Power Of Two Choice Hashing

Walid Safi - 100623815
Izien Iremiren - 100651421
Faisal Pindar - 100629476

Problem

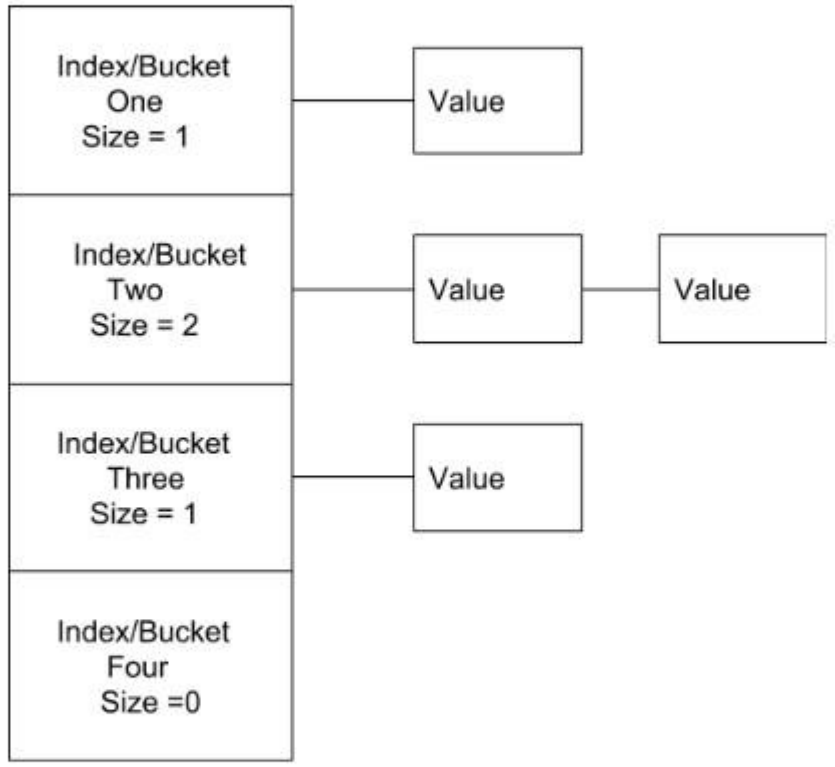
An interesting strategy for hashing with separate chaining is known as ***power of two-choices hashing***. Two independent hash functions are computed for each key, and a newly inserted element is placed into the choice of the two indicated buckets that currently has the fewest entries.

Hash Maps

- A Hash Map is a type of data structure that implements arrays and utilizes ADT (Abstract Data Type) that provides key-value relationships
- This allows us to quickly search, Delete and Insert elements into the hash table
- The hash map uses a hash function to convert the input, into an index of the hash map
- The hash map could have buckets that store more than one value in each index
- Example: A bucket can be a linked list to store the values with the same index (Separate Chaining)

Buckets

- Buckets are the located at each of the array
- Buckets in this case are linked lists of the Map entries
- We will be utilizing this in our hashing strategy


Index/Bucket One Size = 1
Value
Index/Bucket Two Size = 2
Value
Value
Index/Bucket Three Size = 1
Value
Index/Bucket Four Size =0

Power of two Choice Hashing

- The Power of to Choice Hashing is a strategy that uses two independent hash functions (The functions do not affect each other in anyway)
- Each key uses both the hash functions in generate two possible indexes to be inserted
- Example (Key = Toronto , H_1(Key) = 16 , H_2(Key) = 3)
- The algorithm checks which index of the table (Table[16], Table[3]) contains the least amount of entries, and inserts the new element into that bucket

Approach

- **The problem**: Our goal is to take in strings as the input, convert them to keys and the insert them into a hash table using the *power of two choices*.

