CompSci 261P | Data Structures
Project 1: Hashing Algorithms
Shefali Gupta | 57806943

Contents:

1) Linear Hashing

The main idea behind a Linear Hash Table is that we would, ideally, like to store the element x with hash value i = hash(x) in the table location t[i]. If we cannot do this (because some element is already stored there) then we try to store it at location t[(i + 1) mod t.length]; if that's not possible, then we try t[(i+ 2) mod t.length], and so on, until we find a place for x.

Algorithm pseudo-codes:

Data structures used for implementation (Java):

- Dynamic Array

Insert operation (Pseudocode):

(T here refers to the hash table)

```
boolean insert(T x) {
if (find(x) != null) return false;
if (2*(q+1) > t.length) resize(); // max 75% occupancy
int i = hash(x);
while (t[i] != null && t[i] != del)
i = (i == t.length-1) ? 0 : i + 1; // increment i if (t[i] == null) q++;
n++;
t[i] = x;
return true;
}
```

Delete operation (Pseudocode):

- Eager delete operation implemented.

//Code snippet

```
public void remove(String key)
	{
		if(!contains(key))
			return;
		//Find position of key and delete
		int i=hash(key);
		while(!key.equals(keys[i]))
			i=(i+1)%tableSize;
		keys[i]=null;
		//Rehash all keys //Eager delete
		for(i=(i+1)%tableSize;keys[i]!=null;i=(i+1)%tableSize)
		{
			String temp=keys[i];
			keys[i]=null;
			currentSize--;
			insert(temp);
		}
		currentSize--;
	}
```

Search operation(Pseudocode):

```
T Search(T x) {
int i = hash(x);
while (t[i] != null) {
if (t[i] != del && x.equals(t[i]))
return t[i];
i = (i == t.length-1) ? 0 : i + 1; // increment i
} return null;
}
```

Rehashing condition (Load factor >0.75):

Rehashing algorithm (Java code snippet):

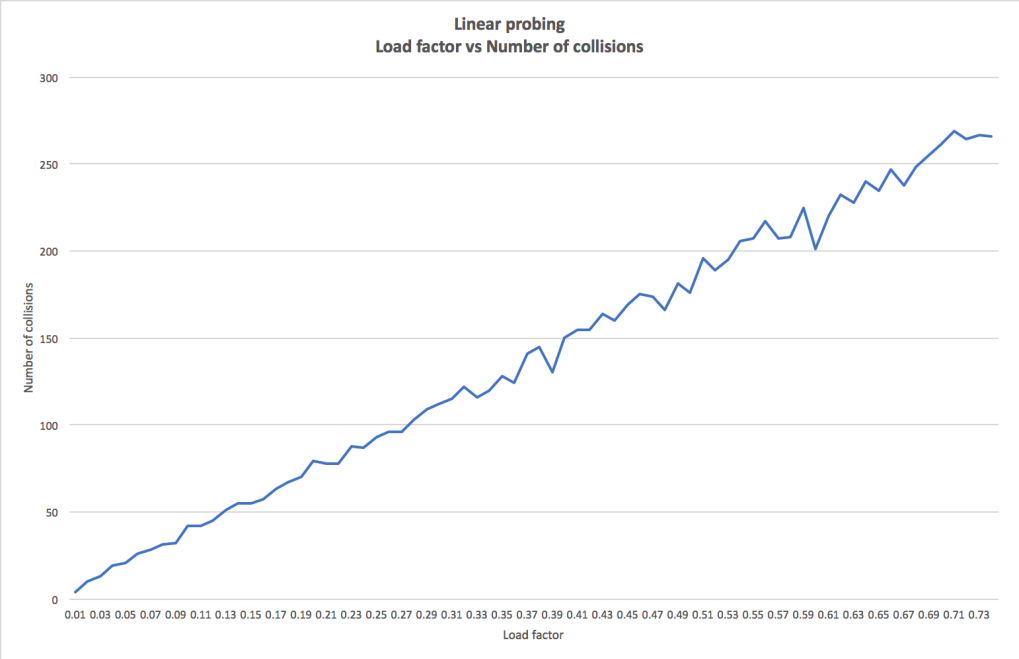
```
private void rehash()
	{
		newTableSize=tableSize*2;
		String temp[]=new String[tableSize];
		for(int i=0;i<tableSize;i++)
		{
			temp[i]=keys[i];
		}
		keys=new String[tableSize*2];
		int oldTableSize=tableSize;
		this.tableSize=tableSize*2;
		for(int i=0;i<oldTableSize;i++)
		{
			if(temp[i]!=null)
				insertIntoNewHashTable(temp[i]);
		}
	}
```

Empirical Analysis of Linear probing:

Load Factor = (Number of keys in the hash table / Maximum number of buckets in hash table)

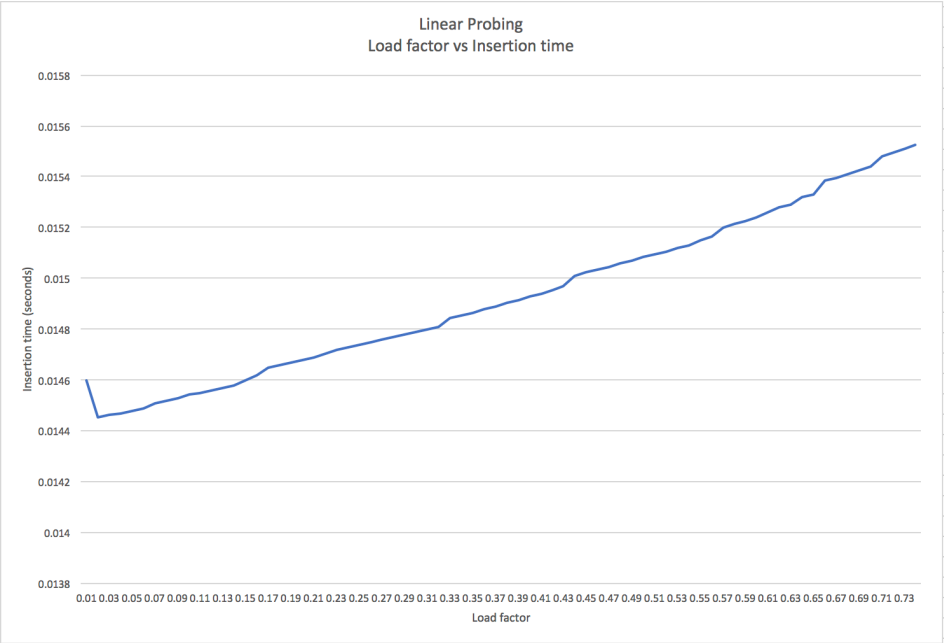
Graph: Load factor vs Number of collisions (for say, 1000 elements):

(Hash Table size kept, such as to pause rehashing, and capture a plot of increasing load factor with number of collisions).
As the load factor increases, number of collisions (on inserting an element) increase.
Amortized running time : O(n)



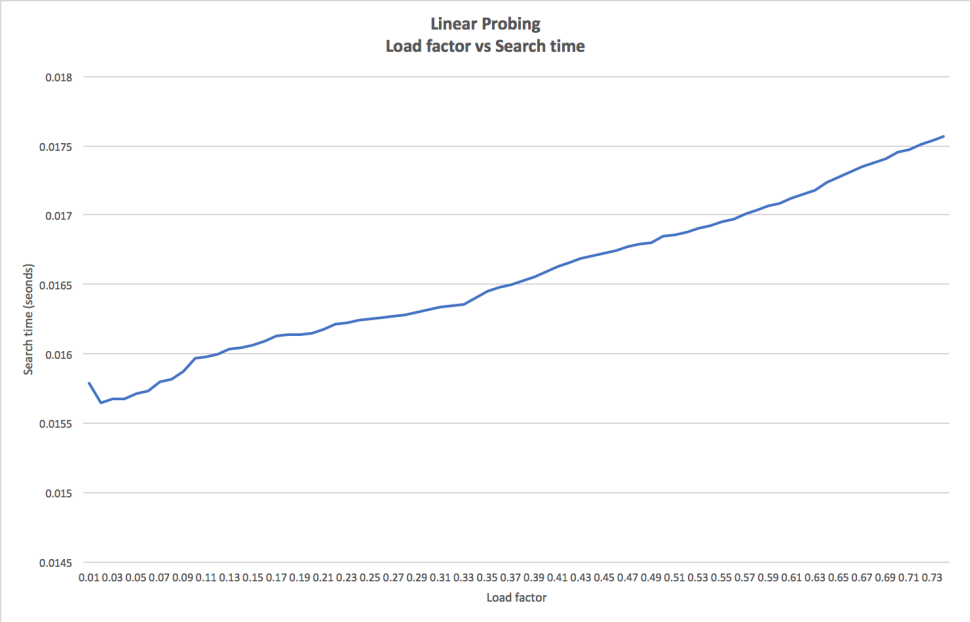
Graph: Load Factor vs Insertion Time

Amortized running time : Insertion time with respect to load factor, is almost O(1).



