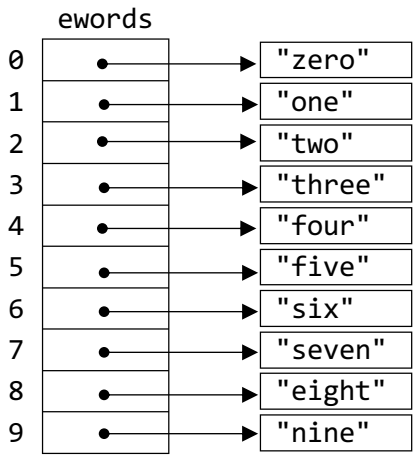
アルゴリズムとデータ構造 2018　第 11 回　　年　　組　　番 氏名

1. 配列は**ランダムアクセス**が可能なデータ構造であり，整数の添字（インデックス）で要素を参照することができる。この特徴を用いると，整数からデータへの変換表を作ることができる。以下は，配列を用いて 1 桁の整数を英単語に“翻訳”するプログラムである。空欄 1 ヶ所を埋めて動作を確認せよ。

```
import java.util.Scanner;

public class Program {
   final static String[] ewords = {
      "zero", "one", "two",  "three", "four",
      "five", "six", "seven", "eight", "nine" };

   public static void main(String[] args) {
      Scanner sc = new java.util.Scanner(System.in);
      System.out.print("number? ");
      int n = sc.nextInt();
      if (0 <= n && n <= 9)
         System.out.println(ewords[    ]);
   }
}
```



2. **ハッシュ関数**は**キー**と呼ばれるデータを入力とし，それからなるべくバラバラな整数値（**ハッシュ値**）を算出する関数である。以下は文字列のハッシュ関数の例であり，入力された文字列に対してなるべく重複しない整数を生成する。このハッシュ関数によって，次の文字列はどのような数値に変換されたか。

(a) zero _____　(b) one _____　(c) two _____　(d) three _____　(e) four _____

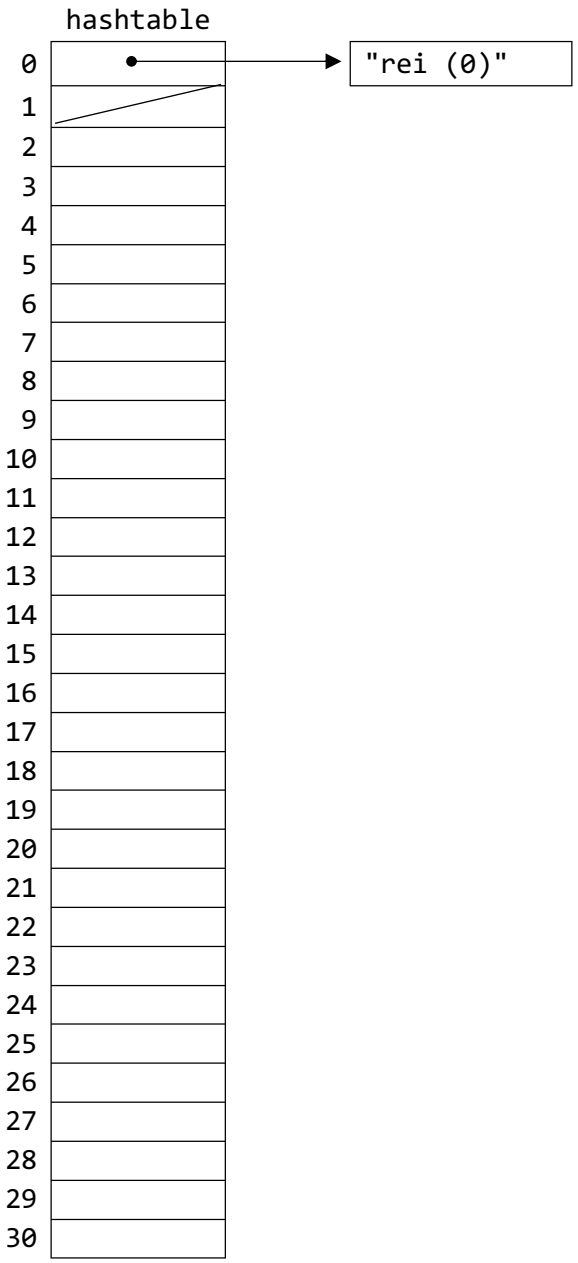
(f) five _____　(g) six _____　(h) seven _____　(i) eight _____　(j) nine _____

```
import java.util.Scanner;

public class Program {
   final static int HASHSIZE = 31;   // ハッシュ値の上限（最大値+1）（素数がよい）

   // ハッシュ関数の例：文字列を 0〜30 の整数（ハッシュ値）に変換する
   public static int hash(String key) {

      final int X = 37;              // 計算結果を散らばらせるには，（別の）素数がよい
      int L = key.length();
      int h = 0;
      for (int i = 0; i < L; i++) {
         int c = key.charAt(i);      // i 番目の文字の文字コード（整数）を求める
         h = h * X + c;              // 直前の結果に X を掛けながら，全文字分足し合わせる
      }                              // ★
      return Math.abs(h % HASHSIZE); // Java では%の結果は負になる可能性があるので絶対値
   }

   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      for (int i = 0; i < 10; i++) {
         System.out.print("string? ");
         String str = sc.next();
         int h = hash(str);          // ハッシュ関数によって文字列 str を整数 h に変換する
         System.out.println("code: " + h);
      }
   }
}
```

