Numbers

1. Define function `read` that has two parameters:

   - a name of a input text file and
   - a fixed-size array of **double**s (the length of an array is a constant value .

   A file consists of integers. Each line of the file has exactly `COLUMN_NUMBER` of integers separated with white spaces.

   An example of a file with `COLUMN_NUMBER` == 5:

   ```
   2          -5         12         2          8
    4          2         -6           6         32
   -1           3         4          2           4
   ```

   Number of rows (lines) is not known. We can assume that if the file exists it is in a correct format.

   The functions reads data from a file and writes into an array (passed as a parameter) averages of columns from the file. If the file does not exist or is empty then the array is filled with zeros.

2. Define a function `sort` that has one parameter

   - a fixed-size array (length: `COLUMN_NUMBER`) of **double**s

   and sorts it in non descending order.

3. Define function `merge` that has three parameters:

   - a fixed-size array of `COLUMN_NUMBER` **double**s,
   - a fixed-size array of `COLUMN_NUMBER` **double**s,
   - a fixed-size array of $2 * $ `COLUMN_NUMBER` **double**s.

   The first two array hold numbers sorted in a non descending order. The third array is an output array for merged numbers from the first and second array ie. the series of ordered all numbers from the first and second array. You may not use a sort function to merge the numbers.

   An example of merging of series of numbers:

`first`	2	3	7	8	12					
`second`	0	1	5	8	15					
`third`	0	1	2	3	5	7	8	8	12	15

4. Define function `write` that has two parameters:

   - a name of a binary output file and
   - an array of $2 * $ `COLUMN_NUMBER` **double**s

   and writes numbers from the array into the binary file.

5. Define function `operation` that has three parameters:

   - a name of the first text input file,
   - a name of the second text input file and
   - a name of a binary outpu file.

   The function read averages of numbers from the first and second file, merges them and writes the merged series in an output file. Of course the function uses the functions defines above.

Pictures

A text file consists of integer numbers representing pixels of a picture. The numbers are separated by white spaces. The first two number have a special meaning: the former is a number of rows (not greater than `W`), the latter–number of columns (not greater than `K`). The other numbers are values of pixels in the picture.

An example of a picture with 7 rows and 5 columns:

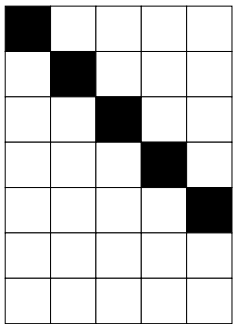
```
 7    5
50   33   16    0    0
33   33   22   11    0
16   22   33   22   16
 0   11   22   33   33
 0    0   11   22   33
 0    0    0   11   16
 0    0    0    0    0
```

1. Defina a function `read` that read the dimensions and pixels of a picture from a named file.
2. Define a function `write` that writes a picture to a named files in the format described above.
3. Define a function Proszę napisać funkcję `transform` that transforms a picture.
   (a) The function has two parameters: a name of an input file and a name of an outpu file.
   (b) An input file holds an input picture in the format defined above.
   (c) A transformed picture is written into an output file in the same format.
   (d) A pixel in the w-th row and k-th column of an output picture is an integer value of an average of a pixel in the w-th row and k-th column and its neighbours in an input file.
   (e) A neighbour of a pixel p is any pixel that has a common edge or vertex with p.
   (f) The input picture is not modified in transformation.
   (g) The function uses functions defined in pp. 1 and 2.

Example

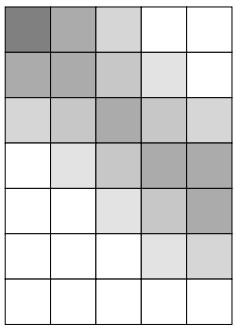
input file

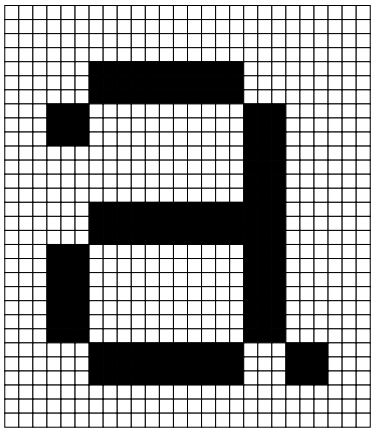
```
  7     5
100     0     0     0     0
  0   100     0     0     0
  0     0   100     0     0
  0     0     0   100     0
  0     0     0     0   100
  0     0     0     0     0
  0     0     0     0     0
```

