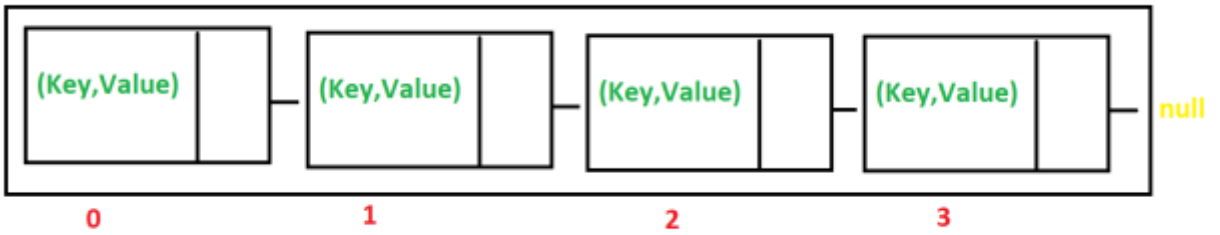
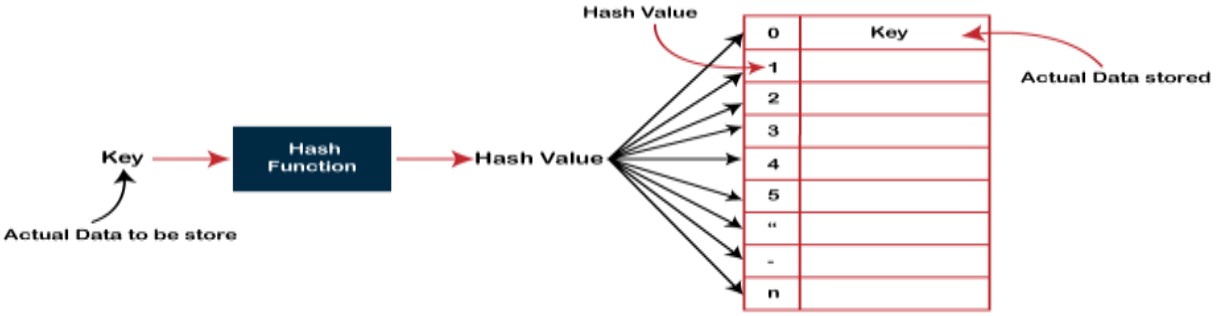
Lab#14 – Implementation of HashTable





