-------------------------------Hashing---------------------------------------------

Hashing is a technique used for performing insertions, deletions, and searches in constant average time.

The central data structure we will discuss is called the HASH TABLE.

------------------------------------------------------------------------------------------

The ideal hash table data structure is merely an array of some fixed size, containing items

A search is performed on some data field (on some part) of the item.

This PART/DATA FIELD of the item is called the KEY.

- An item could consist of additional data fields, but the data field we search for is called the KEY

We will refer to the size of the hash table, or the size of the ARRAY, as TableSize.

The common convention is to have the table run from 0 to TableSize - 1.

--------------------------------------------------------------------------------------------

Each key is mapped into some number in the range 0 to TableSize - 1 and placed in the appropriate cell.

This mapping is called the HASH FUNCTION.

This is ideally simple to compute, and ensures that any two distinct keys get DIFFERENT cells

However, since there are a finite number of cells and a virtually inexhaustible supply of keys, we want

a hash function that distributes the keys EVENLY among the cells.

The problems are:

1. Choosing a function

2. Deciding what to do when the two keys hash to the same value: also known as a COLLISION

3. Deciding on the table size

--------------------------------------------------------HASH FUNCTION---------------------------------------------------------------

It is often a good idea to ensure that the table size is prime.

Usually, the keys are strings.

In the case below, we add up the ASCII (Unicode) values of the characters in the string.

Ex:

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;

}

Not a good hash function because a string with the same kind of characters and table size will return the

same value, and create a collision.

If the table size is large, the function also does not distribute the keys well.

For ex: if the table size is 10,007, and all the keys are 8 or fewer characters long, then the hash function

only assumes values between 0 and 1016 (or 127 \* 8).

This is clearly NOT an equitable distribution.

----------------------------------------------------------------------------

Another hash function is shown below.

This hash function assumes that the Key has at least 3 characters.

The value 27 represents the number of letters in the English alphabet plus the blank.

729 is 27^2.

This function only examines the first three characters,

but if these characters are RANDOM, and the table size is 10,007, then we would expect a reasonably equitable distribution.

Ex: 26 ^ 3 = 17,567 combinations (ignoring blanks)

HOWEVER, English is not random.

A check on the dictionary shows that there are only 2851 combinations of characters that make up WORDS.

As a result, even if none of these combinations collide, only 28% of the table is actually hashed to.

THEREFORE, it's not appropriate if the hash table is too large.

Ex:

public static int hash( String key, int tableSize )

{

return ( key.charAt( 0 ) + 27 \* key.charAt( 1 ) + 729 \* key.charAt( 2 ) ) % tableSize;

}

Also, if you have the same pre-fix or post-fix words, you would get the same hash value.

INCREASING the chance of collisions (which we want to AVOID)

-------------------------------------------------------------------------------

The hash function is below is a good hash function.

This hash function can be generally expected to distribute well (but is still not the best at it)

If the keys are very long though, the hash function will take too long to compute.

Ex:

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;

}

Some programmers implement their haash function by using only the characters in odd spaces, with the idea

that the time saved computing the hash function will make up for a slightly less evenly distributed function.

The main programming detail left is collision resolution.

--------------------------------------------------Hash code----------------------------------------------

You can use ".equals()" to see if two hash codes are equal.

-------------------------------Linear Probing----------------------------------------

In linear probing, "f" is a linear function of "i".

TYPICALLY, f(i) = i.

This results in trying cells sequentially (with wraparound) in search of an empty cell.

Eventually, however, even if the table is relatively empty, blocks of occupied cells start forming.

This is called PRIMARY CLUSTERING.

This means that any key that hashes into the cluster will require SEVERAL attempts to resolve

the collision, and then when it finally finds a empty space, it will only ADD to the cluster.

The expected number of probes using linear probing is roughly:

1/2 \* ( 1 + 1/(1-LAMBDA)^2 )

for insertions and unsuccessful searches (these require SAME NUMBER of probes).

