06 A: Hashing

CS1102S: Data Structures and Algorithms

Martin Henz

February 23, 2010

Generated on Tuesday 23rd February, 2010, 12:01

Example

Setup

We would like to quickly find out if a given data item is included in a collection.

Example

In an underground carpark, a system captures the licence plate numbers of incoming and outgoing cars.
Problem: Find out if a particular car is in the carpark.

How About Lists, Arrays, Stacks, Queues?

Problem with Lists, Arrays, Stacks, Queues

With lists, arrays, stacks and queues, we can only access the collection using an index or in a LIFO/FIFO manner. Therefore, search takes linear time.

How to avoid linear access?

For efficient data structures, we often exploit properties of data items.

Example

Simple license plates

Let us say the license plate numbers are positive integers from 0 to 9999.

Solution

- Keep an array inCarPark of boolean values (initially all false).
- insert(i) sets inCarPark[i] to true
- remove(i) sets inCarPark[i] to false
- contains(i) returns inCarPark[i].

The Sad Truth

Not all data items are small integers!

In Singapore, license plate numbers start with 2–3 letters, followed by a number, followed by another letter.

But: one property remains

We can *compare* two license plate numbers, for example lexicographically.

Comparison-based Search

- If items can be compared (total ordering), we can organize them in a binary search tree
- Result: $O(\log N)$ retrieval time

Back to Integers

Simplest case

License plate numbers are positive integers from 0 to 9999.

A slight variation

What if the license plate numbers are positive integers from 150,000 to 159,999?

Solution

Store the numbers in an array from 0 to 9999, and apply a *mapping* that generates index from license plate number:

$$hash(key) = key - 150000$$

Type of Hash Key

The most common data structures for search are not integers but strings.

Examples:

- License plate numbers: “SBX 101 W”
- Names: “Lau Tat Seng, Peter”
- NRIC numbers: “F543209X”

A HashTable Interface

```
public interface HashTable<Any> {
    public void insert(Any x);
    public void remove(Any x);
    public void contains(Any x);
}
```

A First Attempt

```
public class NaiveHashTable<Any> {
 private static final int DEFAULT_TABLE_SIZE = 100;
 private static boolean[] theArray;
 public NaiveHashTable( ) {
   this( DEFAULT_TABLE_SIZE );
 }
 public NaiveHashTable(int size) {
   theArray = new boolean[size];
 }
```

A First Attempt

```
public void insert(Any x) {
    theArray[myhash(x)] = true;
}
public void remove(Any x) {
    theArray[myhash(x)] = false;
}
public boolean contains(Any x) {
    return theArray[myhash(x)];
}
private int myhash( Any x ){
    // mapping x to 0..theArray.length
}   }
```

Some Practical Considerations

Consideration 1: Size of array

The size of array cannot be too large; it must fit into main memory!

Consideration 2: Spread

How to “spread” the hash keys evenly over the available hash values?

Consideration 3: Collision

How to handle multiple hash keys mapping to the same value?

Hashing Strings

Requirement

Map arbitrary strings to integers from 0 to a given limit such that the integers are evenly spread between 0 and the limit

First idea

Sum up the characters in the string

Summing up Characters

```
public static int hash(String key,
                       int tableSize) {
  int hashVal = 0;
  for(int i = 0; i < key.length(); i++)
     hashVal += key.charAt( i );
  return hashVal % tableSize; }
```

Summing up Characters

```
public static int hash(String key,
                       int tableSize) {
  int hashVal = 0;
  for(int i = 0; i < key.length(); i++)
     hashVal += key.charAt( i );
  return hashVal % tableSize; }
```

What if tableSize = 10007 and all strings have a length of at most 3 characters?

Second Attempt

Idea

If the string consists of English words, we could make sure that each different combinations of the first three letters hash to a different value.

```
public static int hash(String key,
                       int tableSize) {
  return ( key.charAt(0) +
           27 * key.charAt(1) +
           729 * key.charAt(2)
         ) % tableSize; }
```

Second Attempt

```
public static int hash(String key,
                       int tableSize) {
  return ( key.charAt(0) +
           27 * key.charAt(1) +
           729 * key.charAt(2)
         ) % tableSize; }
```

Analysis

There are $26^3 = 17,576$ possible combinations of three letter characters, but only 2851 actually occur in English!

Third Attempt

Idea

Compute

$$\sum_{i=0}^{KeySize-1} Key[KeySize - i - 1] \cdot 27^i$$

and bring result into proper range between 0 and tableSize.