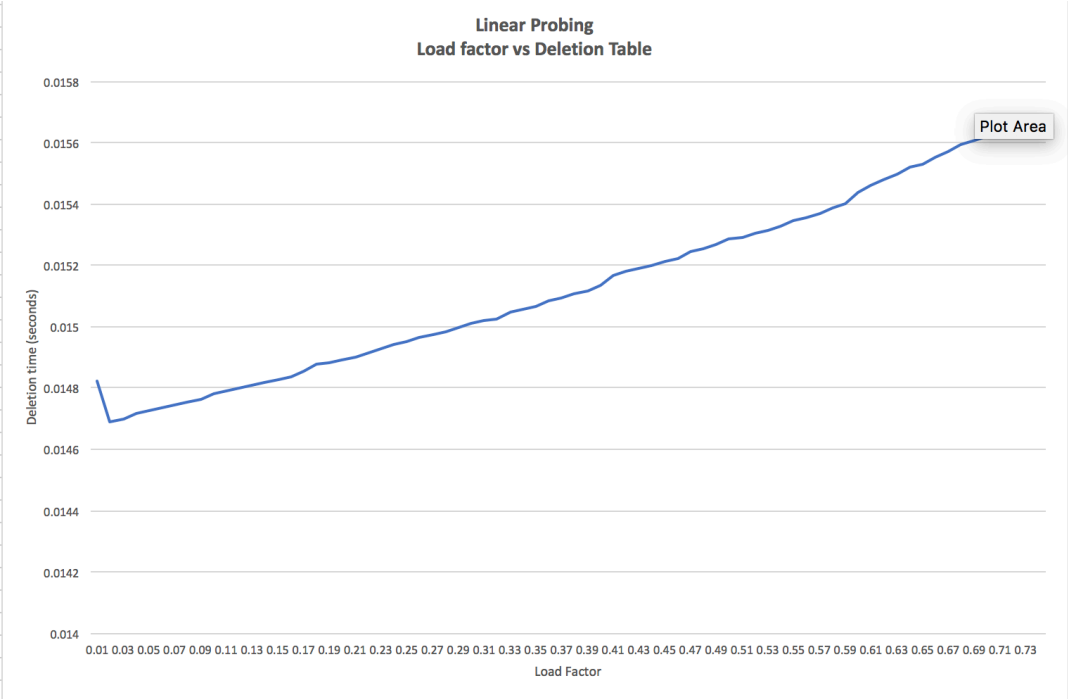
Graph: Load factor vs Search time

Amortized running time : O(1)



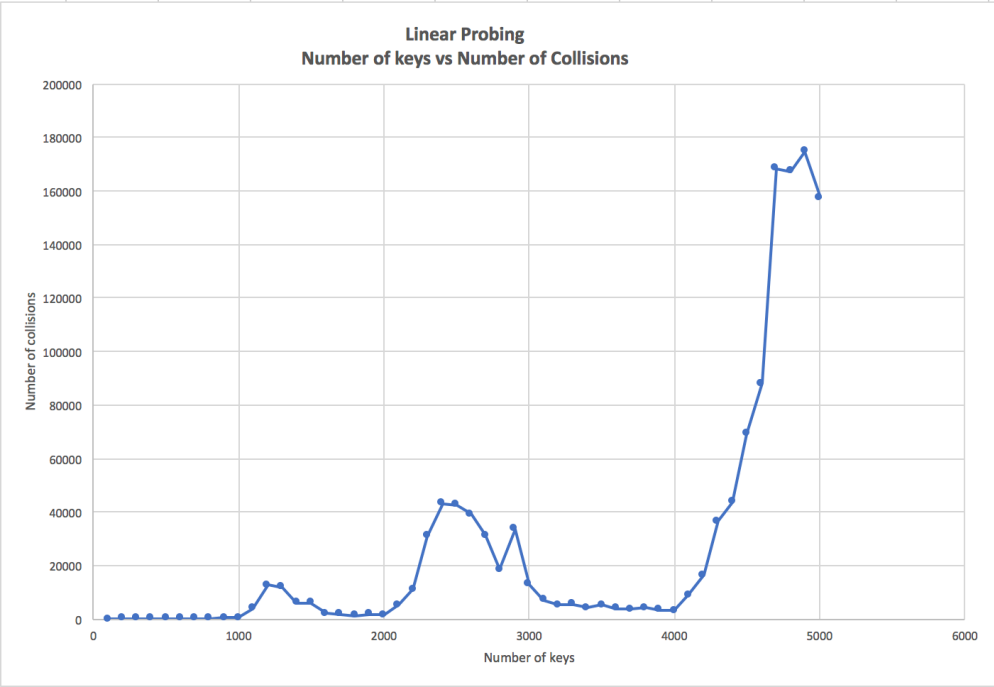
Graph: Load factor vs Deletion time

Amortized running time : O(1)



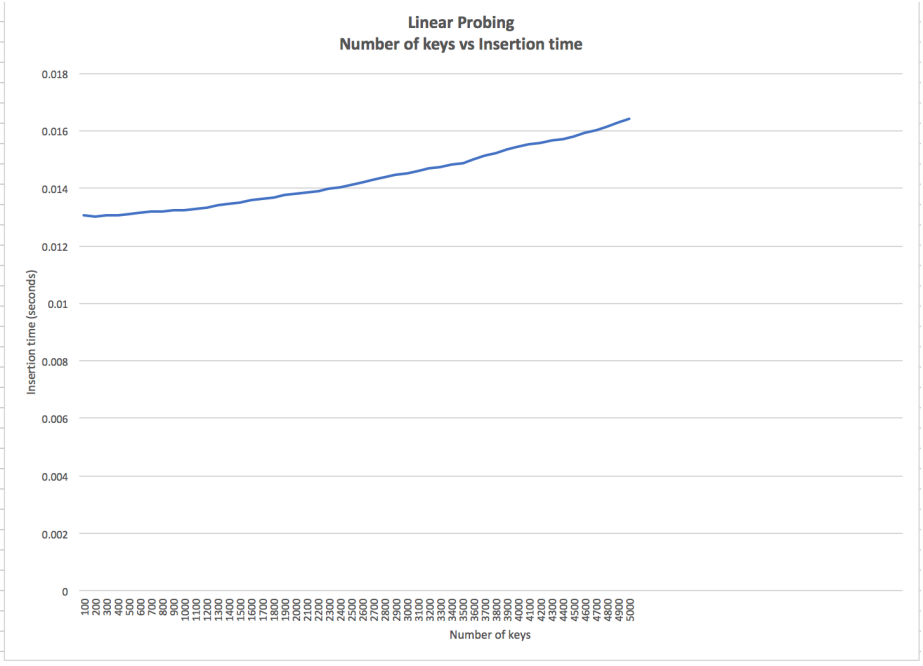
Graph: #Keys vs #Collisions (Insert operation)

Sudden decrease in number of collisions, is due to rehashing (and doubling size of hash table) on those points (number of keys in hash table).
As load factor reached beyond a threshold of 0.75, we rehash our tables.