For successful searches, it's:

1/2 \* ( 1 + 1/(1 - LAMBDA) )

On average, successful searches obviously will take LESS time than unsuccessful searches.

------------------------------------Random Collision Resolution Strategy----------------------------------------------

We will assume a very large table and that each probe is independent of the previous probes.

These assumptions are satisfied by a RANDOM collision resolution strategy and are reasonable

unless LAMBDA is very close to 1.

First, we derive the expected number of probes in an unsuccessful search.

This is the expected number of probes until we find an empty cell.

The fraction of empty cells is 1 - LAMBDA, so the number of cells we expect to probe is 1/(1 - LAMBDA)

The number of probes for a successful search is equal to the number of probes required when the particular

element was inserted.

When an element is inserted, it is done as a result of an unsuccessful search.

As a result, we can use the cost of an unsuccessful search to compute the average cost of a successful search.

NOTE: The earlier insertions are cheaper and bring the average down.

For example, if an element was inserted when LAMBDA was 0.2, a relatively empty table, then the cost of accessing it

is already determined to be that of an unsuccessful search of a table with a LAMBDA value of 0.2, no matter how much

the table grows afterwards

Accessing it should be easier than instead of a more recently inserted element with the LAMBDA at

0.5 or some other greater value.

We can estimate the average by using an integral to calculate the mean value of the insertion time, ultimately

resulting in:

I(LAMBDA) = 1/LAMBDA \* ln( 1/(1-LAMBDA) )

These formulas are better than the corresponding formulas for linear probing.

Ex:

If LAMBDA = 0.75, then the formula above indicates that 8.5 probes are expected for an insertion in linear probing.

If LAMBDA = 0.9, then 50 probes are expected, which is UNREASONABLE.

Compare this with 4 and 10 probes for the respective load factors, if cluserting were not a problem.

Therefore: LINEAR PROBING is a BAD IDEA if the table is expected to be MORE than HALF FULL.

-------------------------------Hash Tables without Linked lists---------------------------

Separate chaining has the DISADVANTAGE of using Linked Lists.

This might slow the algorithm down because of the time needed to allocate new cells (esp. in other languages)

and because of the implementation of a second data structure.

An alternative to resolving collisions is to try alternative cells until an empty cell is found.

In other words:

h0(x), h1(x), h2(x),.... are tried in succession,

where hi(x) = (hash(x) + f(i)) % TableSize, with f(0) = 0

The function "f", is the collision resolution strategy.

Because all the data go INSIDE the table, meaning EACH data takes up ONE CELL,

a BIGGER table is needed in such a scheme than for separate chain hashing.

GENERALLY, the load factor should be below 0.5 for a hash table that doesn't use separate chaining.

We call such tables PROBING HASH TABLES.

Let's look at THREE common collision resolution strategies that stand as ALTERNATIVES to the Separate chaining

Method

----------------------------------------Quadratic Probing-----------------------------

Quadratic probing eliminates the primary clustering problem of linear probing.

As expected, the collision function is QUADRATIC.

The popular choice is: f(i) = i ^ 2

For linear probing, it's a bad idea to let the hash table get nearly full.

However, for quadratic probing, the situation is even MORE drastic.

There is no guarantee of finding an empty cell once the table gets more than half full,

or even before the table gets half full if the table size is NOT PRIME.

THEOREM 5.1:

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 (i.e. AT MOST half full).

If the table is even one more than half full, the insertion could fail (although it's very unlikely)

If the table size is not prime, the number of alternative locations can be severely reduced.

Ex: if the table size were 16, then the only alternative locations would be at distances 1, 4, or 9 away.

IMPORTANT: Determine the size of the table BASED on the TYPE OF PROBING you're doing.

NOT the other way around.

Standard deletion cannot be performed in a probing hash table, because the cell might have caused a collision

to go past it

Ex:

Linear probing

If we remove an element at an array index, the elements that were placed ELSEWHERE due to a

COLLISION with the element will NOT be successfully found by the "contains" method now.

