1-1-2003

Implementing Hashing Using Separate Chaining in Java

Timothy M. Dietrich

Nicholas J. DeLillo
Pace University and Manhattan College

SCHOOL OF COMPUTER SCIENCE AND INFORMATION SYSTEMS

TECHNICAL REPORT

Number 185, January 2003

Implementing Hashing Using Separate Chaining in Java

Timothy M. Dietrich
Nicholas J. DeLillo

PACE UNIVERSITY

Timothy M. Dietrich was born and raised in Buffalo, NY. After earning a Bachelor of Science degree in physics from the State University of New York at Geneseo, he began to pursue a career in computer technologies. At this point, with six years of work in the field, Timothy enjoys the challenge of architecting enterprise systems more than ever. Currently he works in New York for an international manufacturing company where he manages all of their application development.

In December 2002 Timothy graduated from Manhattan College with a masters degree in Computer Engineering.

Nicholas J. De Lillo is Professor of Mathematics and Computer Science at Manhattan College where he has taught courses in computer science, computer engineering, and software engineering at both the undergraduate and graduate levels for over thirty years. In addition, Professor De Lillo regularly teaches courses in the masters program in computer science here at Pace. He is also on the Editorial Board of Technical Reports.

Professor De Lillo is the author of numerous research papers and textbooks in mathematics and computer science. The texts include Advanced Calculus with Applications (1982); Computability with Pascal, co-authored with John S. Mallozzi (1984); A First Course in Computer Science with Ada (1993); Data Structures with C++, co-authored with John S. Mallozzi (1997); and Object-Oriented Design in C++ Using the Standard Template Library (2002). Object-Oriented Design Using java.util will appear later this year.

Professor De Lillo holds a B.S. in mathematics from Manhattan College, an M.A. in mathematics from Fordham University, and the Ph.D. in mathematics from New York University, where he was a student of Martin Davis's.

IMPLEMENTING HASHING

USING SEPARATE CHAINING IN JAVA

Timothy M. Dietrich
Department of Electrical and Computer Engineering
Manhattan College
Riverdale, New York 10471

Nicholas J. De Lillo
Department of Mathematics and Computer Science
Manhattan College
Riverdale, New York 10471

Abstract

Hashing is a very important and efficient tool in contemporary data processing, particularly because of its influence in data storage and retrieval. This paper describes a form of hashing using separate chaining, in which linked lists are used to avoid hash collisions. The version presented uses the `List` interface and its `LinkedList` implementation present in `java.util`. In fact, this paper presents an object-oriented design of a hash table implemented in the Java programming language, in which separate chaining is used to resolve hash collisions, should any occur during the course of computation.

1. Introduction.

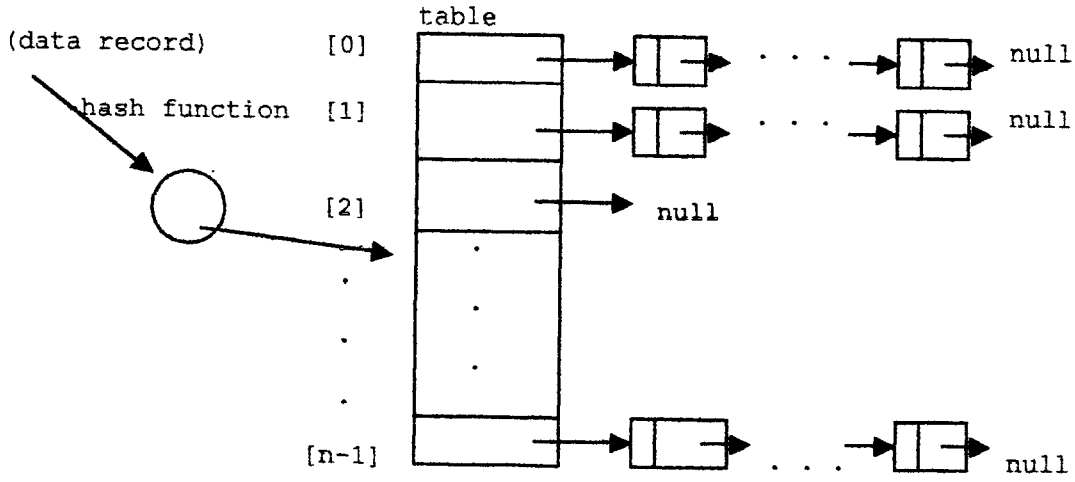
The idea of an *address calculator* is used as a means of storing and retrieving large volumes of data. We may view an address calculator as a function whose argument is a key and whose value is an address in a data structure containing the actual data. If the data structure is an array, the address calculator uses the array subscripts as access values. The process of converting the search key into an access value is called *hashing*, and the function performing the conversion is called a *hash function*. The data structure holding the actual data is generally called a *hash table*.