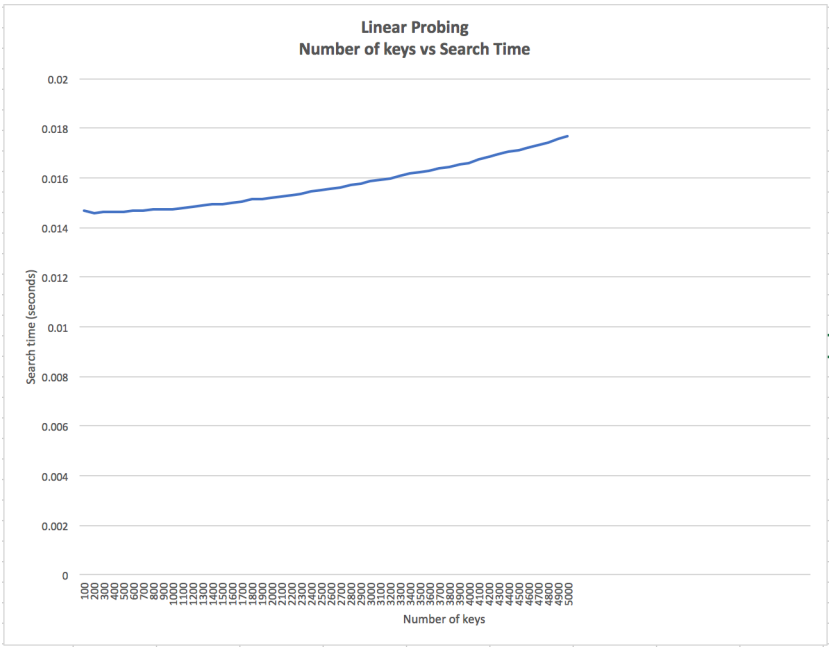
Graph: #Keys vs Insertion Time

Amortized running time : (Inserting 5000 keys in hash table) : O(1) operation



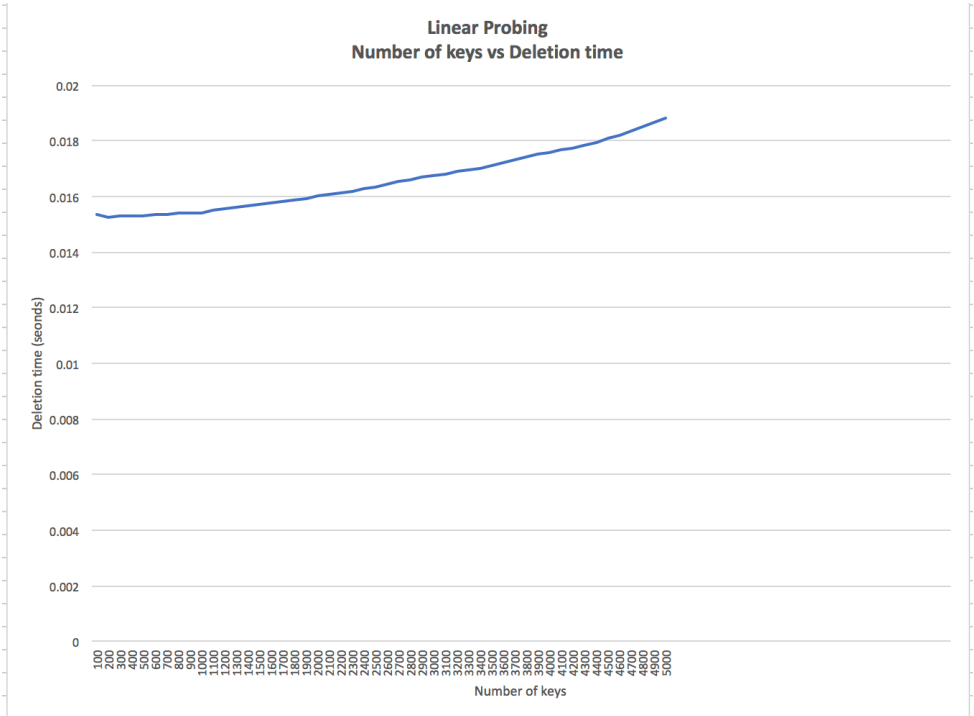
Graph: #Keys vs Search Time

Amortized running time : O(1)



Graph: #Keys vs Deletion Time

Amortized running time : O(1)



Advantages of linear hashing:

- It works and it is simple
- Single-level store: space efficient, sequential memory is fast.

Disadvantages:

- Degrades badly when α gets close to 1.

2) Chained Hashing

A Chained Hash Table data structure uses hashing with chaining to store data as an array, t, of lists. An integer, n, keeps track of the total number of items in all lists.

Data structures used for implementation (Java):

- Dynamic Array
- Linked List

Algorithm pseudo-codes:

Insert operation (Pseudocode):

```
boolean insert(T x)
{
if (find(x) != null) return false;
if (n+1 > t.length) resize();
 t[hash(x)].add(x);
n++;
return true;
}
```

Delete operation (Pseudocode):

```
T delete(T x)
{ Iterator it = t[hash(x)].iterator();
while (it.hasNext())
{ T y = it.next();
if (y.equals(x))
{ it.remove();
 n--;
return y;
} }
return null;
}
```

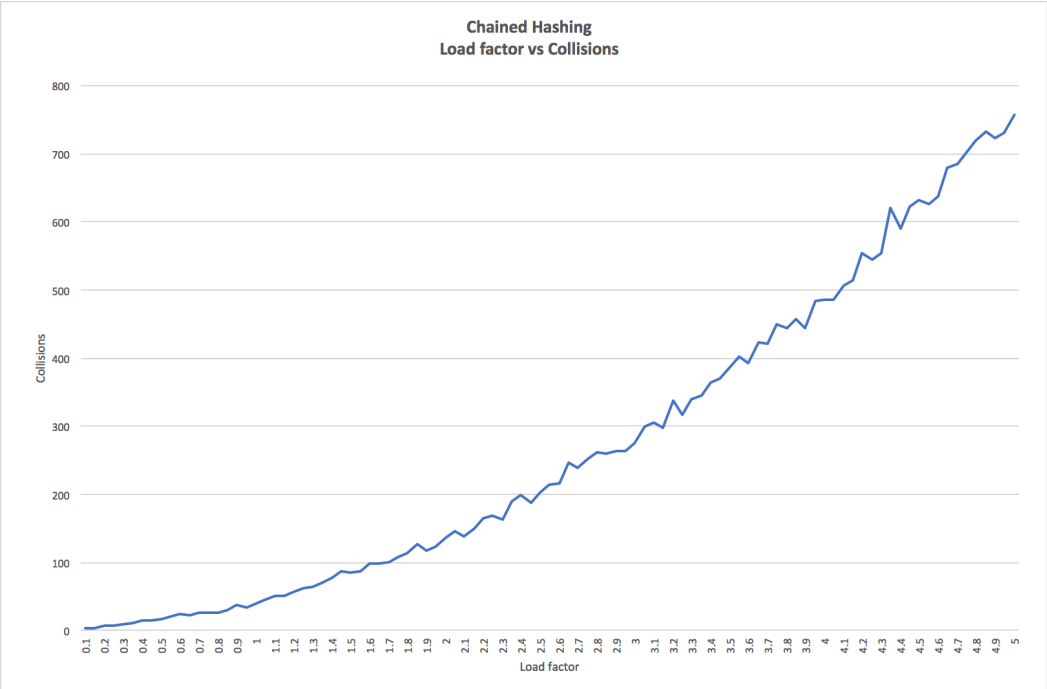
Search operation(Pseudocode):

```
T search(Object x) {
 for (T y : t[hash(x)])
 if (y.equals(x))
return y;
return null; }
```

Empirical Analysis:

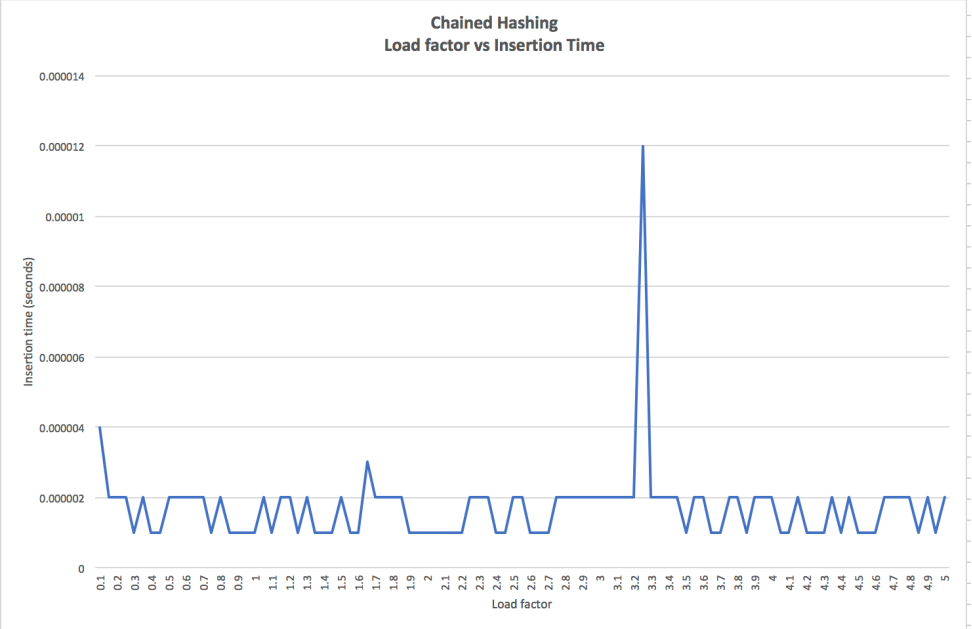
Graph: Load factor vs Number of collisions

Amortized running time : O(N) (As load factor increases, number of collisions increases.)



Graph: Load factor vs Insertion time

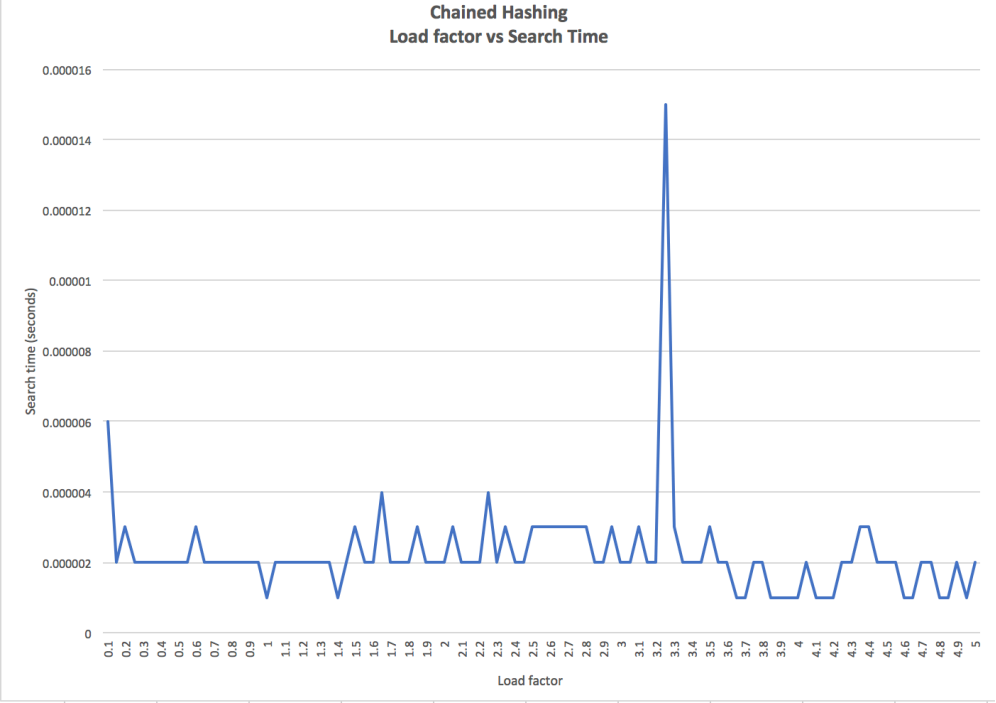
Amortized running time : O(1)