Code:

```
import java.util.Scanner;

/*
       This file defines a HashTable class.  Keys and values in the hash table
       are of type Object.  Keys cannot be null.  The default constructor
       creates a table that initially has 64 locations, but a different
       initial size can be specified as a parameter to the constructor.
       The table increases in size if it becomes more than 3/4 full.
   */

public class HashTable {

    static private class ListNode {
         // Keys that have the same hash code are placed together
         // in a linked list.  This private nested class is used
         // internally to implement linked lists.  A ListNode
         // holds a (key,value) pair.
       Object key;
       Object value;
       ListNode next;  // Pointer to next node in the list;
                       // A null marks the end of the list.
    }

    private ListNode[] table;  // The hash table, represented as
```

```
                              // an array of linked lists.

   private int count;  // The number of (key,value) pairs in the
                       // hash table.

   public HashTable() {
        // Create a hash table with an initial size of 64.
      table = new ListNode[64];
   }

   public HashTable(int initialSize) {
        // Create a hash table with a specified initial size.
        // Precondition: initalSize > 0.
      table = new ListNode[initialSize];
   }

   void dump() {
         // This method is NOT part of the usual interface for
         // a hash table.  It is here only to be used for testing
         // purposes, and should be removed before the class is
         // released for general use.  This lists the (key,value)
         // pairs in each location of the table.
      System.out.println();
      for (int i = 0; i < table.length; i++) {
           // Print out the location number and the list of
           // key/value pairs in this location.
         System.out.print(i + ":");
         ListNode list = table[i]; // For traversing linked list number i.
         while (list != null) {
            System.out.print("  (" + list.key + "," + list.value + ")");
            list = list.next;
         }
         System.out.println();
      }
   } // end dump()

   public void put(Object key, Object value) {
         // Associate the specified value with the specified key.
         // Precondition:  The key is not null.
      int bucket = hash(key); // Which location should this key be in?
      ListNode list = table[bucket]; // For traversing the linked list
                                     // at the appropriate location.
      while (list != null) {
            // Search the nodes in the list, to see if the key already exists.
         if (list.key.equals(key))
```

```
            break;
         list = list.next;
      }
       // At this point, either list is null, or list.key.equals(key).
      if (list != null) {
           // Since list is not null, we have found the key.
           // Just change the associated value.
         list.value = value;
      }
      else {
           // Since list == null, the key is not already in the list.
           // Add a new node at the head of the list to contain the
           // new key and its associated value.
         if (count >= 0.75*table.length) {
              // The table is becoming too full.  Increase its size
              // before adding the new node.
            resize();
         }
         ListNode newNode = new ListNode();
         newNode.key = key;
         newNode.value = value;
         newNode.next = table[bucket];
         table[bucket] = newNode;
         count++;  // Count the newly added key.
      }
   }

   public Object get(Object key) {
         // Retrieve the value associated with the specified key
         // in the table, if there is any.  If not, the value
         // null will be returned.
      int bucket = hash(key);  // At what location should the key be?
      ListNode list = table[bucket];  // For traversing the list.
      while (list != null) {
            // Check if the specified key is in the node that
            // list points to.  If so, return the associated value.
         if (list.key.equals(key))
            return list.value;
         list = list.next;  // Move on to next node in the list.
      }
       // If we get to this point, then we have looked at every
       // node in the list without finding the key.  Return
       // the value null to indicate that the key is not in the table.
      return null;
   }
```

```
public void remove(Object key) {
      // Remove the key and its associated value from the table,
      // if the key occurs in the table.  If it does not occur,
      // then nothing is done.
   int bucket = hash(key);  // At what location should the key be?
   if (table[bucket] == null) {
        // There are no keys in that location, so key does not
        // occur in the table.  There is nothing to do, so return.
      return;
   }
   if (table[bucket].key.equals(key)) {
        // If the key is the first node on the list, then
        // table[bucket] must be changed to eliminate the
        // first node from the list.
      table[bucket] = table[bucket].next;
      count--; // Remove new number of items in the table.
      return;
   }
    // We have to remove a node from somewhere in the middle
    // of the list, or at the end.  Use a pointer to traverse
    // the list, looking for a node that contains the
    // specified key, and remove it if it is found.
   ListNode prev = table[bucket];  // The node that precedes
                                   // curr in the list.  Prev.next
                                   // is always equal to curr.
   ListNode curr = prev.next;  // For traversing the list,
                               // starting from the second node.
   while (curr != null && ! curr.key.equals(key)) {
      curr = curr.next;
      prev = curr;
   }
    // If we get to this point, then either curr is null,
    // or curr.key is equal to key.  In the later case,
    // we have to remove curr from the list.  Do this
    // by making prev.next point to the node after curr,
    // instead of to curr.  If curr is null, it means that
    // the key was not found in the table, so there is nothing
    // to do.
   if (curr != null) {
      prev.next = curr.next;
      count--;  // Record new number of items in the table.
   }
}
```

```
    public boolean containsKey(Object key) {
          // Test whether the specified key has an associated value
          // in the table.
       int bucket = hash(key);  // In what location should key be?
       ListNode list = table[bucket];  // For traversing the list.
       while (list != null) {
             // If we find the key in this node, return true.
          if (list.key.equals(key))
             return true;
          list = list.next;
       }
        // If we get to this point, we know that the key does
        // not exist in the table.
       return false;
    }

    public int size() {
          // Return the number of key/value pairs in the table.
       return count;
    }

    private int hash(Object key) {
          // Compute a hash code for the key; key cannot be null.
          // The hash code depends on the size of the table as
          // well as on the value returned by key.hashCode().
       return (Math.abs(key.hashCode())) % table.length;
    }

    private void resize() {
          // Double the size of the table, and redistribute the
          // key/value pairs to their proper locations in the
          // new table.
       ListNode[] newtable = new ListNode[table.length*2];
       for (int i = 0; i < table.length; i++) {
             // Move all the nodes in linked list number i
             // into the new table.  No new ListNodes are
             // created.  The existing ListNode for each
             // key/value pair is moved to the newtable.
             // This is done by changing the "next" pointer
             // in the node and by making a pointer in the
             // new table point to the node.
          ListNode list = table[i]; // For traversing linked list number i.
          while (list != null) {
                // Move the node pointed to by list to the new table.
             ListNode next = list.next;  // The is the next node in the list.
```

```
                                        // Remember it, before changing
                                        // the value of list!
            int hash = (Math.abs(list.key.hashCode())) % newtable.length;
                 // hash is the hash code of list.key that is
                 // appropriate for the new table size.  The
                 // next two lines add the node pointed to by list
                 // onto the head of the linked list in the new table
                 // at position number hash.
            list.next = newtable[hash];
            newtable[hash] = list;
            list = next;  // Move on to the next node in the OLD table.
         }
      }
      table = newtable;  // Replace the table with the new table.
   } // end resize()

} // end class HashTable



//A Program for Testing HashTable:

/*
   A little program to test the HashTable class.  Note that I
   start with a really small table so that I can easily test
   the resize() method.
*/

class TestHashTable {

   public static void main(String[] args){
       Scanner textIO=new Scanner(System.in);
      HashTable table = new HashTable(2);
      String key,value;
      while (true) {
         System.out.println("\nMenu:");
         System.out.println("   1. test put(key,value)");
         System.out.println("   2. test get(key)");
         System.out.println("   3. test containsKey(key)");
         System.out.println("   4. test remove(key)");
         System.out.println("   5. show complete contents of hash table.");
         System.out.println("   6. EXIT");
         System.out.print("Enter your command:  ");
         switch (textIO.nextInt()) {
            case 1:
```

```
               System.out.print("\n   Key = ");
               key = textIO.next();
               System.out.print("");
               System.out.print("   Value = ");
               value = textIO.next();
               table.put(key,value);
               System.out.print("");
               break;
            case 2:
               System.out.print("\n   Key = ");
               key = textIO.next();
               System.out.println("   Value is " + table.get(key));
               break;
            case 3:
               System.out.print("\n   Key = ");
               key = textIO.next();
               System.out.println("   containsKey(" + key + ") is "
                                        + table.containsKey(key));
               break;
            case 4:
               System.out.print("\n   Key = ");
               key = textIO.next();
               table.remove(key);
               break;
            case 5:
               table.dump();
               break;
            case 6:
               return;  // End program by returning from main()
            default:
               System.out.println("   Illegal command.");
               break;
         }
         System.out.println("\nHash table size is " + table.size());
      }
   }

} // end class TestHashTable
```

Task#01

Remember in lab#03 (Implementation of array) I have given you a task, implement same task using hashTable

Task#01: The contact app on our phone contains a lot of contacts. In **ContactApp(class)** perform the following operations:

Display all contact
Search a contact by its name//name -> number
Add a new contact // name, number , pos/index
Update the contact //name1, name2
Delete any contact //