Because it will stop at the supposed array index, and not see an element there.

Thus, probing hash tables require LAZY DELETION.

----------------------------Constructing Hash Table and HashEntry Cells----------------------------------------------------------------

Instead of an array of lists, we have an array of hash table entry cells.

Each entry in the array of HashEntry references either:

1. null

2. Not null, and the entry is ACTIVE (isActive = true)

3. Not null, and the entry is MARKED DELETED (isActive = false)

Constructing the table thus consists of allocating space and then setting each HashEntry reference to null.

Ex:

public class QuadraticProbingHashTable<AnyType>

{

/\*\*

\* Construct hash table

\*/

public QuadraticProbingHashTable()

{

this( DEFAULT\_TABLE\_SIZE );

}

/\*\*

\* Construct the hash table.

\* @param size the approximate intial size.

\*/

public QuadraticProbingHashTable( int size )

{

allocateArray( size );

makeEmpty();

}

/\*\*

\* Make the hash table logically empty

\*/

public void makeEmpty()

{

currentSize = 0;

for( int i = 0; i < array.length; i++ )

array[ i ] = null;

}

/\*\*

\* Internal method to allocate array

\* @param arraySize the size of the array

\*/

private void allocateArray( int arraySize )

{

array = new HashEntry[ nextPrime( arraySize ) ];

}

private static class HashEntry<AnyType>

{

public AnyType element; // the element

public boolean isActive; // false if marked deleted

public HashEntry( AnyType e )

{ this(e, true); }

public HashEntry( AnyType e, boolean i )

{ element = e; isActive = i; }

}

private static final int DEFAULT\_TABLE\_SIZE = 11;

private HashEntry<AnyType> [] array; // the array of elements

private int currentSize; // the number of occupieid cells

--------------------------------------------Contains, findPos, and isActive----------------------------------------------------------------------------

"contains" invokes private methods "findPos" and "isActive"

The private method "findPos" performs the conflict resolution.

Ex:

/\*\*

\* Find an item in the hash table.

\* @param x the item to search for

\* @return the matching item

\*/

public boolean contains( AnyType x )

{

int currentPos = findPos( x );

return isActive( currentPos );

}

/\*\*

\* Method that performs quadratic probing resolution in HALF-EMPTY table

\* @param x the item to search for

\* @return the position where the search terminates

\*/

private int findPos( AnyType x )

{

int offset = 1;

int currentPos = myhash( x );

//NOTICE: This is the collision resolution, offsetting the probe if the element

// at the myhash index does not match the desired element

while( array[ currentPos ] != null && // The ORDER of these conditions

!array[ currentPos ].element.equals( x )) // is IMPORTANT. DON'T SWITCH

{

currentPos += offset; //Compute the ith probe

offset += 2;

if( currentPos >= array.length )

currentPos -= array.length;

}

return currentPos;

}

/\*\*

\* Return true if currentPos exists and is active.

\* @param currentPos the result of a call to findPos

\* @return true if currentPos is active

\*/

private boolean isActive( int currentPos )

{

return array[ currentPos ] != null && array[ currentPos ].isActive;

}

In the implementation above, elements that are marked as deleted COUNT as being in the table.

This can cause the table to get too full prematurely.

This is addressed in our quadratic resolution.

The definiiton of the quadratic resolution function is: f(i) = f( i - 1 ) + 2i - 1

So the next cell to try is a distance from the previous cell tried and this distance increased by 2

on successive probes.

If the new location is past the array, notice how it's put back in range by subtracting TableSize

NOTE: The order of testing in the while statement, with the testing for "null" before the testing of the element

to be equal to"x", is IMPORTANT.

DON'T SWITCH

------------------------------------------------------------Insert-----------------------------------------------------------------------------

NOTICE: We ensure in the "insert" routine that the hash table is at least TWICE as large as the number of elements in

the table, so quadratic resolution will always work.

We do nothing if "x" is alraedy present.