- **Create a working algorithm**:
    - The strings are entered either as user-input, hard-coded or randomly-generated strings. Whatever the method, a hash object is created using the size declared by the user to created a *Linked List* representing the *Hash Table*. And for each index of the *Linked List*, a *Linked List* is created which represents *Collision Handling*.
    - The strings are then placed in an *Insert* function which converts them, using two independent hash functions, into two keys. The keys are then compared w.r.t the *Hash Table* to find out which “bucket” has fewer strings and is the placed in that “bucket”.

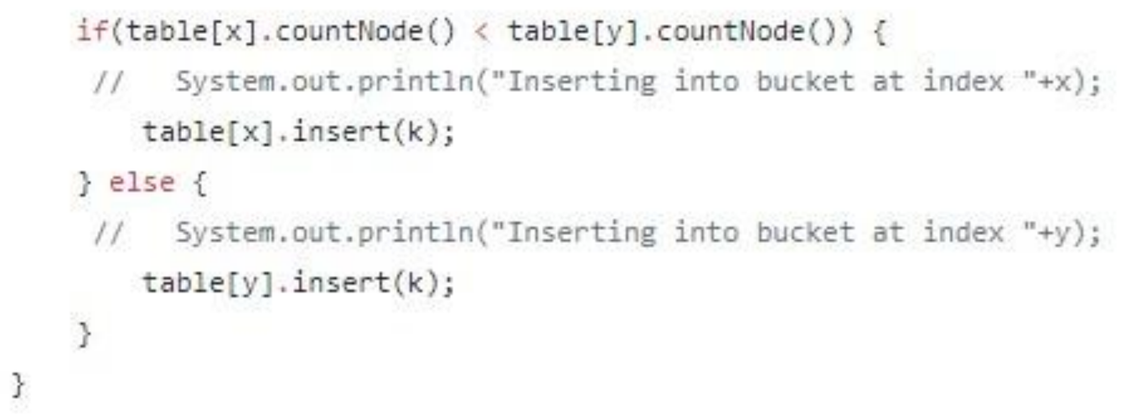
- **Data structure:** We made use of Linked Lists and Nodes.

Project goals

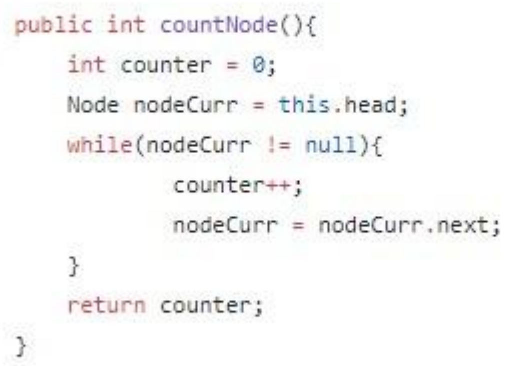
- Our goal was to develop an algorithm that is able to take any input and map the the input to a bucket in our array
- Our algorithm must be able to use two independent hash functions to map/index the input
- Our algorithm must be able to check the size of the buckets at the given index and insert the element in the bucket with the least amount of entries
- Our algorithm must be able to take user inputted key and search for that key in our hash map
- Test the program using various input sizes (Keys)
- n = 100 , 1000 , 10000 , 100000 , 1000000

Checking the Size of the Buckets

- Uses the countNode method in the LinkedList Class to determine the sizes of buckets at both indexes