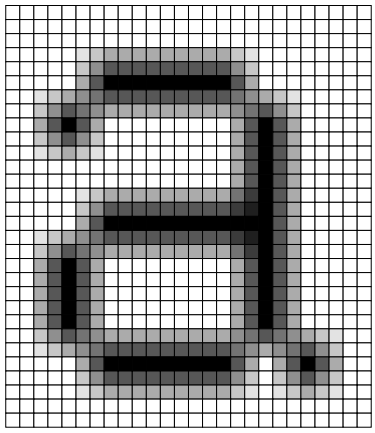


output file:

```
  7     5
 50    33    16     0     0
 33    33    22    11     0
 16    22    33    22    16
  0    11    22    33    33
  0     0    11    22    33
  0     0     0    11    16
  0     0     0     0     0
```



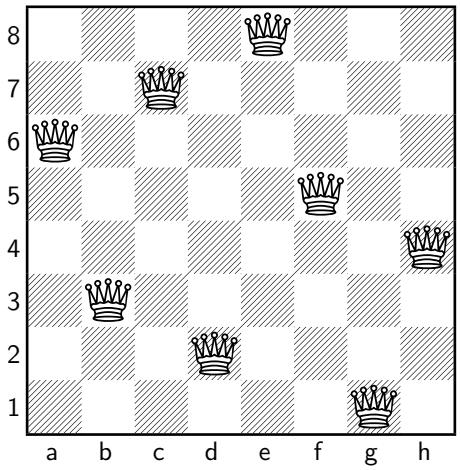




Scientific fact: The operation described above is a *discrete convolutional low-pass filter* ☺

Eight queens problem

The eight queens problem is a problem of placing eight chess queen on a chessboard so that no two queens threaten each other. The chessboard is a standard 8×8 board.



Define a function that **verifies** placing of queens on an 8×8 chessboard. The placing of queens is saved in an input text files as a matrix of characters. There are 8 rows and each row has 8 characters. The legal characters are: '`.`' (empty square) i '`H`' (queen).

The function has one parameter: a name of an input file and returns **true** when the placing is correct and **false** when the placing is incorrect. The input file may be absent, but if it exists then it is correct. Example of an input file:

```
....H...
..H.....
H.......
.....H..
.......H
.H......
...H....
......H.
```

Attention! The function **verifies** a solution to the eight queen problem. It **does not solve** the eight queen problem.

Knight's tour problem

The knight's tour problem is a problem of finding a sequence of moves of a chess knight on an 8×8 chessboard such that the knight visits all chessboard squares once and only once.

Define a function that **verifies** a sequence of moves of a knight on an 8×8 chessboard saved in a text file. The chessboard is saved as a matrix of 8 rows and 8 columns of integers. The numbers are separated by blacks. The numbers represent consecutive moves of the knight. The knight start on the square labelled 1, then moves to square 2, ..., and ends on square 64.

The function has one parameter: a name of an input text file and returns **true** when the tour is correct or **false** when the tour is incorrect. The input file may be absent, but if it exists then it is correct. An example of an input file:

```
50 63  34 13  48 11  22 19
35 14  49 62  23 20  47 10
64 51  16 33  12 45  18 21
15 36  61 52  17 24   9 46
38  1  32 25  60 53  44  7
29 26  37  4  41  8  59 56
 2 39  28 31  54 57   6 43
27 30   3 40   5 42  55 58
```

Attention! The function **verifies** a solution to the knight's tour problem. It **does not solve** the knight's tour problem.

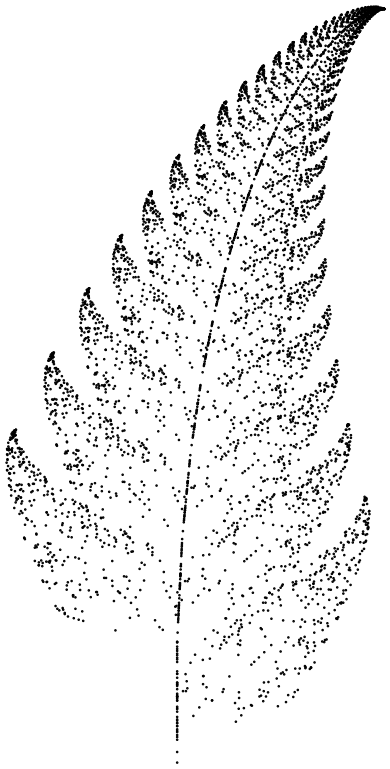


Figure 1: The Barnsley fern (5000 points).

