \( \text{hash}(987654321) = (987+456+321) \mod N \)

More Hash Functions

- Mid-Square
  - Square the key, select the middle part, then extract or divide
  - Example
    - Let the key be 4321. Square the key: \( 4321^2 = 18671041 \). If only three digits are needed, choose 671 or 710.

- Radix
  - Transform the key from one number base to another, then use other method
    - Example
      - First transform 23_{10} into 32_7. Then compute 32 modulo \( N \)
Hash Function For String

Let \( S = c_0c_1 \ldots c_{n-1} \), where \( c_i \) is a character

\[
h(S) = \sum_{i=0}^{n-1} c_i \times 31^{n-1-i}
\]

```java
public int hashCode(String s) {
    int hashcode = 0;
    for (int i = 0; i < s.length(); i++) {
        hashcode *= 31;
        hashcode += s.charAt(i);
    }
    return hashcode;
}
```

Chaining

- **External Chaining.**
  - Each bucket stores the overflow records in a linked list (outside of the hash table)

  ![External Chaining Diagram]

- **Internal Chaining.** The linked list uses the space inside the hash table.

Open Addressing

- If the bucket is occupied, find an open (not yet occupied) bucket to hold the overflow element.
  - The tricky part is to make sure that the element can later be retrieved.
- **Linear Probing.**
  - For \( i = 1, 2, \ldots \), check buckets \( hash(key) + i \), until an open bucket is found
  - Problem: the search is localized, so the performance can be poor

  ![Linear Probing Diagram]

- **Quadratic Probing.**
  - Probe positions that are increasingly farther
    
    \[
    newPos = pos + (-1)^{(i-1)}((i-1)/2)^2
    \]

    which results in probing positions \( pos + 1, pos - 1, pos + 4, pos - 4, pos + 9, pos - 9, \ldots \)
  - Complications: How to delete element?

- **Double hashing.** Use a second hash function.
  
  \[
  newPos = hash(key) + i \times hash2(key)
  \]
Java Hashing Classes

- HashTable
- HashSet. Implement Set interface
  - Uses chaining, fixed capacity, 75% load factor, no guarantee of ordering
- HashMap. Implement Map interface
- IdentityHashMap. Based on reference (rather than object) equality
- WeakHashMap. Allows null key and null value
- LinkedHashSet and LinkedHashMap. Use doubly linked list to keep ordering