Graph: Load factor vs Search time

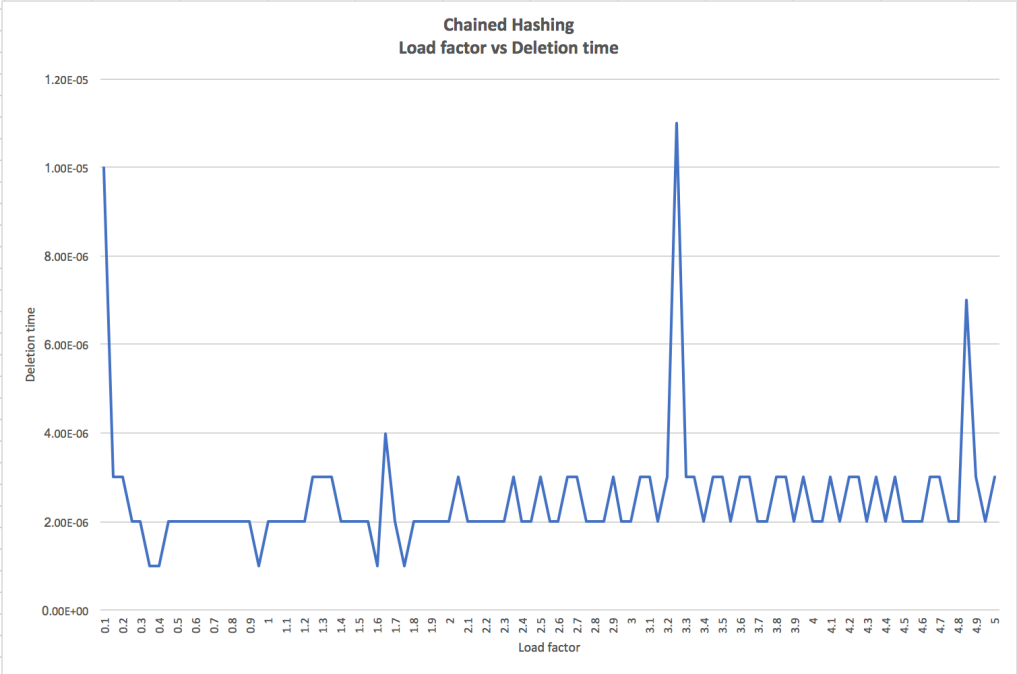
Amortized running time : O(1)

Sudden peaks in time arise, when a longer linked list is traversed to insert a key.



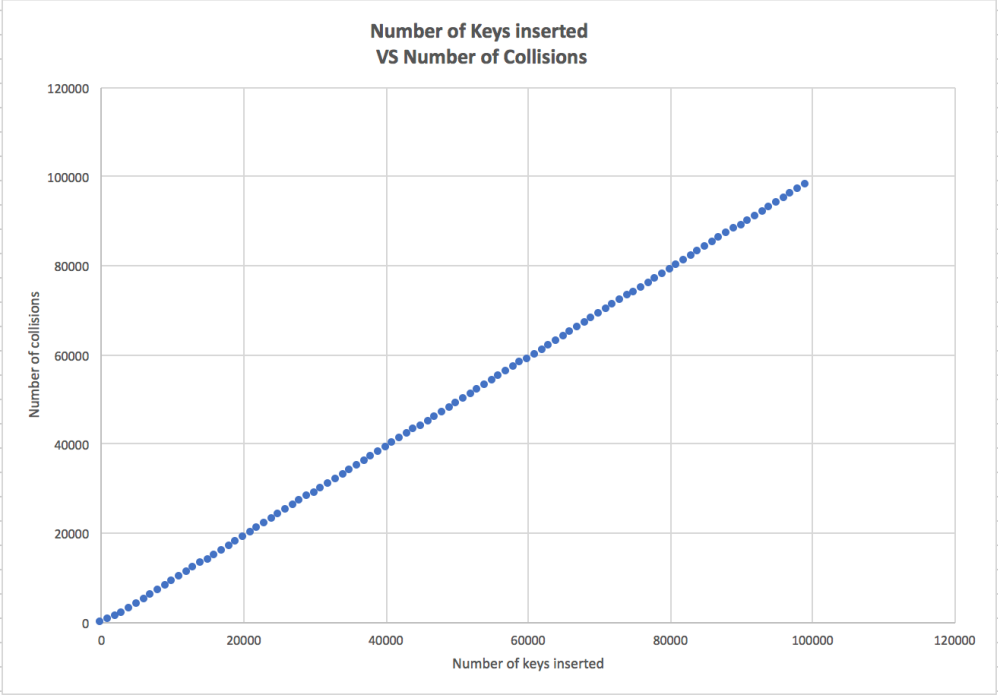
Graph: Load factor vs Deletion time

Amortized running time : O(1)



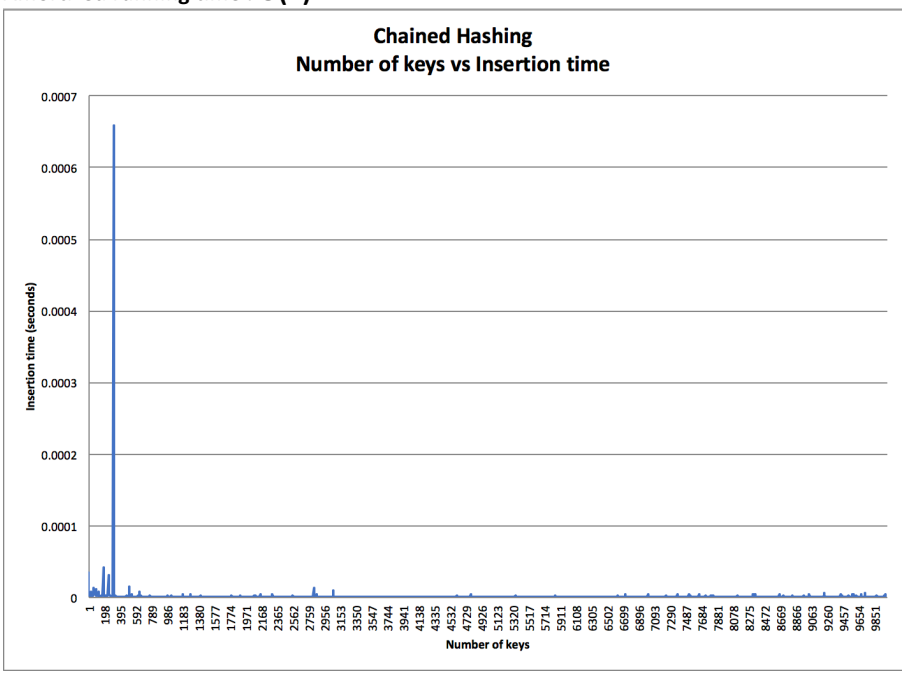
Graph: #Keys vs #Collisions (Insert operation)

Amortized running time : O(N)



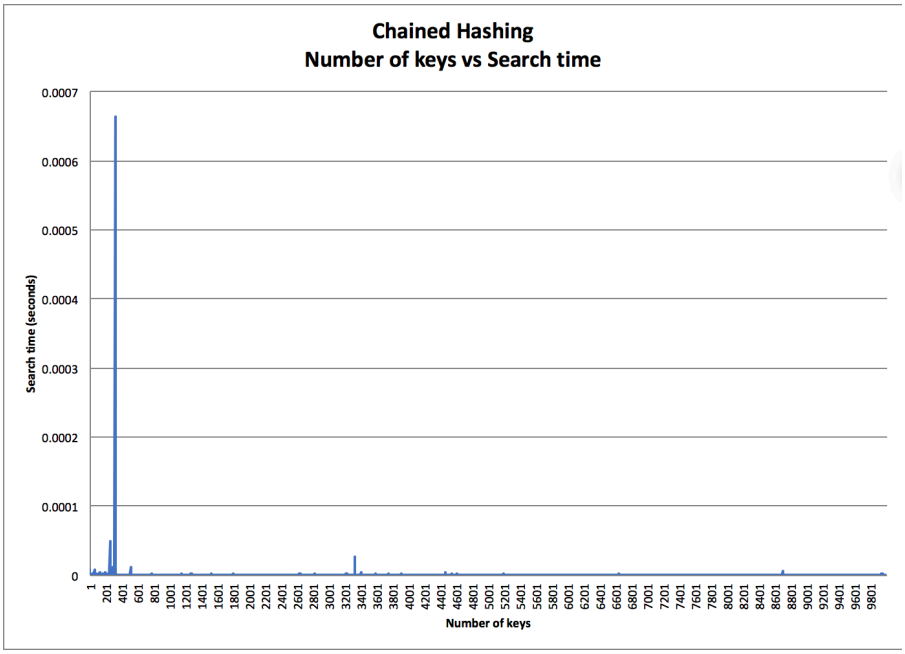
Graph: #Keys vs Insertion Time

Amortized running time : O(1)