Barnsley fern

The Barnsley fern is a fractal resembling a fern leaf. It can be drawn in a quite simple way. The first point is always $(0, 0)$. The coordinates of the n-th point are generated from coordinates of the $(n-1)$-th point with four functions:

$$\begin{cases} x_n = 0.85x_{n-1} + 0.04y_{n-1} \\ y_n = -0.04x_{n-1} + 0.85y_{n-1} + 1.6 \end{cases} \tag{1}$$

$$\begin{cases} x_n = -0.15x_{n-1} + 0.28y_{n-1} \\ y_n = 0.26x_{n-1} + 0.24y_{n-1} + 0.44 \end{cases} \tag{2}$$

$$\begin{cases} x_n = 0.20x_{n-1} - 0.26y_{n-1} \\ y_n = 0.23x_{n-1} + 0.22y_{n-1} + 1.6 \end{cases} \tag{3}$$

$$\begin{cases} x_n = 0 \\ y_n = 0.16y_{n-1} \end{cases} \tag{4}$$

The probabilities of application of functions are:

- function (1) with probability $p_1 = 0.85$,
- function (2) with probability $p_2 = 0.07$,
- function (3) with probability $p_3 = 0.07$,

- function (4) with probability $p_4 = 0.01$.

Fig. 1 presents the Barnsly fern generated with 5000 points. When you use more points the picture is less fuzzy.

Define a function `Barnsley` that calculates points and saves them into a text file. Each line of the output file holds one point as a pair: abscissa and ordinate separated with a space. The function has two parameters: a number of points and a name of an output file.

Shopping

Let's define two structures:

```
struct Item
{
    string name;
    double price;
    Item * pPrev, *pNext; // pointer to previous and next Item in a double-linked list
    // some other data
};

struct Customer
{
    string name, surname;
    Item * pBoughtItems;          // head of list of items
    Customer * pLeft, * pRight; // children of tree node
    // some other data
};
```

The structures are used to build a complicated data structure – binary tree of doubly linked lists – Fig. 2. Customers are gathers in a binary tree ordered by surnames. Items of one customer build a doubly linked list in ascending order of items' prices. A customer may have any length of items, no items as well.

1. Define a function `add` that adds an item (price and name given) to a customer (name and surname given). If a customer is absent, a new customer object is created in the correct localisation. An empty list of items is also possible.

2. Define a function `remove` that removes an item from a customer. Parameters of the function: root of a tree, name and surname of a customer, name and price of an items. The function returns:

   - **true** if item removed successfully,
   - **false** otherwise.

   An empty tree is possible.

