In this exercise we study Hash tables for dictionaries.

Excercise 2a. (Implement and test a dictionary ADT implemented using separate chaining hash table, 1p)

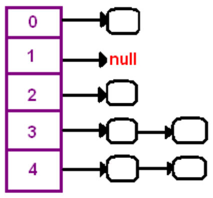
Implement flexible class MyDictionary whose operations are

```
public MyDictionary(int n)
Object get(String key);
int    put(Object item, String key);
int    del(string key);
void   printDictionary();
```

Constructor creates the dictionary with at least *n* buckets. `get()` tries to find the element with key, returns null if not found, reference to the object if found. `put()` inserts the element with key into the dictionary replacing older element (if it exists) in the dictionary. `del()` removes the element with key from the dictionary. printDictionary() prints the content of the dictionary like

```
0: [empty]
1: [empty]
2: ysEbLaehqtiVb0bRgtyTBjjoLtjyb2oa,WczDTVmnvGiMCwrPf1N8BLSshBqkUoWK,oEqGLKk7DqMAM9ckQ9paFTciCsf1LtxE
3: Cn4VzvdwuaAcO5zxhmCqLeO01c5AaqtE
```

Implementation of ADT dictionary should be based on separate chained hash table. In this hash table collisions are resolved by creating a simple linked list if more than one element should be placed on the same bucket. The picture below clarifies the consept. Hash table contains only pointers (so that it is relatively small in size). There is NULL on the hash table if the bucket (one entry in the hash table) is empty, and pointer to the node containing an element info and pointer to (possible) next node on the list. List is terminated by a node whose next pointer is NULL.



In order the implementation to be so flexible as possible, all the data structures should be dynamic. Even the hash table should be dynamic.

One candidate for the hash function is the so called “K & R hash function” given below

```
    /* K & R hash function for strings */
    private static final int MULTIPLIER = 31;
    private int hash(String str) {
        int h;

        h  = 0;
        for (int i = 0; i < str.length(); i++)
            h  = MULTIPLIER * h + str.charAt(i);

        return h & Integer.MAX_VALUE;
    }
```

You give the string as an argument to the function, and you’ll get integer hash value out. This must be converted to the hash table size by modulo operator (%).

Note that in order the hash operation to work in the optimal way, the size of the hash table should be prime number. With the following function, you can find out what is the next largest prime number given the argument n.

```
/* find next prime number */
private static int nextPrime(int n) {
        int prime = 0, i, nextPrime;

        /* check first if this is a prime number */
        for(i=2; i<n/2; i++) {
                if(n%i == 0) {
                        prime = 1;
                        break;
                }
        }

        if(prime == 1) {
                /* no, try to find next one */
                nextPrime = n;
                prime = 1;

                while(prime != 0) {
                        nextPrime++;
                        prime = 0;

                        for(i=2; i<nextPrime/2; i++) {
                                if(nextPrime%i == 0) {
                                        prime = 1;
                                        break;
                                }
                        }

                }

                return (nextPrime);
        } else
                /* yes, return this as is */
                return (n);
}
```

Test your implementation eith the following test application.

```
public class Lab02 {

    private final static int STRLEN = 32;
    private final static int N      = 30;
    static final String AB = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
    static SecureRandom rnd = new SecureRandom();

    private static String randomString(int len) {
        StringBuilder sb = new StringBuilder(len);
        for (int i = 0; i < len; i++)
            sb.append(AB.charAt(rnd.nextInt(AB.length())));
        return sb.toString();
    }

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        MyDictionary dict = new MyDictionary(N/2);

       String[] tableStr;
        String   search_element;
       int      j = 0;
       Object   empty, result;
       int      r;
```

```
        /* first create sorted collection of strings */
        tableStr = new String[N];
        for (int i = 0; i < tableStr.length; i++)
            tableStr[i] = randomString(STRLEN);

        /* then store it to the dictionary */
        empty = new Object();
        for (int i = 0; i < tableStr.length; i++)
            dict.put(empty, tableStr[i]);
        System.out.println("Content of the dictionary:");
        dict.printDictionary();

        /* try to search half of the strings from the dictionary */
        System.out.println("\n\nHalf of the searches should succeed, half fail:");
        for (int i = 0; i < N/2; i++) {
            j = rnd.nextInt(N);
            search_element = tableStr[j];

            if (i > N/4) search_element += "#"; // quater of the strings should not be found
            result = dict.get(search_element);

            System.out.format("%1$2d:   element   %2$s   (%3$02d),   search   result   %4$s\n",   i,
search_element, j, result!=null ? "F" : "N");
        }

        /* then delete first and the middle element from the dictionary */
        dict.del(tableStr[0]); dict.del(tableStr[N/2-1]);

        /* try to search again strings from the dictionary, first and the last should not be found
*/
        System.out.println("\n\nFirst and last search should fail, other should succeed:");
        for (int i = 0; i < N/2; i++) {
            j = i;
            search_element = tableStr[i];
            result = dict.get(search_element);

            System.out.format("%1$2d:   element   %2$s   (%3$02d),   search   result   %4$s\n",   i,
search_element, j, result!=null ? "F" : "N");
        }
    }
}
```

Excercise 2b. (Optional, 0,25p bonus. Implement and test a dictionary ADT implemented using open addressing hash table)

Implement the previous exercise system, but now with open addressing scheme to resolve collisions. Use the same test system to the test the operation of your implementation.

Excercise 2c. (Optional, 1,00p bonus. Implement and test a LZW-compression system)

Read input from the file and then write LZW-compressed results of the input to the output file. Notice that the output file should be opened to the binary mode for writing. Implement also the decompression operation. Read the decompressed file and then write uncompressed version the file back to the output.
Warning: this is a difficult exercise