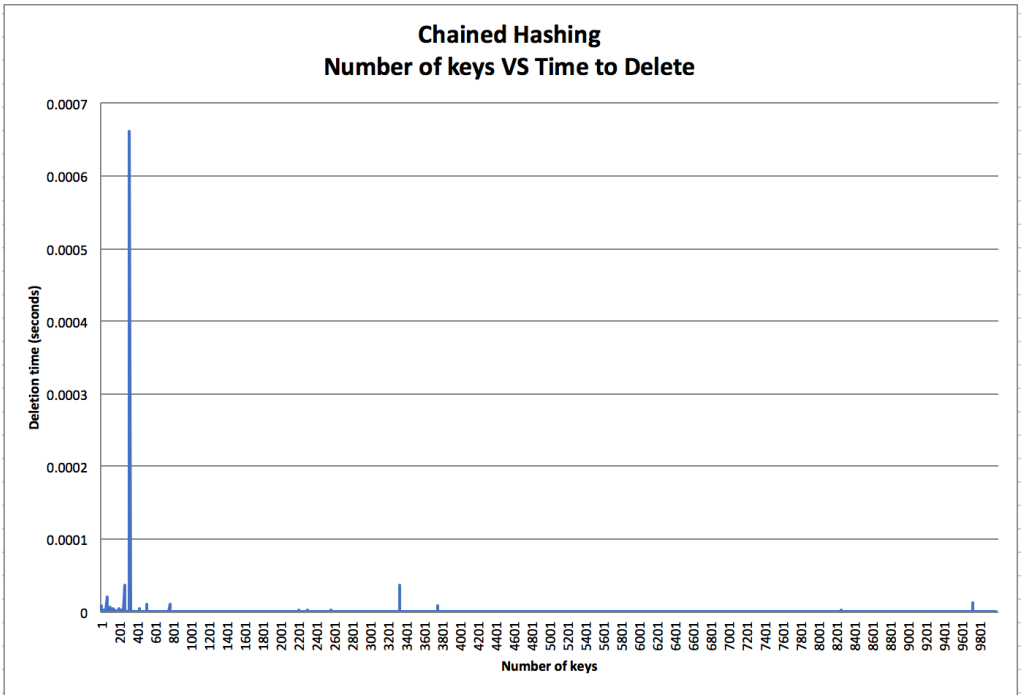
Graph: #Keys vs Search Time

Amortized running time : O(1)



Graph: #Keys vs Deletion Time

Amortized running time : O(1)



Advantages:

It works and it is simple

Disadvantages:

Extra space for collections.

Extra time for two-level access pattern.

3) Cuckoo Hashing

Cuckoo hashing is a scheme in computer programming for resolving hash collisions of values of hash functions in a table, with worst-case constant lookup time. The name derives from the behavior of some species of cuckoo, where the cuckoo chick pushes the other eggs or young out of the nest when it hatches; analogously, inserting a new key into a cuckoo hashing table may push an older key to a different location in the table.

Data structures used for implementation:

- Dynamic Arrays

Algorithm pseudo-codes

Insert operation (Pseudocode):

function insert(x)
if lookup(x) then return
loop MaxLoop times
x ↔ T1[h1(x)]
if x = ⊥ then return
x ↔ T2[h2(x)]
if x = ⊥ then return
end loop
rehash();
insert(x);
end

Delete operation (Pseudocode):

We look at the two possible locations, and if the element is there, we delete it.

Search operation(Pseudocode):

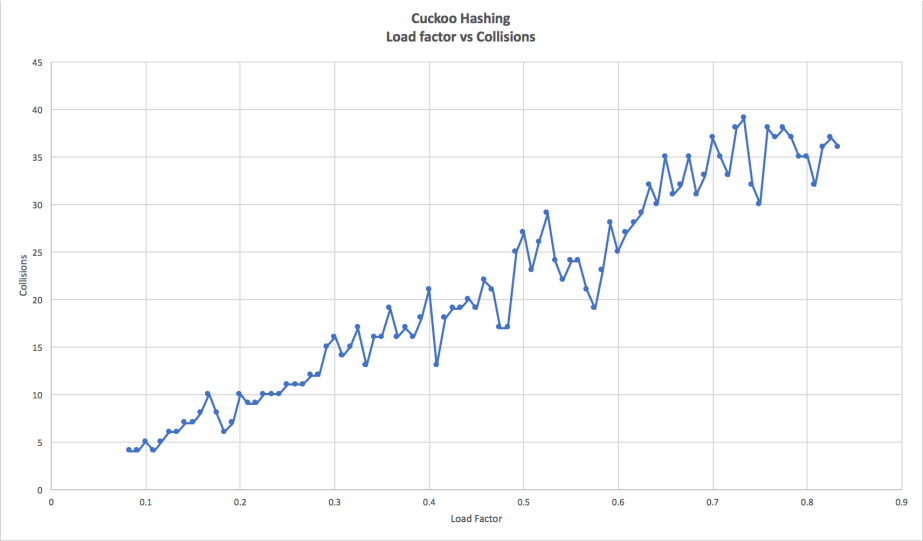
function search(x)
return T1[h1(x)] = x ∨ T2[h2(x)] = x
end

Rehashing condition - Detecting a cycle:
When there is an infinite loop, such that the key to be inserted originally, is again picked for displacement by another key in the hash table, and this loop never stops. In our project, we have taken a limit on the count of number of times this shifting of keys happens, as N (N: number of buckets in each hash table). When the counter reaches N, we stop the loop and rehash our hash tables.

Cuckoo Hashing: Empirical Analysis

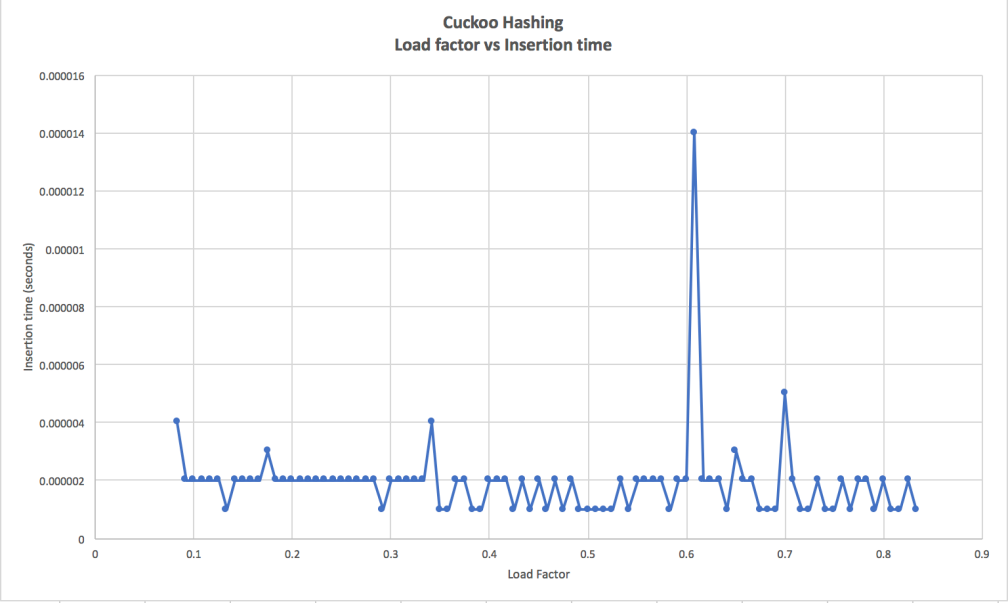
Graph: Load factor vs Collisions (Insert operation)

Amortized running time : O(N)