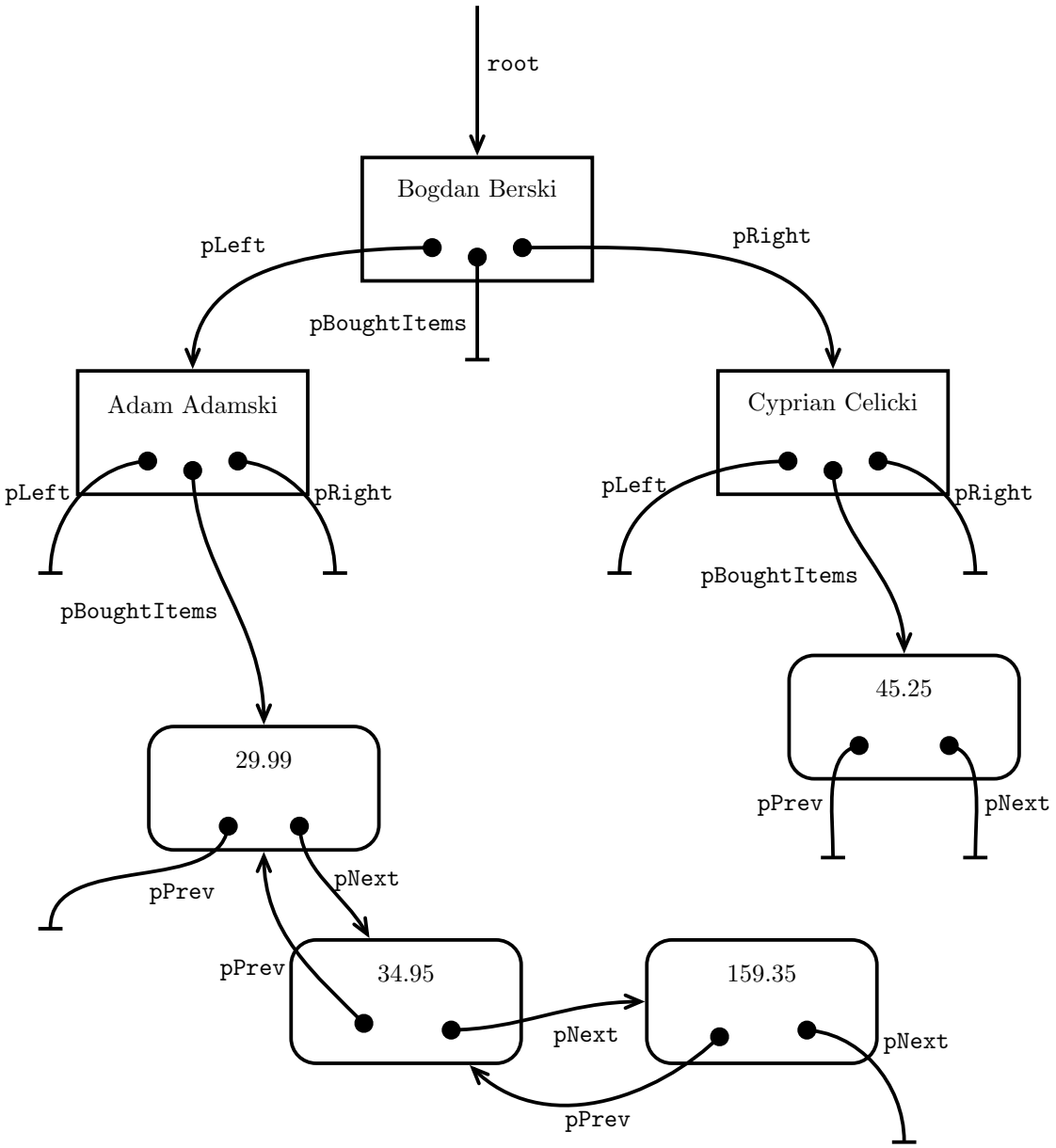


Figure 2: Example of a structure with customers and items.

Cruise

Let's define two structures:

```
struct Passenger
{
    string name, surname;
    Passenger * pNext;
    // some other data
};

struct Cruise
{
    string start_harbour, end_harbour;
    string date;
    Passenger * pPassengers;
    Cruise * pPrev, * pNext;
    // some other data
};
```

The structures are used to build a complicated data structure presented in Fig. 3. Cruises build a doubly link list in non descending order by dates. Passengers in a cruise are gathered in a singly linked list in non descending order by their surnames. A cruise may have any length of passengers (also no passengers).

Define functions:

1. `Cruise * findCruise (Cruise * pH, const string & start_harbour, const string & end_harbour, const string & date);`

   The function searches for a cruise defined with a date, start harbour, and end harbour in a list of cruises `pH`. The function returns an address of the found cruise. If there is not matching cruise, the function returns `NULL` (or **nullptr** or 0).

2. `Cruise * addCruise (Cruise * & pH, Cruise * & pTail, const string & date, const string & start_harbour, const string & end_harbour);`

   The function adds in a non descending order a new cruise to a list of cruises (head: `pH`, tail `pT`). The adding of a new cruise keeps the non descending order by date. The function returns an address of an added cruise. It is possible that the list is empty. The function add a new cruise **iteratively**.

3. `Passenger * addPassengerToCruise (Passenger * pH, const string & name, const string & surname);`

   The function adds a new passenger to a singly linked list of cruises (head `pH`). The function returns a head of a modified list. The function keeps the non descending order by surnames of passengers. An empty list is possible. The function adds passenger **recursively**.

4. `void addPassenger (Cruise * & pH, Cruise * & pT, const string & date, const string & start_harbour, const string & end_harbour, const string & name, const string & surname)`

   The function add a passenger (name, surname given) to a doubly linked list of cruises (with head `pH` and tail `pT`). If a cruise is missing, it is added so that the ordering of the list is kept. It is possible that the list of cruised is empty. Cruises with no passengers are also possible. Function uses the functions defined above.