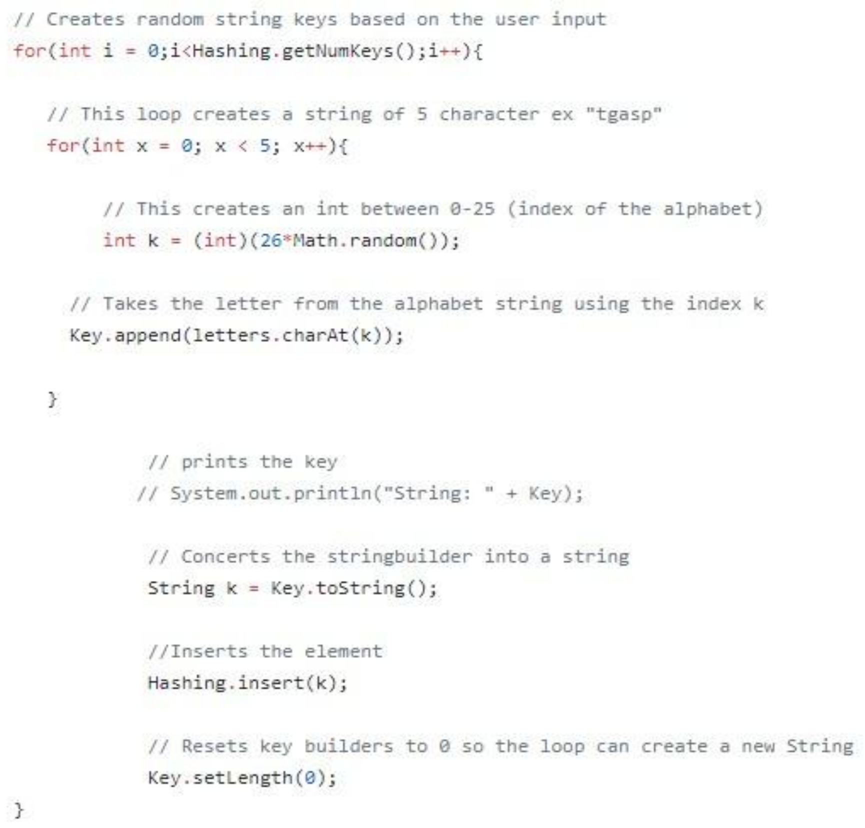
```
    if(table[x].countNode() < table[y].countNode()) {
     //   System.out.println("Inserting into bucket at index "+x);
        table[x].insert(k);
    } else {
     //   System.out.println("Inserting into bucket at index "+y);
        table[y].insert(k);
    }
}
```



```
public int countNode(){
    int counter = 0;
    Node nodeCurr = this.head;
    while(nodeCurr != null){
            counter++;
            nodeCurr = nodeCurr.next;
    }
    return counter;
}
```

Random Generated Keys

- The method to generate random strings (Keys) is calculated to be $O(n^2)$
- First Loop = n
- Second Loop = n+5
- $O(n^2) = n(n+5)$
- $n^2 + 5n$



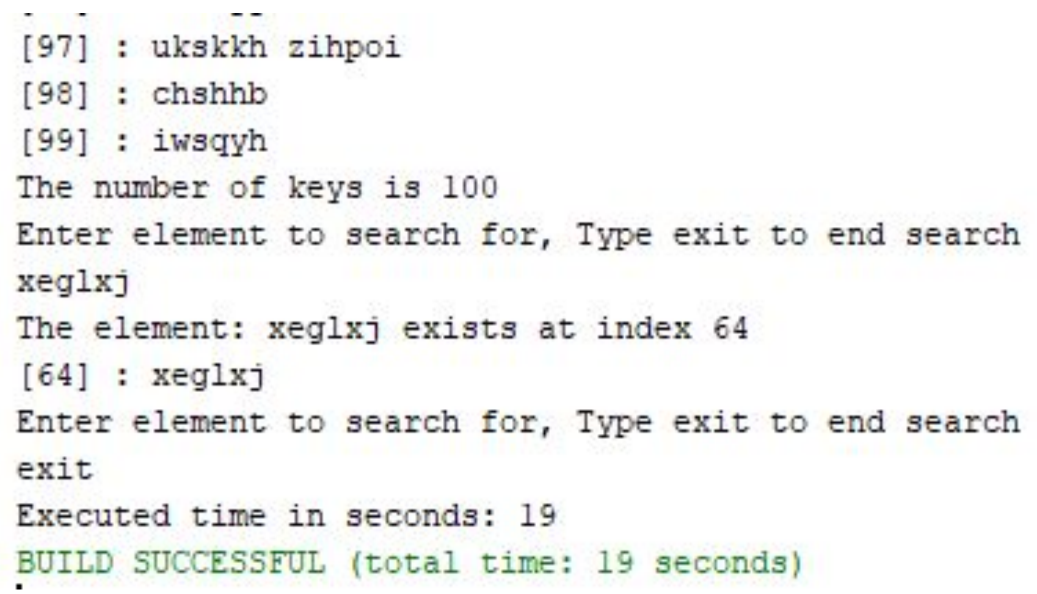
```
// Creates random string keys based on the user input
for(int i = 0;i<Hashing.getNumKeys();i++){

   // This loop creates a string of 5 character ex "tgasp"
   for(int x = 0; x < 5; x++){

        // This creates an int between 0-25 (index of the alphabet)
        int k = (int)(26*Math.random());

     // Takes the letter from the alphabet string using the index k
     Key.append(letters.charAt(k));

   }

            // prints the key
           // System.out.println("String: " + Key);

            // Concerts the stringbuilder into a string
            String k = Key.toString();

            //Inserts the element
            Hashing.insert(k);

            // Resets key builders to 0 so the loop can create a new String
            Key.setLength(0);
}
```