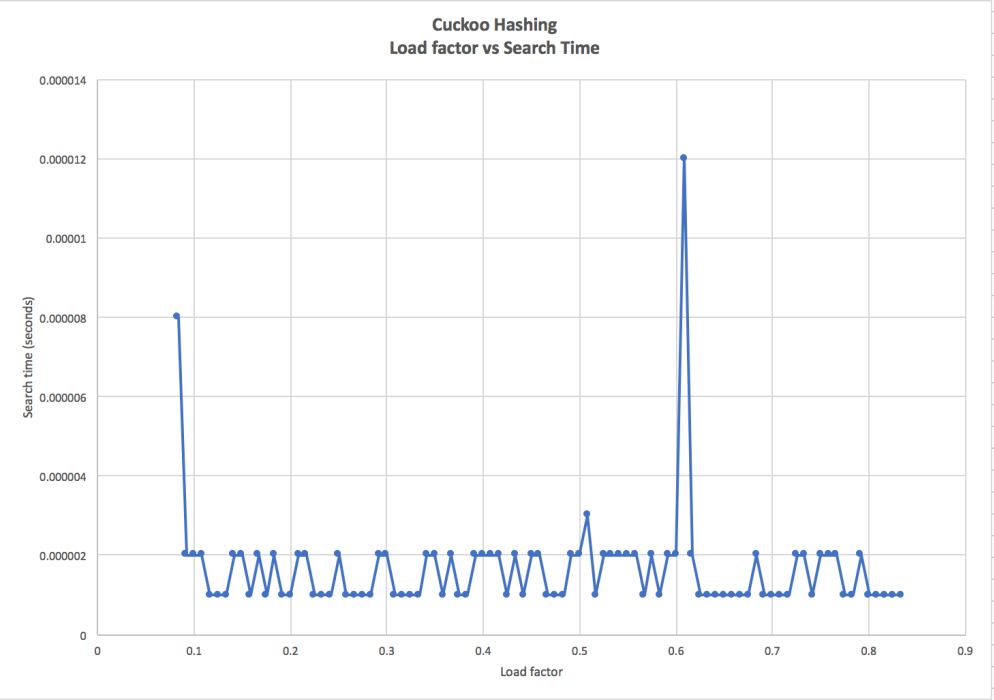
Graph: Load factor vs Insertion time

Amortized running time : O(1)



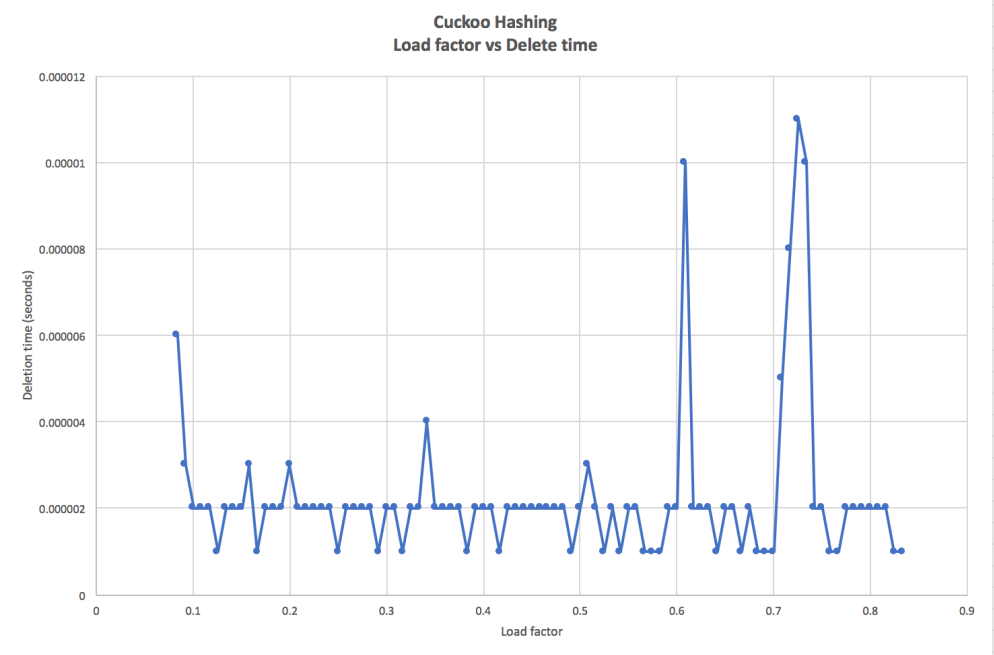
Graph: Load factor vs Search time

Amortized running time : O(1)



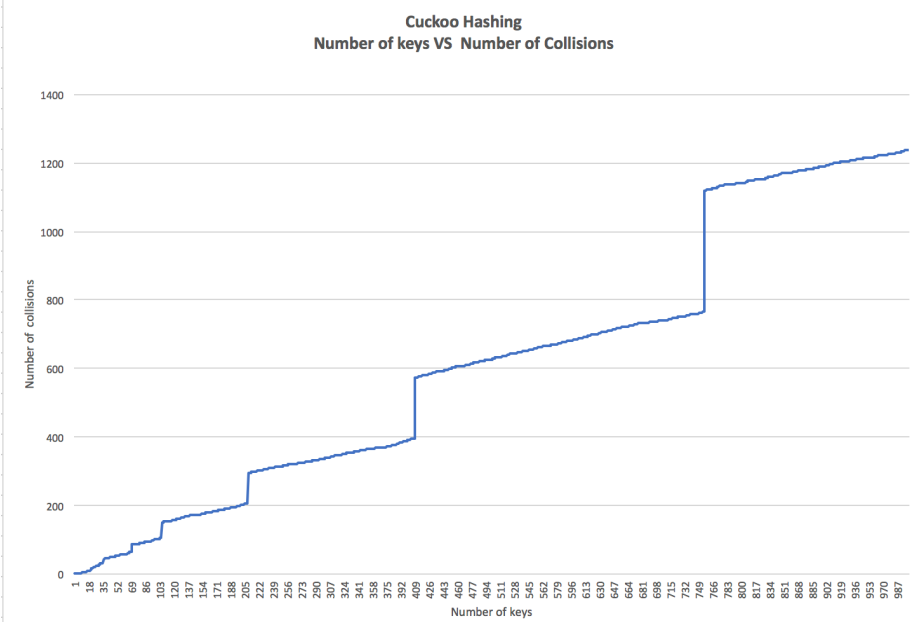
Graph: Load factor vs Deletion time

Amortized running time : O(1)



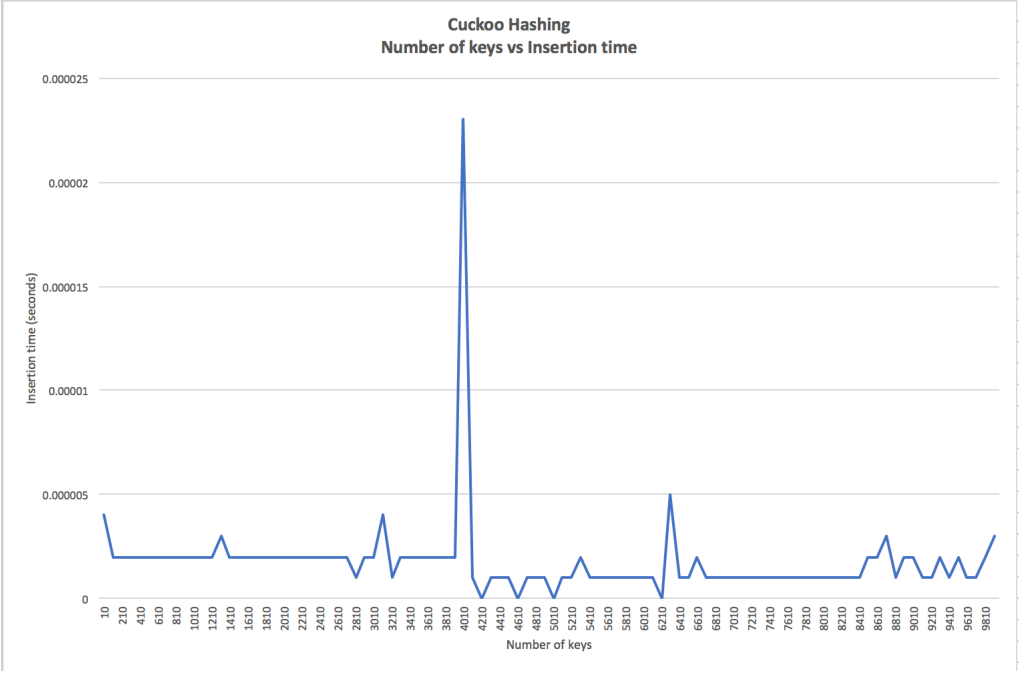
Graph: #Keys vs #Collisions (Insert operation)

Amortized running time : O(N)



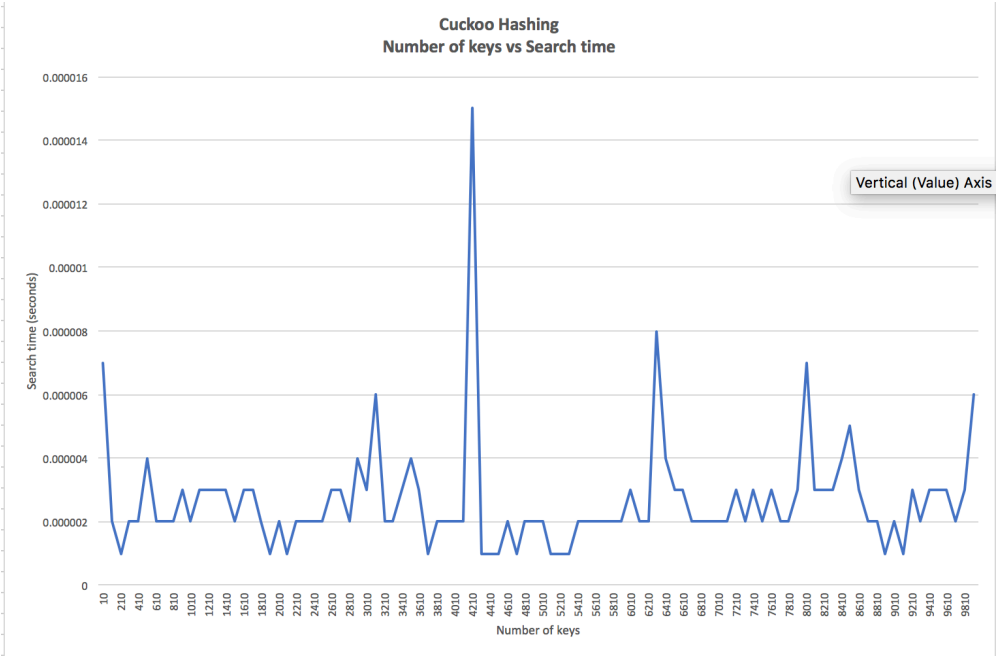
Graph: #Keys vs Insertion Time

Amortized running time : O(1)