Third Attempt

```
public static int hash(String key,
                       int tableSize) {
  int hashVal = 0;
  for(int i = 0; i < key.length( ); i++)
    hashVal = 37 * hashVal + key.charAt(i);
  hashVal %= tableSize;
  if(hashVal < 0)
    hashVal += tableSize;
    return hashVal; }
```

Common Variations

- Use only prefix of overall string
- Use every second character
- Use specific data (street address)

Recap: Considerations

Consideration 1: Size of array

The size of array cannot be too large; it must fit into main memory!

Consideration 2: Spread

How to “spread” the hash keys evenly over the available hash values?

Consideration 3: Collision

How to handle multiple hash keys mapping to the same value?

Separate Chaining

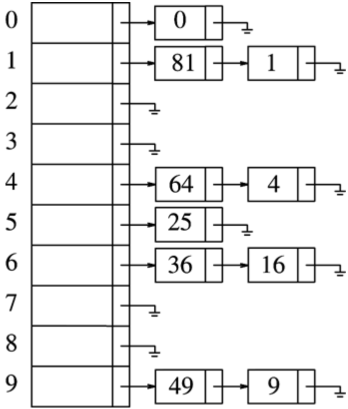
Idea

Keep all elements that hash to the same value in a linked list

Modify hash table operations

Hash table operations (insert, remove, contains) now iterate through list

Separate Chaining Example



Excursion: The Class Object

```
public class Object {
  protected Object clone() {...}
  boolean equals(Object obj) {...}
  protected void finalize() {...}
  Class<?> getClass() {...}
  int hashCode() {...}
  String toString() {...}
}
```

Excursion: Preparing a Class for Hashing

```
public class Employee {
  public boolean equals(Object rhs) {
    return rhs instanceof Employee &&
       name.equals( ((Employee)rhs).name ); }
  public int hashCode() {
    return name.hashCode(); }
  private String name;
  private double salary;
  private int seniority; }
```

Separate Chaining Implementation

```
public class SeparateChainingHashTable<Any> {
  public SeparateChainingHashTable( )
    { ... }
  public SeparateChainingHashTable( int size )
    { ... }
  public void insert( Any x )
    { ... }
  public void remove( Any x )
    { ... }
  public boolean contains( Any x )
    { ... }
  public void makeEmpty( )
    { ... }
```

Separate Chaining Implementation

```
private static final int DEFAULT_TABLE_SIZE = 101;
private List<Any> [ ] theLists;
private int currentSize;
private int myhash(Any x) {
  ... }
```

Separate Chaining Implementation

```
private int myhash(Any x) {
  int hashVal = x.hashCode( );
  hashVal %= theLists.length;
  if( hashVal < 0 )
    hashVal += theLists.length;
  return hashVal;
}
```

Separate Chaining Implementation

```
public SeparateChainingHashTable() {
  this( DEFAULT_TABLE_SIZE );
}
public SeparateChainingHashTable(int size) {
  theLists = new LinkedList[ nextPrime( size ) ];
  for( int i = 0; i < theLists.length; i++ )
     theLists[ i ] = new LinkedList<Any>( );
}
public void makeEmpty( ) {
  for( int i = 0; i < theLists.length; i++ )
     theLists[ i ].clear( );
  currentSize = 0;
}
```

Separate Chaining Implementation

```
public boolean contains(Any x) {
  List<Any> whichList = theLists[ myhash( x ) ];
  return whichList.contains( x );
}
public void insert(Any x) {
  List<Any> whichList = theLists[ myhash( x ) ];
  if( !whichList.contains( x ) ) {
    whichList.add( x );
    if( ++currentSize > theLists.length )
      rehash( );
  }
}
```

Separate Chaining Implementation

```
public void remove( Any x ) {
  List<Any> whichList = theLists[ myhash( x ) ];
  if( whichList.contains( x ) ) {
    whichList.remove( x );
    currentSize--;
  }
}
```

Analysis

Effectiveness

Separate chaining is a simple and effective technique to deal with collisions

Disadvantage

Linked lists add inefficiency due to the need to create objects at runtime.

Idea

Store items directly into array; use alternative cells if a collision occurs

Hash Tables without Linked Lists

Idea

Store items directly into array; use alternative cells if a collision occurs

More formally

Try cells $h_0(x), h_1(x), h_2(x), \ldots$ until an empty cell is found.

How to define h_i?

$h_i(x) = (hash(x) + f(i)) \mod TableSize, \text{where} f(0) = 0$

Definition

They function f is called the *collision resolution strategy*.

Linear Probing

Idea

If $hash(x)$ is taken, try the next cell to the right. If that is taken, too, try the next one, etc.