Time Complexity

- Insert and Search, each have a time complexity of O(1) in their best case, where the key to be executed on is first in the table. However, in cases where the key to be executed on is random the expected number of probes for an insertion with open addressing is 1 / (1 - *a*). Where a=n/N, and n=number of keys and N=table size.

Test Cases

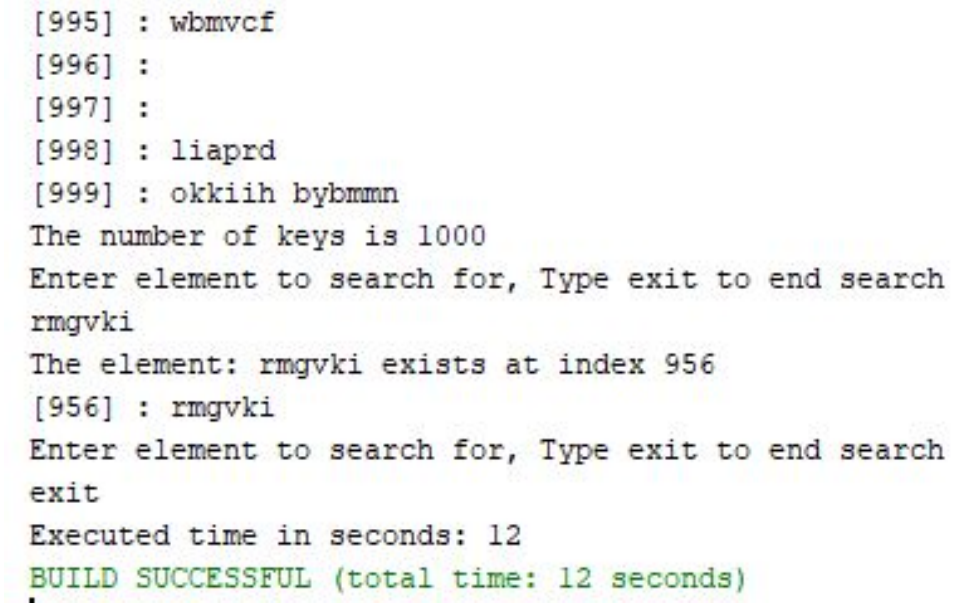
N = 100



```
[97] : ukskkh zihpoi
[98] : chshhb
[99] : iwsqyh
The number of keys is 100
Enter element to search for, Type exit to end search
xeglxj
The element: xeglxj exists at index 64
[64] : xeglxj
Enter element to search for, Type exit to end search
exit
Executed time in seconds: 19
BUILD SUCCESSFUL (total time: 19 seconds)
```