Otherwise, we place it at the location suggested by "findPos"

If the load factor exceeds 0.5, the table is full and we ENLARGE the hash table.

This is called REHASHING.

Ex:

/\*\*

\* @param x the item to insert

\*/

public void insert( AnyType x )

{

int currentPos = findPos( x );

if( isActive( currentPos ))

return;

//Insert x as active

array[ currentPos ] = new HashEntry<>(x, true);

//Rehash

if( ++currentSize > array.length / 2)

rehash();

}

/\*\*

\* @param x the item to remove

\*/

public void remove( AnyType x )

{

int currentPos = findPos( x );

if( isActive( currentPos ))

array[ currentPos ].isActive = false; // LAZY DELETION

}

------------------------------------------------------------------------------------------------------------------------------------------------------

Even though quadratic probing eliminates PRIMARY CLUSTERING, there will still be elments that hash to the same

position that will probe the SAME alternative cells.

Thus, we have SECONDARY CLUSTERING.

Secondary clustering, however, only causes less than an extra half probe per search.

We can eliminate Secondary Clustering with DOUBLE HASHING, at the cost of computing an extra hash function.

`

---------------------------------Rehashing----------------------------------------

If the table gets too full, the running time for the operations will start taking too long and insertions might fail for

open addressing hashing with quadratic resolution.

This can happen with too many removals mixed with insertions.

SOLUTION:

Build another table twice as big.

Scan down the entire original hash table, and compute a new hash value for each

non-deleted element with a NEW hash function.

Insert that new hash value into the new table.

Ex:

A table of size 7 has a hash function of h(x) = x mod 7.

If a 5th element is inserted into a table of size 7, then the table is now over 70% full.

As a result, we create a NEW table of size 17,

because it's the FIRST PRIME that's TWICE as LARGE as the old table size

The new hash function is h(x) = x mod 17

The old table is scanned, and the 5 elements are inserted into the new table.

This operation is called REHASHING.

Rehashing is a very expensive operation, with a running time of O(N).

(N elements to rehash, and the table size is roughly 2N)

However, it's not too bad because it happens very infrequently.

-------------------------------------------------------------------------------------------------------------------------------------------------

Rehashing can be implemented in several ways with quadratic probing.

1. Rehash AS SOON AS the table is HALF FULL. (One extreme)

OR

2. Rehash ONLY when an INSERTION FAILS (Other extreme)

OR

3. Rehash when the table reaches a CERTAIN LOAD FACTOR (Middle-ground)

The third strategy, implemented with a good cutoff, could potentially be best.

Ex: (We CALL this method when one of the chosen conditions ABOVE are met)

/\*\*

\* Rehashing for quadratic probing hash table

\*/

private void rehash()

{

HashEntry<AnyType> [] oldArray = array;

// Create a new double-sized, empty table

allocateArray( nextPrime( 2 \* oldArray.length ) ); //NOTICE: it's the NEXT PRIME

currentSize = 0; // that's TWICE as LARGE

// Copy table over

for( int i = 0; i < oldArray.length; i++ )

if( oldArray[ i ] != null && oldArray[ i ].isActive )

insert( oldArray[ i ].element );

}

--------------------------------------------------------------------------------------------------------------------------------------------------

Rehashing for separate chaining hash tables is similar.

Ex:

/\*\*

\* Rehashing for separate chaining hash table

\*/

private void rehash()

{

List<AnyType> [] oldLists = theLists;

// Create new double-sized, empty table

theLists = new List[ nextPrime( 2 \* theLists.length ) ];

for( int j = 0; j < theLists.length; j++ )

theLists[ j ] = new LinkedList<>();

// Copy table over

currentSize = 0;

for( int i = 0; i < oldLists.length; i++ )

for( AnyType item : oldLists[ i ] ) // NOTICE how we insert each element of

insert( item ); // EACH of the LINKED LIST ELEMENTS

} // of the ORIGINAL HashTable List

// into the NEW List

---------------------------Hash Tables in the Standard Library--------------------------

The Standard library includes hash table implementations of "Set" and "Map",

namely HashSet and HashMap.