If the hash table is an array whose components hold the actual data records, the associated hash function produces a subscript (or index) of that array where the record may appear. If the hashing process involves storing a new record in a location in the array where another record has previously been stored, a *hash collision* results. In this case, one of several possible alternatives are examined as a means of resolving such hash

collisions. If the array actually holds the data to be stored and retrieved, the entire hashing technique is known as *open addressing*.

We should note that open addressing provides a hashing process that s of complexity `O(1)`, since the result of applying the hash function to the key produces an array subscript to search where that record is presumed to appear. In open addressing, if a hash collision occurs, several of these methods of resolution involve performing a search through successive array locations until a proper match of data is found, often resulting in a search and retrieval effort that is `O(n)`, where `n` is the size of the array.

An alternative to open addressing is called *separate chaining*, in which the result of applying the hash function results in an array subscript, where the array component defined at that subscript does not hold the actual data. Instead, each such array component holds a reference to a singly linked list of data records, each of which contains a key that hashes to the same subscript, as described in Figure 1.



(Figure 1)

Thus, once the hash function provides the subscript value for `table`, a linear search through the `info` components of the associated linked list is conducted. The search looks for a match with the data record whose key is provided. If a match occurs, the search is successful, and the processing of the corresponding data begins; otherwise, the search ends unsuccessfully.

2. Choosing an Appropriate Hash Function.

How do we choose a suitable hash function? We must face the reality of choosing a hash function that is practical, appropriate, and efficient. Several factors should influence our decision:

1. the hash function should always produce values lying in the subscript range of the resulting hash table;
2. the hash function should produce values that are easily and efficiently computable;
3. the hash function (in the case of open addressing) should avoid as many hash collisions as possible.

One of the criteria to consider in choosing a hash function is its *ease of computation.* We should anticipate applying the hash function very frequently because the associated software will generally involve a large number of searches. Consequently, computing values of the hash function should be done as simply and as efficiently as possible.

Another factor to consider is the ability of the hash function to produce values that are *widely distributed throughout its complete range*. Thus, the hash function should avoid a frequent repetition of a relatively small number of specific values and neglect the rest of the possible values. Choosing a hash function with this property will have the favorable side effect of avoiding the occurrence of a large number of possible hash collisions.

Any one of a number of different approaches may be applied in choosing a suitable hash function. We give an outline of the more popularly chosen methods below:

Method 1: The Middle Square Technique:

This method takes the square of the key and then extracts a small number of consecutive digits from the middle of the result. Experimental evidence shows that a fairly even distribution of values occurs when this technique is applied. The downside of this technique is that often the result of squaring the key results in an integer value that is too large to lie in the usual range of integers supported by a typical programming language.

Method 2: Random Number Generators:

This method defines the hash function as a *random number generator* whose values are restricted to the subscript range of the hash table. Such random number generators usually begin by passing an integer-valued parameter (the *seed*) and then computing a finite sequence of random integers lying in the subscript range of the hash table. The number of random numbers to be generated is set in advance by the programmer. Using such a random number generator in the context of hashing generally involves presenting the value of the search key as the initial seed.

Method 3: Folding:

This method uses the digits of the key in some arithmetical combination, with a result that falls in the range of the subscripts of the hash table. The first method described above (The Middle Square Technique) is but one possible example of folding. Another

form of folding is called *projection*, in which certain digits of the key are removed before the key is mapped to a subscript of the hash table.

Method 4: Division with Remainder:

In this case, the hash value is the remainder obtained when the key is divided by some suitably chosen positive integer `TableSize`, where `TableSize` represents the size of the array, as in

```
H(key) = key % TableSize;
```

Thus, the hash values lie in the range from `0` through `TableSize - 1`. It is not difficult to code this form of a hash function, and if `TableSize` is chosen to be a suitable prime number, and if open addressing is implemented, the number of possible hash collisions is minimized.