Test Cases

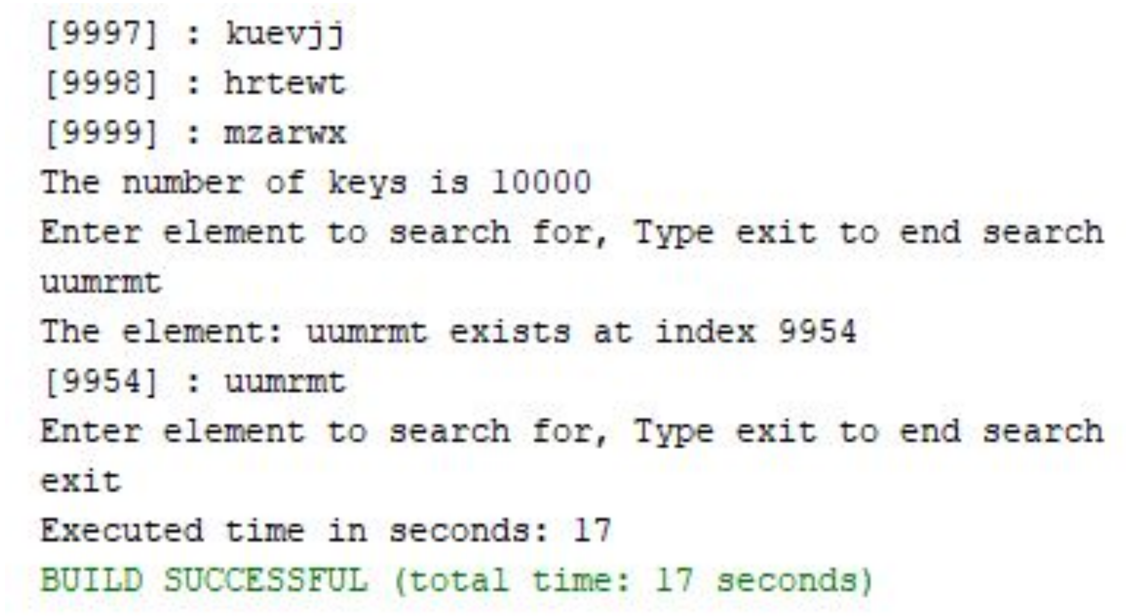
N = 1,000



```
[995] : wbmvcf
[996] :
[997] :
[998] : liaprd
[999] : okkiih bybmmn
The number of keys is 1000
Enter element to search for, Type exit to end search
rmgvki
The element: rmgvki exists at index 956
[956] : rmgvki
Enter element to search for, Type exit to end search
exit
Executed time in seconds: 12
BUILD SUCCESSFUL (total time: 12 seconds)
```

Test Cases

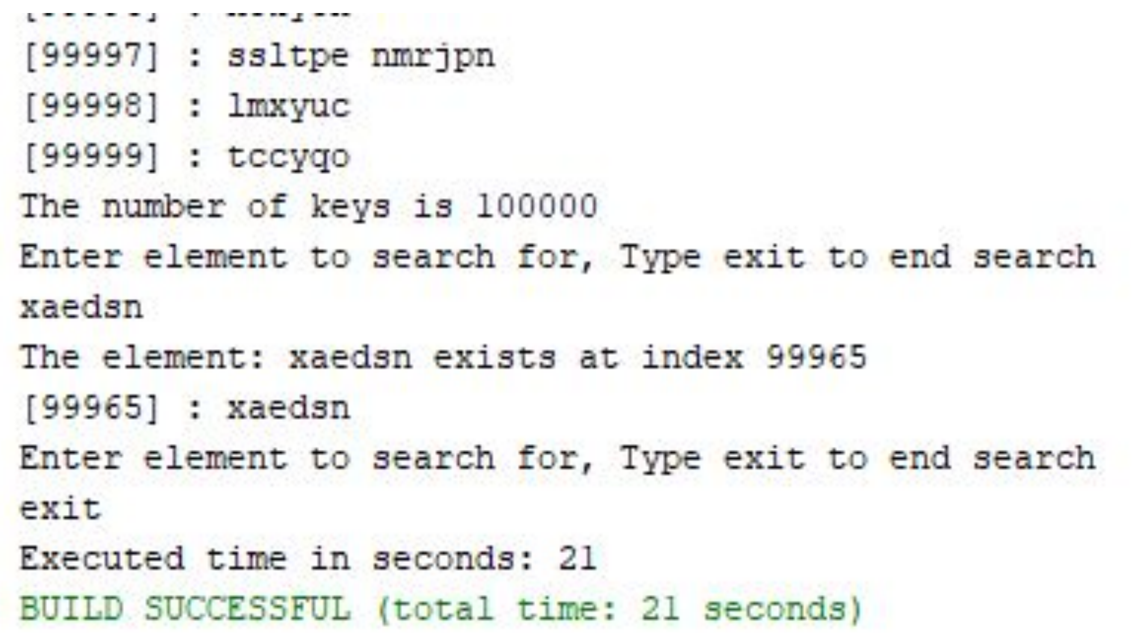
N = 10,000



```
[9997] : kuevjj
[9998] : hrtewt
[9999] : mzarwx
The number of keys is 10000
Enter element to search for, Type exit to end search
uumrmt
The element: uumrmt exists at index 9954
[9954] : uumrmt
Enter element to search for, Type exit to end search
exit
Executed time in seconds: 17
BUILD SUCCESSFUL (total time: 17 seconds)
```