The items in the HashSet (or the keys in the HashMap) must provide an "equals" and a "hashCode" method

as described earlier.

The HashSet and HashMap are currently implemented using SEPARATE CHAINING HASHING.

These classes can be used if it is NOT IMPORTANT for the entries to be viewable in SORTED ORDER.

-------------------------------------------------------------------------------------------------------------------------------------------------

There were three maps this textbook has apparently showcased already:

1. A map in which the key is a WORD LENGTH, and the value is a collection of all words of that word length

2. A map in which the key is a REPRESENTATIVE, and the value is a collection of all words with that REPRESENTATIVE

3. A map in which the key is a WORD,

and the value is a collection of all words that differ ONLY ONE CHARACTER from that word.

Because the order in which word lengths are processed DOES NOT MATTER, the first map can be a HASHMAP.

Because the representatives are NOT EVEN NEEDED after the second map is built, the second map can be a HASHMAP

The third map can also be a HASHMAP, unless we want to spell out something alphabetically (as apparently the

program did in whatever example the textbook is referencing here)

The performance of a HASHMAP can often be SUPERIOR to that of a TREEMAP, but it's hard to know for sure

without writing the code both ways.

As a result, in cases where EITHER a HashMap or TreeMap is acceptable, it's preferable to declare variables using the

interface type MAP.

And THEN change the instantiation from a TreeMap to a HashMap, and perform timing tests.

-------------------------------------------------------------------------------------------------------------------------------------------------

In Java, library types that can be reasonably inserted into a HashSet or as KEYS into a HASHMAP already have

"equals" and "hashCode" methods defined

Because the expensive part of the hash table operations is computing the HASH CODE,

EACH String object stores INTERNALLY the value of its hashCode.

Initially, the property is zero for each object.

However, if invoked, the value is remembered.

THUS: if hashCode is computed on the same String object a second time, we can avoid the expensive computation.

This technique is called CACHING THE HASH CODE

Ex:

public final class String

{

public int hashCode()

{

if( hash != 0 )

return hash;

for( int i = 0; i < length(); i++ )

hash = hash \* 31 + (int) charAt( i );

return hash;

}

private int hash = 0; }

NOTE: Caching the hash code ONLY works because Strings are IMMUTABLE (unable to be changed)

If the String were allowed to change, it would invalidate the hashCode, and the hashCode would reset back to 0.

-------------------------------------------------------------------------------------------------------------------------------------------------

One situation where caching the hash code helps occurs during REHASHING,

because all the Strings involved in the rehashing have already had their hash codes CACHED.

However, caching the hash code does not help in the representative map for the word changing example

(the second bullet above - whatever example the textbook happens to be referring to......)

Each of the representatives is a different String computed by removing a character from a larger String,

thus each individual String has to have its hash code computed separately

HOWEVER, in the third map, caching the has code DOES HELP, because the keys are only STRINGS that were stored

in the original array of Strings.

-------------------------------Double Hashing---------------------------------------

For double hashing, the popular collision resolution function is:

f( i ) = i \* hash2(x)

We apply a second hash function to x and probe at a distance hash2(x), 2hash2(x), etc..

The function must NEVER evaluate to zero.

Also, ALL cells must be able to be probed.

A function such as:

hash2(x) = R - (x mod R), with R a prime smaller than TableSize

will work well.

For example, if we choose R to be 7 and the table ranges from 0 to 9, and we get a collision while trying to insert 49:

1. hash2(x) is computed. --> hash2(49) = 7 - 0 = 7

2. 49 is inserted whatever number of spaces down calculated by hash2(x).

In this case, it's 7. So we move 49 seven spaces down, or to position 6.

(Originally, we attempted to insert 49 at "9").