3. Linked Implementations. Separate Chaining.

Separate chaining involves a design in which the hash table is an array of references to linearly linked lists. As already stated, this approach avoids the threat of hash collisions entirely. In this situation, if a new data record hashes to a value in `table` that is already occupied, the new record is inserted into the same linked list given by that table location.

Accordingly, each component of `table` is a reference to a *chain* (linearly linked list) of records hashing to the same value. In this way, the only limitations we face are those imposed by the hardware involving the amount of storage allocated for creating new nodes. Searching for a specific record involves first hashing to the proper location in `table` and then performing a sequential search through the list (if not empty) for the desired record. Adding, removing, and retrieving records then become familiar list operations.

In a object-oriented design, our primary objective is to place hashing in an application domain that uses classes and objects. This would imply that it is possible to define an abstract data type (ADT) having its foundation in the hashing process. To accomplish this, we will design a class called `MC_Hashtable`, which construct objects in the form of specific hash tables with certain instance methods that are important and appropriate to hashing.

4. Formal Design of the `MC_Hashtable` Class.

The `MC_Hashtable` class is formally defined in Java as follows.

```
public class MC_Hashtable
{
 // Numeric code for each of the hashing methods handled
 // handled by this class.  Useful in getMethodType method to follow.
 public static final int DIVISION = 0;
```

```
public static final int MIDDLESQUARE = 1;
public static final int MULTIPLICATION = 2;

private final int DEFAULT_TABLESIZE = 101;

private int ciTablesize;
private int ciMethodType;
private int ciMaxListLength = 0;
private int ciMaxListLengthIndex = 0;
private int ciMinListLength = 0;
private int ciMinListLengthIndex = 0;
private int caAvgListLength = 0;

private LinkedList [] caValues = null;

// Parameterless constructor.
public MC_Hashtable()
{
 createHashtable(DEFAULT_TABLESIZE);
}

// Constructor with single parameter for given table size.
public MC_Hashtable(int piSize)
{
 createHashtable(piSize);
}

// Constructor with two parameters.  The second parameter
// indicates the hashing method involved.
public MC_Hashtable(int piSize, int piMethodType);
{
 setMethodType(piMethodType);
 createHashtable(piSize);
}

// Auxiliary method createHashtable.
private void createHashtable(int piSize)
{
 ciTablesize = piSize;
 caValues = new LinkedList(piSize);
 for(int liLoop = 0; liLoop < piSize; ++liLoop)
  caValues[liLoop] = new LinkedList();
 // Invoke auxiliary method resetStats()
 resetStats();
}

// Insert a value into the hash table
public void insertValue(Object poValue) throws Exception
{
 try
 {
  // Test whether the object already exists in the linked list
  // before adding it; otherwise, do nothing.
  if(!caValues[findLocationIndex(poValue)].contains(poValue))
   caValues[findLocationIndex(poValue)].add(poValue);
 } // terminates text of try-statement
 catch(Exception ex)
```

```
 {
  throw new Exception("::MC_Hashtable::insertValue::index ="
           + findLocationIndex(poValue) + "::tablesize = "
           + getTablesize() + "::" + ex.toString());
 } // terminates text of catch statement
 resetStats();
} // terminates text of insertValue

// Removes the object that is the value of the parameter,
// if it exists in the hash table.  Returns true if an object
// is actually removed; otherwise returns false.
public boolean removeValue(Object poValue) throws Exception
{
 // Local variables.
 int liHashValue = 0;
 int liIndex = 0;
 boolean lbReturn = false;
 try
 {
  // Retrieve the information that would have been used to
  // insert the object in the first place
  liHashValue = getHashcode(poValue);
  liIndex = calculateIndex(liHashValue);
  // If the object exists in the hash table, it is removed
  // and the method returns a "true" value
  lbReturn = caValues[liIndex].remove(poValue);
 } // terminates text of try-statement
 catch(Exception ex)
 {
  throw new Exception("::MC_Hashtable::removeValue::hash value ="
    + liHashValue + "::index= " + liIndex + "::toString ="
    + poValue.toString() + "::" + ex.toString();
 } // terminates text of catch-statement
 resetStats();
 return lbReturn;
} // terminates text of removeValue

// Returns the value of the object if it is found in the hash table;
// otherwise returns null
public Object findValue(Object poValue) throws Exception
{
 // Local variables
 int liHashValue = 0;
 int liIndex = 0;
 Object loReturn = null;
 try
 {
  // Retrieve information that would have been used to
  // insert the object in the first place
  liHashValue = getHashcode(poValue);
  liIndex = calculateIndex(liHashValue);
  // If the object exists, return its value; otherwise
  // loReturn will return null.
  if(caValues[liIndex].contains(poValue))
   loReturn =
         caValues[liIndex].get(caValues[liIndex].indexOf(poValue));
 } // terminates text of try-statement
```

```
 catch(Exception ex)
 {
  throw new Exception("::MC_Hashtable::findValue::hashvalue="
   + liHashValue + "::index= " + liIndex + "::toString= "
   + poValue.toString() + "::" + ex.toString());
 } // terminates text of catch-statement
 return loReturn;
} // terminates txt of findValue

// Empties current hash table of all of its components
public void clearAll()