Test Cases

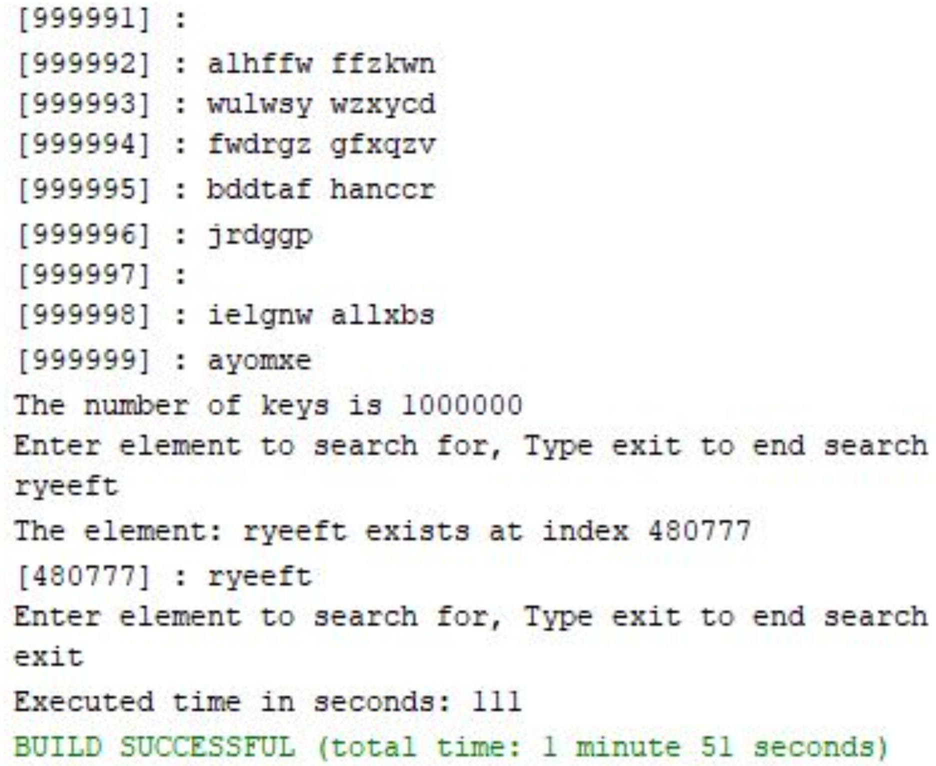
N = 100,000



```
[99997] : ssltpe nmrjpn
[99998] : lmxyuc
[99999] : tccyqo
The number of keys is 100000
Enter element to search for, Type exit to end search
xaedsn
The element: xaedsn exists at index 99965
[99965] : xaedsn
Enter element to search for, Type exit to end search
exit
Executed time in seconds: 21
BUILD SUCCESSFUL (total time: 21 seconds)
```