3. 文字列 key の構成文字（の文字のコード）を $c_0\, c_1\, c_2\, c_3\, c_4$ …（例えば，"one"の場合は c_0='o', c_1='n', c_2='e'）とおくとき，2.の★印のところでの変数 h の値を，変数 X, L, c_0,c_1,c_2,…を用いた計算式で表してみよ。

4. **ハッシュテーブル**は，データをそのハッシュ値が示す配列の要素（**バケット**）に格納することによって，探索を高速化する仕組みである。下記は，この原理を理解するための簡単な英和辞書のプログラムである。空欄に 2.で求めた英単語のハッシュ値を埋めて動作を確認し，ハッシュテーブルの構造を図示せよ。

```
import java.util.Scanner;

public class Program {
   // 2.と同じ数にする
   final static int HASHSIZE = 31;

   // ハッシュテーブル（要素は文字列）
   static String[] hashtable;

   // ハッシュ関数
   public static int hash(String key) {
      // 2.と全く同じ内容
   }

  public static void main(String[] args) {
     hashtable = new String[HASHSIZE];

     // 各バケットに項目のデータを登録する
     hashtable[     ] = "rei (0)";
     hashtable[     ] = "ichi (1)";
     hashtable[     ] = "ni (2)";
     hashtable[     ] = "san (3)";
     hashtable[     ] = "shi (4)";
     hashtable[     ] = "go (5)";
     hashtable[     ] = "roku (6)";
     hashtable[     ] = "shichi (7)";
     hashtable[     ] = "hachi (8)";
     hashtable[     ] = "kyuu (9)";

     Scanner sc = new Scanner(System.in);
     System.out.print("English? ");
     String eng = sc.next();

     // ハッシュ値の算出と登録データの探索
     int h = hash(eng);
     String entry = hashtable[h];

     if (entry == null)
        System.out.println("Not found.");
     else
        System.out.println(entry + " ?");
  }
}
```



5. 4. の単純なハッシュテーブルを用いた探索ついて，登録データ数 n に対する計算量を考察せよ。ただし，データが文字列の場合は，その長さのばらつきによる影響は無視するものとする。

6. 4.の方法では，ハッシュ値が互いに等しい（**衝突**する）データは複数登録することができない。そこで，**チェーン法**では，各要素を連結リストにすることによって複数のデータの登録を可能にする。下記は単語の出現回数を数えるプログラムである。適切に空欄を埋めて実行させるとともに，構造を図示してみよ。

```
/* Node.java */
public class Node {
   public String word; // 単語（キー）
   public int count;  //出現回数
   Node next;  // 次のノード

   Node(String word) {
      this.word = word;
      this.count = 0;
      this.next = null;
   }
}

/* HashTable.java */
public class HashTable {
   private final int HASHSIZE;

   // 各バケットには連結リストの head に
   // 相当する値が入る（初期値は null）
   private Node[] hashtable;

   // コンストラクタ
   public HashTable(int size) {
      HASHSIZE = size;
      hashtable = new Node[size];
   }

   // ハッシュ関数
   public int hash(String key) {
      // 2.と全く同じ内容
   }

   // 全要素を表示する
   public void printAll() {
      // すべてのバケットをループ
      for (int i = 0; i < HASHSIZE; i++) {
         System.out.printf("%2d ", i);
         // すべてのノードをたどる
         for (Node n = hashtable[i];
               n != null; n = n.next) {
            System.out.printf(
               "%s(%d) ", n.word, n.count);
         }
         System.out.println();
      }
   }

   // 単語を検索し，なければ登録する
   public Node lookUp(String word) {
      // まずハッシュ関数でハッシュ値を算出
      int h =

      // ハッシュテーブルを探索し要素を取得
      Node n =

      // 連結リストを順にたどって単語を探索
      while (n != null) {
         if (word.equals(n.word))
            return

         n =
      }

      // 未登録ならば連結リストの先頭に挿入
      //  （第 9 回の push 操作と同様の処理）
      n = new Node(word);
      n.next =

      hashtable[    ] = n;
      return n;
   }
}

/* Program.java */
import java.util.Scanner;

public class Program {
   public static void main(String[] args) {
      HashTable ht = new HashTable(59);

      Scanner sc = new Scanner(System.in);
      while (true) {
         String word = sc.next();
         if (word.equals("QUIT")) break;

         // 要素に対してカウントアップ処理
         Node n = ht.lookUp(word);
         n.count++;
      }
      ht.printAll();
   }
}
```

7. 6.のプログラムについて，「59」の部分はそのままでハッシュ関数を下記のものなどに置き換えて表示を比較し，なるべく高速・効率的な探索のためにはどのようなハッシュ関数がよいか考察せよ。

```
public int hash(String key) { return key.length() % HASHSIZE; }
public int hash(String key) { return key.charAt(0) % 13; }
public int hash(String key) { return key.charAt(0) % HASHSIZE; }
```