Formally

$$f(i) = i$$

Linear Probing: Example

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Problem with linear probing

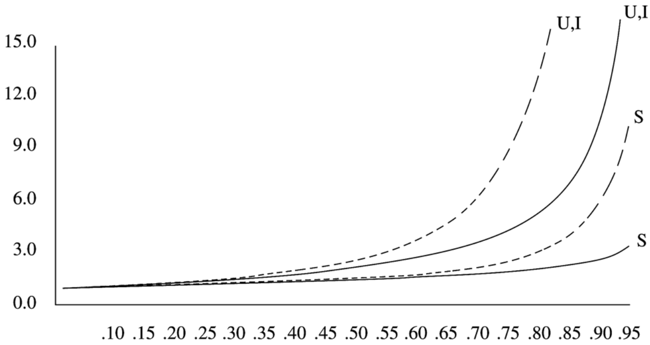
Definition

The *load factor*, λ, of a hash table is the ratio of the number of elements in the hash table to the table size.

Clustering

As the load factor λ increases, occupied areas in the array tend to occur in clusters, leading to frequent unsuccessful insertion tries.

Linear Probing vs Random Strategy



Quadratic Probing

Idea

To avoid clustering, increase the step size with each unsuccessful try.

Formally

$$f(i) = i^2$$

Quadratic Probing: Example

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Properties of Linear and Quadratic Probing

Expected number of probes for linear probing

$$\frac{1}{2}(1 + 1/(1 - \lambda)^2)$$

Quadratic probing

Can we guarantee that we find an empty slot, if an empty slot exists?

Theorem

If quadratic probing is used, and the table size is prime, then a new element can always be inserted if the table is at least half empty.

Rehashing

Idea

When load factor gets too large (for quadratic hashing close to 1/2), double the array size and *rehash* all elements.

Rehashing: Example

0	6
1	15
2	
3	24
4	
5	
6	13

0	6
1	15
2	23
3	24
4	
5	
6	13

Rehashing: Example

0	6
1	15
2	23
3	24
4	
5	
6	13

0	
1	
2	
3	
4	
5	
6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	

Last Puzzler: Shades of Gray

What does the following program print?

```
public class ShadesOfGray {
  public static void main(String[] args) {
    System.out.println(X.Y.Z);
}  }
class X {
  static class Y {
    static String Z = "Black";
  }
  static C Y = new C(); }
class C {
  String Z = "White";
}
```

Obscuring Declarations

```
public class Test {
   public int myVar = 3;
   public void f(int myVar) {
      return myVar + 7;
   }
}
```

There are two declarations of myVar. The inner declaration *obscures* the outer declaration.

Declarations at Same Level...

...are usually not allowed:

```
public class Test {
   public int myVar = 3;
   public int myVar = 4; // leads to
                         // compilation
                         // error
   ...
}
```

Exceptions

- When a variable and a type have the same name and both are in scope, the variable name takes precedence.
- A variable name takes precedence over package names.
- A type name takes precedence over package names.

Puzzler Solution: Shades of Gray

The program

```
public class ShadesOfGray {
  public static void main(String[] args) {
    System.out.println(X.Y.Z);
}  }
class X {
  static class Y {
    static String Z = "Black";
  }
  static C Y = new C(); }
class C {
  String Z = "White";
}
```

How to Avoid Conflicts?

Naming conventions

- Classes (types) begin with a capital letter
- Variables begin with a lowercase letter
- Constants arwe written in ALL_CAPS
- Package names are written in lower.case
- Avoid variable names such as com, org, net, edu, java

The Program using Naming Convention

```
public class ShadesOfGray {
  public static void main(String[] args) {
    System.out.println(Ex.Why.z);
} }
class Ex {
  static class Why {
    static String z = "Black";
  }
  static See y = new See(); }
class See {
  String z = "White";
}
```

New Puzzler: It's Elementary

What does the following program print?

```
public class Elementary {
   public static void main(String[] args) {
      System.out.println(12345 + 5432l);
   }
}
```

New Puzzler: It's Elementary

What does the following program print?

```
public class Elementary {
   public static void main(String[] args) {
      System.out.println(12345 + 5432l);
   }
}
```

Output: 17777

New Puzzler: It's Elementary

What does the following program print?

```
public class Elementary {
    public static void main(String[] args) {
        System.out.println(12345 + 5432l);
    }
}
```

Output: 17777

Why?

Next Week

- Friday: Hashing; priority queues
- After that: Sorting, sorting, and more sorting!