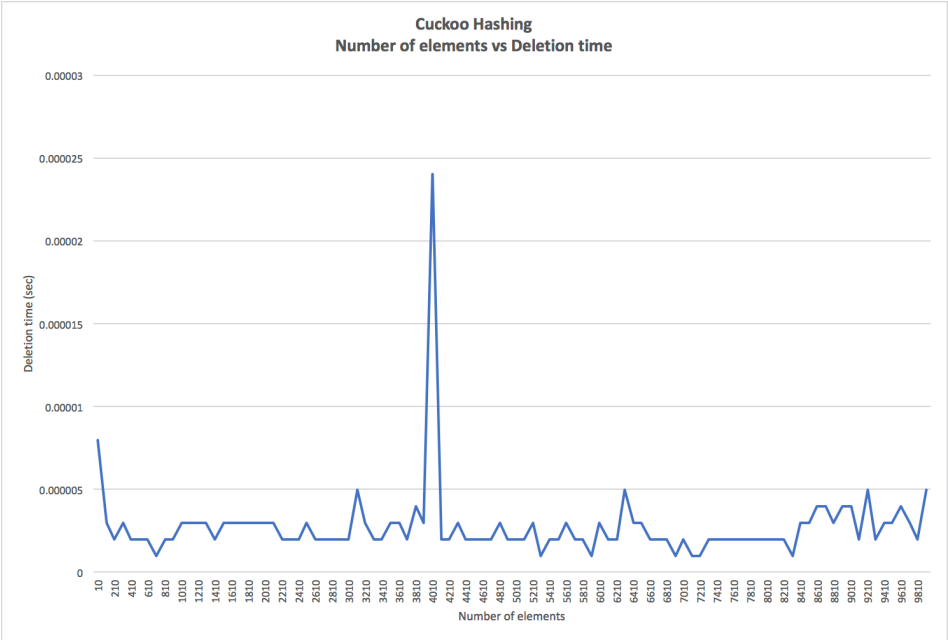
Graph: #Keys vs Search Time

Amortized running time : O(1)



Graph: #Keys vs Deletion Time

Amortized running time : O(1)



Advantages of cuckoo hashing:

- Guaranteed O(1) search

Disadvantages: Slower set operation, more complex.

4) Double Hashing

Double hashing is a collision resolving technique in **Open Addressed** Hash tables. Double hashing uses the idea of applying a second hash function to key when a collision occurs.

Algorithm pseudo-code:

Double hashing can be done using :
(hash1(key) + i * hash2(key)) % TABLE_SIZE
Here hash1() and hash2() are hash functions and TABLE_SIZE is size of hash table.
(We repeat by increasing i when collision occurs)

Data structures used for implementation:

- **Dynamic Array**

Insert operation (Pseudocode):

The insert algorithm for double hashing is:

1. Set indx = H(K); offset = H_2 (K)
2. If table location indx already contains the key, no need to insert it. Done.
3. Else if table location indx is empty, insert key there. Done.
4. Else collision. Set indx = (indx + offset) mod M.
5. If indx == H(K), table is full! (Throw an exception, or enlarge table.) Else go to 2.

Delete operation (Pseudocode):

Using both hash functions, loop through to get to the bucket where the key is placed, and delete it.

Search operation(Code):

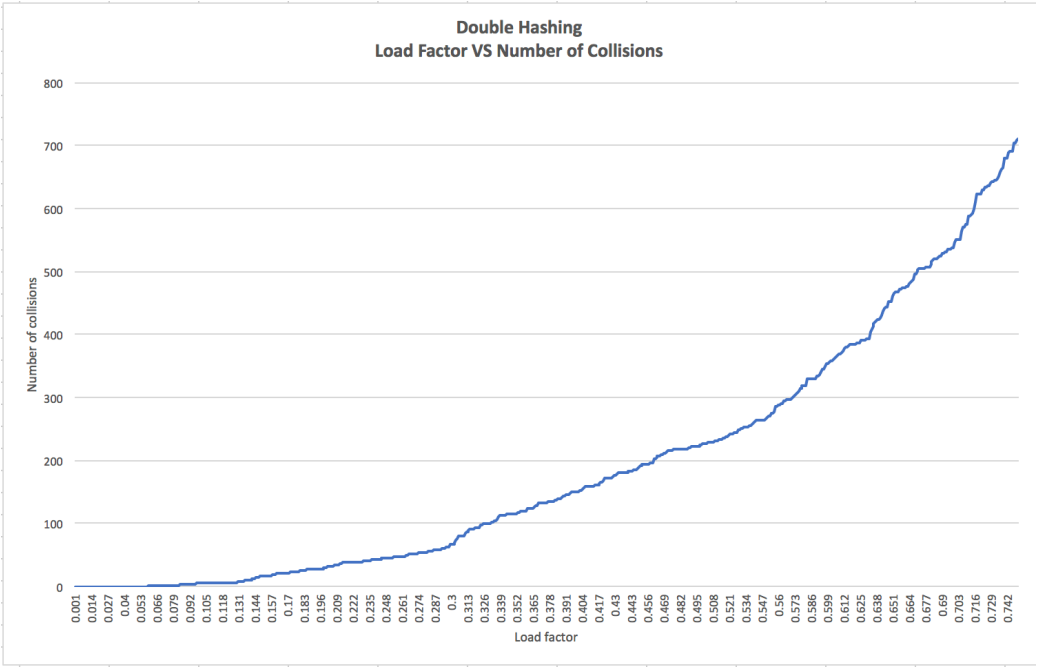
Same looping for search operation.

```
public boolean search(int key)
{
	int index=hash1(key);
	if(table[index]!=key)
	{
		int index2=hash2(key,getNextPrime());
		int i=1;
		while(true)
		{
			int newIndex=(index+i*index2)%TABLE_SIZE;
			if(table[newIndex]==key)
				return true;
			i++;
			if(i==TABLE_SIZE)
				return false;
		}
	}
	else
		return true;
}
```

Empirical Analysis of Double Hashing:

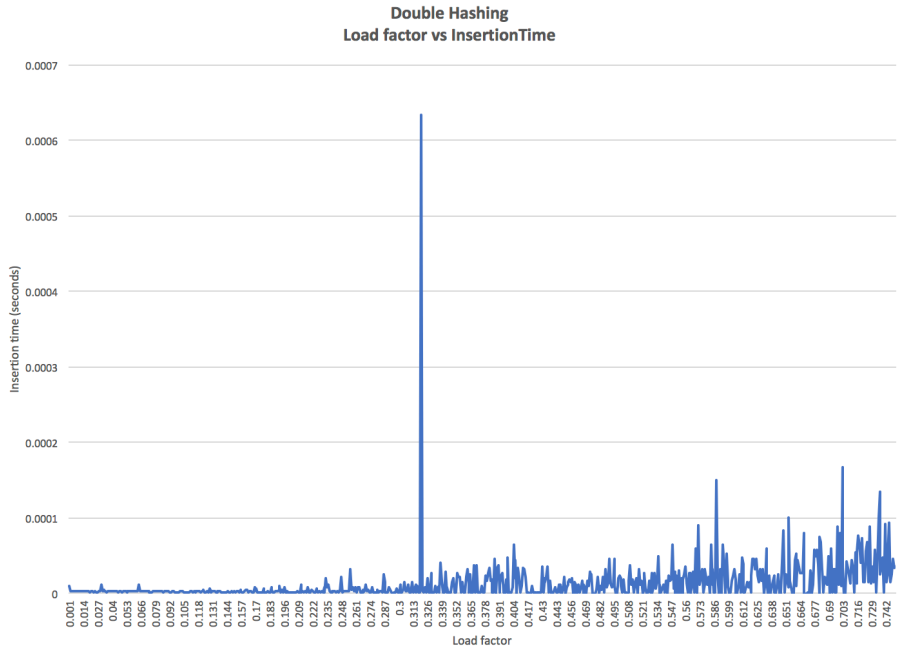
Graph: Load factor vs #Collisions (Insert operation)

Amortized running time : O(N)