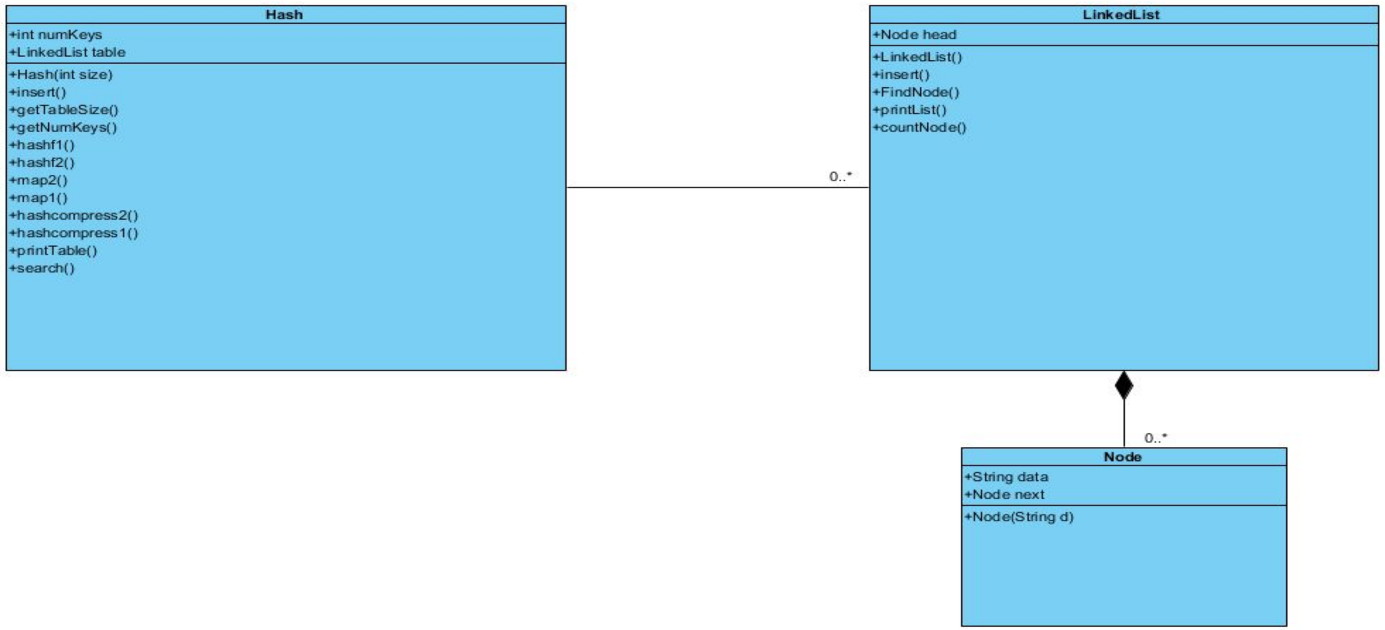
Test Cases

N = 1,000,000



```
[999991] :
[999992] : alhffw ffzkwn
[999993] : wulwsy wzxycd
[999994] : fwdrgz gfxqzv
[999995] : bddtaf hanccr
[999996] : jrdggp
[999997] :
[999998] : ielgnw allxbs
[999999] : ayomxe
The number of keys is 1000000
Enter element to search for, Type exit to end search
ryeeft
The element: ryeeft exists at index 480777
[480777] : ryeeft
Enter element to search for, Type exit to end search
exit
Executed time in seconds: 111
BUILD SUCCESSFUL (total time: 1 minute 51 seconds)
```

CLASS DIAGRAM



Contributions

Group Member	Task Assigned
Walid	Code design and implementation, Test cases and Presentation.
Faisal	Code design and implementation, Time Complexity and Presentation.
Izien	Code design and implementation,Class Diagram design and Presentation.