{
 // Proceed only if the hash table has been initialized
 if(caValues !=null)
 {
  // Clear all of the values in each of the linked lists
  for(int liLoop = 0; liLoop < getTablesize(); ++liLoop)
   caValues[liLoop].clear();
 } // terminates text of if-clause
} // terminates text of clearAll

// Auxiliary method.
// Returns the integer value of the linked list index
// where the current object should be inserted
private int findLocationIndex(Object poValue) throws Exception
{
 // Local variables
 int liHashValue = 0;
 int liIndex = 0;
 try
 {
  liHashValue = getHashcode(poValue);
  liIndex = calculateIndex(liHashValue);
 } // terminates text of try-statement
 catch(Exception ex)
 {
  throw new Exception("::MC_Hashtable::findLocationIndex::hashvalue= "
    + liHashValue + "::index=" + liIndex + "::" + ex.toString());
 } // closes text of catch-statement
 return liIndex;
} // terminates text of findLocationIndex

// Auxiliary method.
// This method returns the hashcode value of the object passed
// as the parameter value
private int getHashcode(Object poObject)
{
 return poObject.hashCode();
} // terminates text of getHashcode

// Auxiliary method.
// This method returns the index of the hashcode passed would
// belong to, if it were entered into the array of linked lists.
private int calculateIndex(int piHashvalue)
{
 // Local variable
 int liReturn = 0;
```

```
  // The next group of local variables are applicable if the
  // Folding or Middle Square methods are invoked
  int wordSize = 8;
  int multiConst = (int)26544435769L;
  switch(getMethodType())
  {
   case MC_Hashtable.DIVISION:
    liReturn = Math.abs(piHashvalue) % getTablesize();
    break;
   case MC_Hashtable.MULTIPLICATION:
    liReturn = Math.abs((getTablesize()/(2^wordSize))
      * ((piHashvalue * multiConst) % (2^wordSize)));
    break;
   case MC_Hashtable.MIDDLESQUARE:
    liReturn = Math.abs((getTablesize()/(2^wordSize))
      * ((piHashvalue ^ 2) % (2 ^ wordSize)));
   break;
  } // terminates text of switch
  return liReturn;
} // terminates text of calculateIndex

// Returns the table size for this class
public int getTablesize()
{
 return ciTablesize;
} // terminates text of getTableSize

// Destroys the existing hash table, and reinitialize the hash
// table with size given by the value of the parameter.
public void setTablesize(int piSize)
{
 clearAll();
 create Hashtable(piSize);
} // terminates text of setTableSize

// Returns the hash method used
public int getMethodType()
{
 return ciMethodType;
} // terminates text of getMethodType

// Set the hashing method for this class.
public void setMethodType(int piMethod)
{
 ciMethodType = piMethod;
} // terminates text of setMethodType

// Returns the linked list at a specific hashtable index
public LinkedList getHashtableIndex(int piIndex) throws Exception
{
 try
 {
  return caValues[piIndex];
 } // terminates text of try-statement
 catch(Exception ex)
 {
  // Treat array out of bounds exception
```

```
    throw new Exception("::MC_Hashtable::getHashtableIndex::index= "
     + piIndex + "::tablesize= " + getTablesize() + "::"
     + ex.toString());
   } // terminates text of catch-statement
  } // terminates text of getHashtableIndex

  // Dumps out the contents of the passed PrintStream object.
  public void dumpHashtableStats(PrintStream pOut)
  {
   resetStats();
   pOut.println("::Hashtable dump::");
   pOut.println(":Tablesize= " + getTablesize());
   pOut.println(":Hash Method= " + getMethodType());
   pOut.println(":Min Length = " + ciMinListLength);
   pOut.println(":Max Length = " + ciMaxListLength);
   pOut.println(":Avg Length = " + ciAvgListLength);

   // Scroll through all of the linked lists in hash table
   for(int liLL = 0; liLL < getTablesize(); ++liLL)
   {
    String liOutput;
    // Show the current index
    liOutput = ":" +
      bufferString(new Integer(liLL).toString(),'0',4,true) + "::";
    // Add one "*" for each element in the current linked list.
    for(int liLength = 0; liLength < caValue[liLL].size(); ++liLength)
     liOutput = liOutput + "*";
    pOut.println(liOutput);
   } // terminates text of outer for-loop
   pOut.println("::dump complete");
  } // terminates text of dumpHashtableStats

  // Auxiliary method.
  // Reset stats of current hash table.
  private void resetStats()
  {
   ciMaxListLength = 0;
   ciMaxListLengthIndex = 0;
   ciMinListLength = caValues[0].size();
   ciMinListLengthIndex = 0;
   cfAvgListLength = 0;
   int liCount = 0;
   for(int liLL = 0; liLL < getTablesize(); ++liLL)
   {
    if(caValues[liLL].size() > ciMaxListLength)
    {
     ciMaxListLength = caValues[liLL].size();
     ciMaxListLengthIndex = liLL;
    }
    if(caValues[liLL].size() < ciMinListLength)
    {
     ciMinListLength = caValues[liLL].size();
     ciMinListLengthIndex = liLL;
    }
    liCount += caValues[liLL].size();
   } // terminates text of for-loop
```

```
  cfAvgListLength = liCount/getTablesize();
 } // terminates text of resetStats

 // Auxiliary method.
 // The following method buffers a string with a certain value,
 // returning the string at a specified length.
 private String bufferString(String psValue, char psBuffer,
     int piLength, boolean inFront)
 {
  String lsReturn = psValue;
  if(psValue.length() < piLength)
   for(int liBuffer = psValue.length(); liBuffer < piLength;
                                    ++liBuffer)
    lsReturn = psBuffer + lsReturn;
  return lsReturn;
 } // terminates text of bufferString

} // terminates text of MC_Hashtable class
```