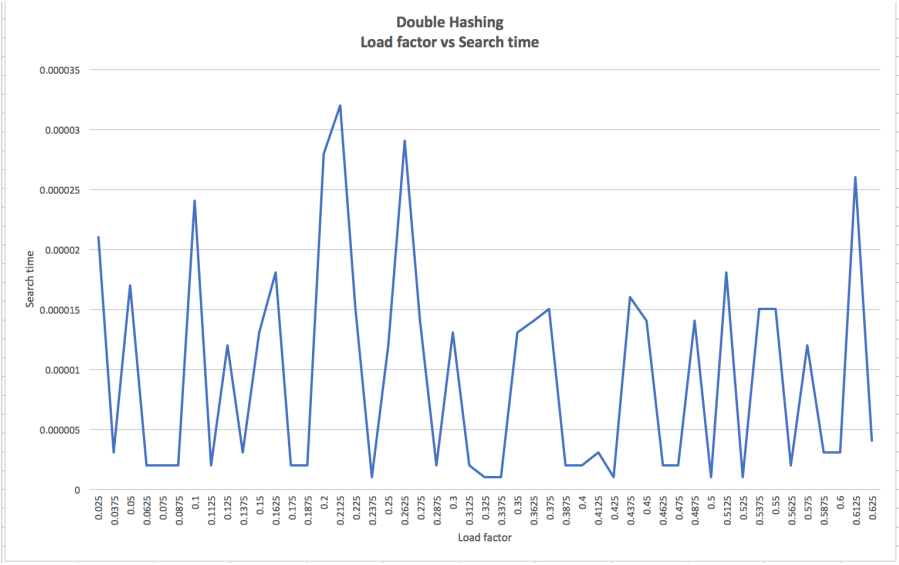


Graph: Load factor vs Insertion time

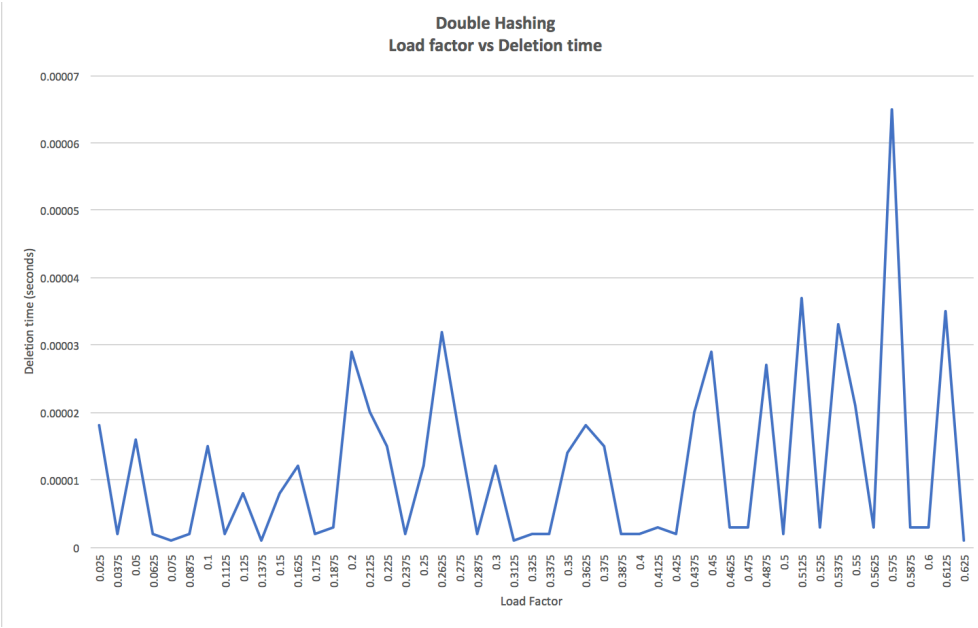
Amortized running time : O(1)



Graph: Load factor vs Search Time



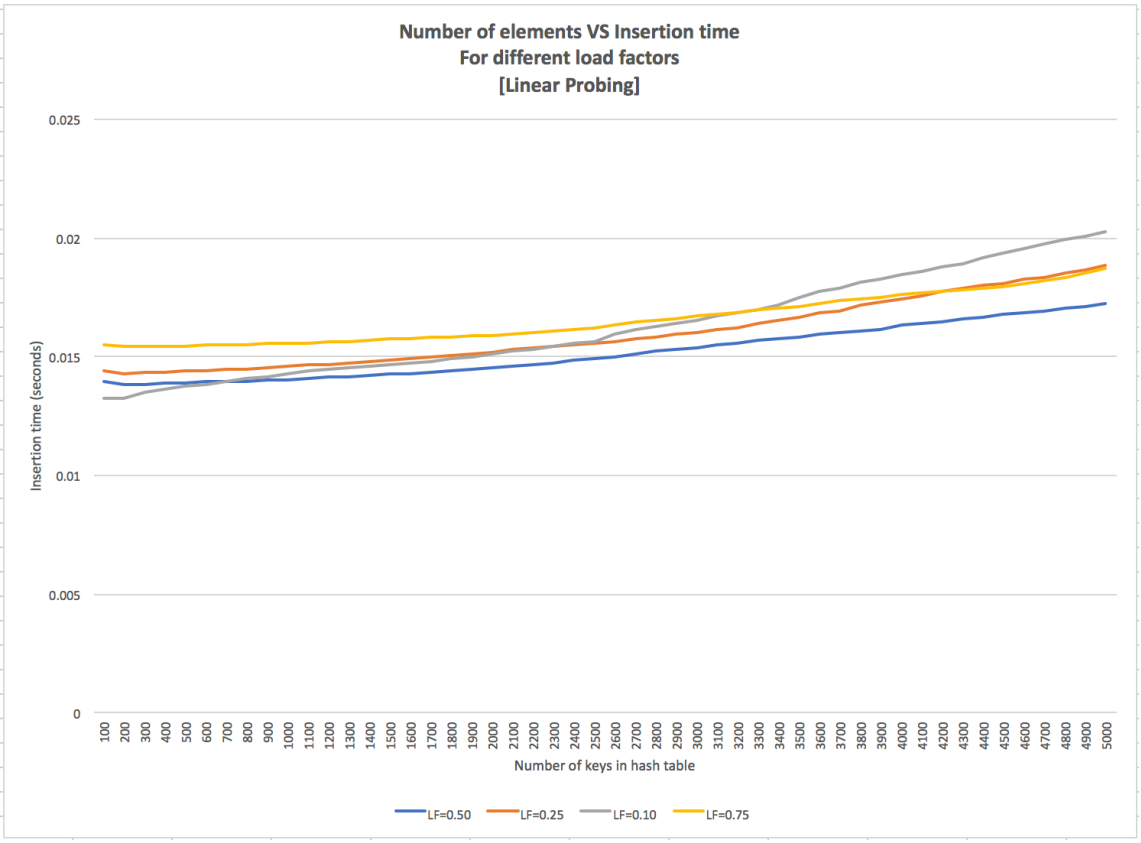
Graph: Load factor vs Deletion Time



Advantages and disadvantages of Double hashing:

The main advantage with double hashing is that it allows for smaller tables (higher load factors) than linear or quadratic probing, but at the expense of higher costs to compute the next probe. The higher cost of computing the next probe may be preferable to longer probe sequences, especially when testing two keys equal is expensive.

Varying LOAD FACTOR THRESHOLD (Linear probing):



As we increase load factor threshold (for rehashing and doubling the hash table) from 0.10 to 0.75, we notice that initially time taken for insertion is lower but as we increase number of elements in hash table (or hash table size), the time taken to insert elements increases.

Hash Functions taken:

Cryptographic hash functions:

- Functions from keys to 256-bit values, or larger, such that it is believed that it is not possible to find a collision using computers that exist today.
- Map down to table size, for example by modN. Slow but high-quality: If they are not as good as random
- numbers, they are breakable. Use pseudorandom number generators for faster schemes.

Test Cases:

The keys to be inserted have been taken from a random probability distribution using Java's Random generator function.

//Java Code

```
Random rand = new Random();
rand.setSeed(1);
int num = rand.nextInt(i) + 1;
```

```
static void LinearProbingTest() {
                Random rand = new Random();
                rand.setSeed(11);
                //Just enter 10000 elements and plot a graph
                LinearProbingHashing lp;
                for(int i=100;i<=5000;i+=100)
                {
                        lp = new LinearProbingHashing();
                        for(int j=1;j<=i;j++)
                        {
                        int num = rand.nextInt(i) + 1;
                        lp.insert(Integer.toString(num));
//                      lp.remove(Integer.toString(rand.nextInt(5000) + 1));
//                      System.out.println(i);
                        }
//                      System.out.println((double)i/1000);
//                      System.out.println(i);
                        double startTime=Time.getUserTime();
//                      lp.collisions=0;
                        lp.insert(Integer.toString(rand.nextInt(i) + 1));
//                      System.out.println(lp.collisions);
//                      lp.remove(Integer.toString(rand.nextInt(i) + 1));
//                      lp.search(Integer.toString(rand.nextInt(i) + 1));
                        double finishTime=Time.getCpuTime();
                        System.out.println((finishTime-startTime)/Math.pow(10.0, 9.0));
                }
```

References:

http://opendatastructures.org/ods-java.pdf

Lecture Notes : Michael Dillencourt (UCI)