3. If a collision occurs again, we move it 7 spaces down again. Therefore, the next alternative would be at 3.

4. Continue until an empty space is found.

Generally there aren't really many bad cases here

Although in this case the table size was not prime, it is IMPORTANT that we make sure the table size IS PRIME.

- Table size was 10 here for mere convenience of computing the hash function as an example.

Ex:

If we attempt to insert 23 into the table, and it happens to collide, then the hash2(23) value is 5.

Since the table size is 10, the only other alternative location is 0.

If 0 is already taken, then you're screwed.

THUS: If the table size is NOT PRIME, then it's possible to RUN OUT of ALTERNATIVE LOCATIONS prematurely.

If double hashing is correctly implemented, then the expected number of probes is almost the SAME as for a

RANDOM COLLISION RESOLUTION STRATEGY.

Quadratic probing, however, does not require the use of a second hash function and is thus likely to be simpler

and faster in practice.

Especially for keys like strings whose hash functions are expensive to compute.

--------------------------------Separate Chaining and Load Factor----------------------

If, when an element is inserted, it hashes to the same value as an already inserted element,

then we have a COLLISION and a need to resolve it.

Two of the simplest methods for dealing with this are: Separate chaining and Open Addressing.

---------------------------------------------Separate Chaining---------------------------------------------------------

Separate chaining is keeping a LIST of all elements that hash to the same value.

We can use the standard library list implementations.

To perform a search, we use the hash function to determine which list to traverse, and then search

the appropriate list.

To perform an insert:

We check the appropriate list to see whether the element is already in place.

If duplicates are expected, a counter is kept that increments in the event of a match.

If the element is new, it's inserted at the front of the list

(convenient and also because recently inserted elements are most likely to be accessed

in the near future)

NOTE: The hash table stores an ARRAY of LINKED LISTS, which are allocated in the CONSTRUCTOR.

--------MyHash-----Appropriate Objects to be Used in a Hash Table----------

Just like the binary search tree only works for objects that are "Comparable", the hash tables we use in this chapter

work ONLY for objects that provide:

An appropriate "equals" method

A "hashCode" method that returns an "int"

The hash table can then scale this "int" into a suitable array index via "MyHash"

Ex:

private int myhash( AnyType x )

{

int hashVal = x.hashCode();

hashVal %= theLists.length;

if( hashVal < 0 )

hashVal += theLists.length;

return hashVal; - This is the INDEX value of the hash table ARRAY

}

An object that could be stored in a hash table would be something like an object of the "Employee" class given below

Ex:

public class Employee

{

public boolean equals( Object rhs )

{ return rhs instanceof Employee && names.equals( ((Employee)rhs).name ); }

public int hashCode()

{ return name.hashCode(); } - NOTICE: it uses the hashCode defined in the standard

String class

private String name;

private double salary;

private int seniority;

//Additional fields and methods, etc....

}

This "Employee" class provides an "equals" method and a "hashCode" method BASED on the Employee's "name"

NOTE: The hashCode for this class works by using the "hashCode" defined in the Standard String class.

- That hashCode is BASICALLY like a portion of a hash function we saw previously, so essentially like this:

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 );

//Removed portion

/\* hashVal %= tableSize; NOTICE: This is instead the function of "myhash"

\*if( hashVal < 0 ) as seen earlier

\* hashVal += tableSize;

\*/

return hashVal;

}

----Constructors and makeEmpty-----

//Construct the hash table

public SeparateChainingHashTable()

{

this( DEFAULT\_TABLE\_SIZE );

}

/\*\*

\* Construct the hash table.

\* @param size approximate table size

\*/

public SeparateChainingHashTable( int size )

{

theLists = new LinkedList[ nextPrime( size ) ]; - The "array" is a linked list here

for( int i = 0; i < theLists.length; i++ )

theLists[ i ] = new Linkedlist<>();

}

NOTICE:

As mentioned before, we want to keep the table size PRIME to ensure a good distribution.

EACH element of the list is ALSO a LINKED LIST.

/\*\*

\* Make the hash table logically empty.