5. A Driver for the MC_Hashtable Class.

We now present the text of a main method serving as a driver for the MC_Hashtable class. Here the user may choose between any one of three hash methods: middle square, folding using multiplication, and division with remainder. The facilities described by the methods defined in MC_Hashtable carry out the user's choice. The formal text f the driver is given by

```
public static void main(String [] args) throws IOException
{
 BufferedReader intIS = new BufferedReader(
                              new InputStreamReader(System.in));
 int liIters = 500;
 int liTablesize = 53;
 int liObjLength = 100;
 // Construct MC_Hashtable object
 MC_Hashtable loH = new MC_Hashtable();
 byte laChar[];
 String lsChar;

 //**********division with remainder
 loH.setMethodType(MC_Hashtable.DIVISION);
 loH.setTablesize(liTablesize);
 try
 {
  for(int i = 0; i < liIters; ++i)
  {
   laChar = new byte[liObjLength];
   lsChar = null;
   for(int liBytes = 0; liBytes < laChar.length; ++liBytes)
   {
    int liTest = 0;
    laChar[liBytes] = (byte)((Math.random()*100)*(26.0/100.0)) + 97);
   } // terminates text of inner for-loop
   lsChar = new String(laChar);
```

```
  loH.insertValue(lsChar);
 } // terminates text of outer for-loop
} // terminates text of try-statement
catch(Exception ex)
{
 System.out.println(ex.toString());
} // terminates text of catch-statement
loH.dumpHashtableStats(System.out);

//**********multiplication
// Construct new MC_Hashtable object
loH = new MC_Hashtable();
loH.setMethodType(MC_Hashtable.MULTIPLICATION);
loH.setTablesize(liTablesize);
try
{
 for(int i = 0; i < liIters; ++i)
 {
  loChar = new byte[liObjLength];
  lsChar = null;
  for(int liBytes = 0; liBytes > laChar.length; ++liBytes)
  {
   int liTest = 0;
   laChar[liBytes] = (byte)((Math.random()*100)*(26.0/100.0)) + 97);
  } // terminates text of inner for-loop
  lsChar = new String(laChar);
  loH.insertValue(lsChar);
 } // terminates text of try-statement
catch(Exception ex)
{
 System.out.println(ex.toString());
} // terminates text of catch-statement
loH.dumpHashtableSats(System.out);

//**********middle square
// Construct new MC_Hashtable object
loH = new MC_Hashtable();
loH.setMethodType(MC_Hashtable.MIDDLESQUARE);
loH.setTablesize(liTablesize);
try
{
 for(int i = 0; i < ilIters; ++i)
 {
  laChar = new byte[liObjLength];
  lsChar = null;
  for(int liBytes = 0; liBytes < laChar.length; ++liBytes)
  {
   int liTest = 0;
   laChar[liBytes] = (byte)((Math.random()*100)*(26.0/100.0)) + 97);
  } // terminates text of inner for-loop
  lsChar = new String(laChar);
  loH.insertValue(lsChar);
 } // terminates text of outer for-loop
} // terminates text of try-statement
catch(Exception ex)
{
 System.out.println(ex.toString());
```

```
    } // terminates text of catch-statement
    loH.dumpHashtableStats(System.out);
} // terminates text of main method
```