\*/

public void makeEmpty()

{

for( int i = 0; i < theLists.length; i++ )

theLists[ i ].clear();

currentSize = 0;

}

---------------------------Contains-------------------------

For the "contains" method of the hash table, we use the "contains" method from the Standard LinkedList class.

./\*\*

\* Find an item in the hash table.

\* @param x the item to search for.

\* @return TRUE if x is NOT FOUND

\*/

public boolean contains( AnyType x )

{

List<AnyType> whichList = theLists[ myhash( x ) ];

return whichList.contains( x ); - Using the "contains" method from

LinkedLists

}

-------------------------Insert----------------------------------------

In the insertion routine, if the item to be inserted is ALREADY PRESENT, then we DO NOTHING.

Otherwise, we place it in the list.

The element can be placed anywhere in the list, but using "add" is most convenient

in our case. (Adding it to the end of the list)

/\*\*

\* Insert into the hash table.

\* If the item is already present, then do nothing.

\* @param x the item to insert

\*/

public void insert( AnyType x )

{

List<AnyType> whichList = theLists[ myhash( x ) ];

if ( !whichList.contains( x ) )

{

whichList.add( x );

//Rehash: we study this later in the next chapter section

if ( ++currentSize > theLists.length )

rehash();

}

}

--------------------------Remove--------------------------------------

/\*\*

\* Remove from the hash table.

\* @param x the item to remove

\*/

public void remove( AnyType x )

{

List<AnyType> whichList = theLists[ myhash( x ) ];

if( whichList.contains( x ) )

{

whichList.remove( x );

currentSize--;

}

}

-------------------------------------------Separate Chaining CLASS SKELETON---------------------------------------------------------------

The class skeleton for the implementation of separate chaining is shown below:

Ex:

public class SeparateChainingHashTable<AnyType>

{

public SeparateChainingHashTable()

{

}

public SeparateChainingHashTable( int size )

{

}

public void insert( AnyType x )

{

}

public void remove( AnyType x )

{

}

public boolean contains( AnyType x )

{

}

public void makeEmpty()

{

}

private static final int DEFAULT\_TABLE\_SIZE = 101;

private List<AnyType> [] theLists;

private int currentSize;

private void rehash()

{

}

private int myhash( AnyType x )

{

}

private static int nextPrime( int n )

{ See online code }

private staticc boolean isPrime( int n )

{ See online code }

-------------------------------------------------------------------------------

-------------------------------------------------------------------------------

Any scheme can be used besides Linked lists to resolve the collisions.

Ex: binary search tree, another hash table, etc...

HOWEVER, we EXPECT that if the:

TABLE is LARGE

&

HASH FUNCTION is GOOD

ALL LISTS should be SHORT

-----------------------------------------------------------------------------

---------------------------Load Factor------------------------------------

Load factor, LAMBDA is defined as:

Ratio of the number of elements in the hash table to the table size

In the example above, LAMBDA is 1.0.

The average length of a list is LAMBDA.

The effort required to perform a search is:

the constant time required to evaluate the hash function + the time to traverse the list

In an UNSUCCESSFUL search, the number of nodes to examine is LAMBDA on average.

A SUCCESSFUL search requires that about 1 + (LAMBDA/2) links be traversed.

Notice, for example, that the list being searched contains the one node that STORES the match PLUS

zero or more OTHER nodes.

The expected number of "other nodes" in a table of N elements and M lists is:

(N - 1)/M = LAMBDA - 1/M

On AVERAGE, HALF the "other nodes" are searched, so combined with the matching node, we obtain an

average search cost of:

1 + LAMBDA/2 nodes.

THUS: The table size is not really important, but the LOAD FACTOR IS.

General rule:

Make the table size about as large as the number of elements expected

In other words: LAMBDA ~= 1

In the previous code for "insert", we EXPAND the table size by calling "rehash", discussed ater.

AS ALREADY MENTIONED, it's good to keep the table size PRIME to ensure a GOOD DISTRIBUTION.