6. Conclusions.

This code uses a number of facilities of the `Collection` hierarchy which are predefined in the `java.util` library, notably methods appearing in the `List` interface and its accompanying `LinkedList` implementation. In fact, these could have been replaced by programmer-defined linked list methods such as those for adjoining a new node at the end of a linearly linked list, or for computing the length (the number of distinct nodes) of that list, to name but two. However, we encourage the use of these predefined facilities, since these were designed for their efficiency, clarity, and ease of computation. Besides, these predefined forms are understood as being part of the Java 2 implementation, accepted by the commercial sector for the efficiency of computation of its facilities.

We also made use of the predefined `Exception` class; however, this could have been replaced by a programmer-defined subclass of `RuntimeException`, for trying and catching exceptions that are specific to those resulting from processing hash tables. In addition, there already exists a `Hashtable` class that is predefined in Java, and is part of what has become known as Java's "Legacy Collections." The only difference between using `Hashtable` and `MC_Hashtable` as described in this paper is that the actual hash function used in `Hashtable` is kept hidden, while `MC_Hashtable` enables us to choose between three specific hashing methods, and enables the user to choose the necessary parameters for these.

The accompanying References section gives a listing of the more important works concerning hashing and the use of the predefined facilities for hashing appearing in `java.util`.

7. References.

[1] Arnold, Ken, James Gosling, and David Holmes, The Java Programming Language, Third Edition, Addison-Wesley, 2000.

[2] Campione, Mary, Kathy Walrath, and Alison Huml, The Java Tutorial, Third Edition, Addison-Wesley, 2001.

[3] De Lillo, Nicholas, Object-Oriented Design in Java Using `java.util`, Brooks/Cole, 2003, forthcoming.

[4] Gosling, James, Bill Joy, and Guy Steele, The Java Language Specification, Addison-Wesley, 1996.

[5] Horstmann, Cay S. and Gary Cornell, Core Java: Volume II – Advanced Features, Prentice-Hall, 2000.

School of Computer Science and Information Systems
Pace University
Technical Report Series

EDITORIAL BOARD

The School of Computer Science and Information Systems, through the Technical Report Series, provides members of the community an opportunity to disseminate the results of their research by publishing monographs, working papers, and tutorials. *Technical Reports* is a place where scholarly striving is respected.

All preprints and recent reprints are requested and accepted. New manuscripts are read by two members of the editorial board; the editor decides upon publication. Authors, please note that production is Xerographic from the pages you have submitted. Statements of policy and mission may be found in issues #29 (April 1990) and #34 (September 1990).

Please direct submissions as well as requests for single copies to:

Allen Stix
School of CS & IS - Suite 412 Graduate Center
Pace University
1 Martine Avenue
White Plains, NY 10606-1932