5. `Cruise * favourite (Cruise * & pH)`

   The function returns an address of a cruise with the maximal length of passengers. Passed parameters: an address of the head of the list `pH`. If the list of cruises is empty, the function returns

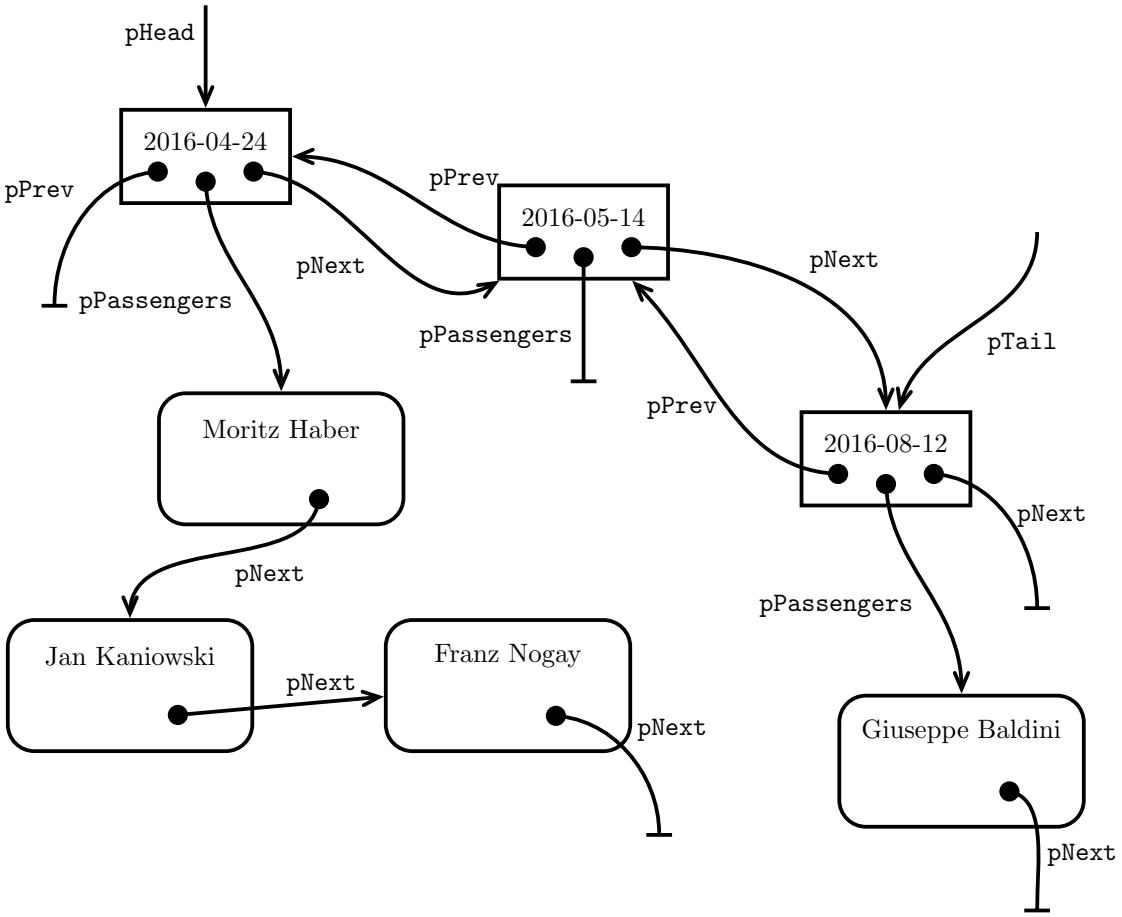


Figure 3: Example of cruises and passengers.

`NULL` (or **nullptr** or 0). If several cruises have the same lengths of passengers, the function returns an address of any of cruises with maximal length of passengers.

Attention: You may not modify the definitions of `Cruise` and **Passenger** structures.

Bacteria

Let's define new structures:

```
struct Bacterium
{
    string name;
    Bacterium * pNext;
    Genome * pGenome;
};

struct Genome
{
    double gene;
    Genome * pNext;
};
```

The structures are used to build a complicated data structure presented in Fig. 4. Bacteria are gathered in a singly lined list. Each bacterium has it own genome (a singly linked circular list). A genome is never empty, it has at least one gene.

Define functions:

1. **void** `add` (**Genome** *& `pGenome`, **Genome** * `pSequence`);

   The function adds a sequence (`pSequence`) to a bacterial genome. The sequence is added at the beginning of a genome, ie after addition the head of a circular list points at the newly added sequence. An example is presented in Fig. 5.

2. **Genome** * `cut` (**Genome** *& `pGenome`, **int** `length`);

   The function cuts a sequence of `length` genes from a genome `pGenome`. The function returns an address of a cut sequence. After cutting a genome is still circular. An example is presented in Fig. 5.

3. **void** `findExtremes` (**Bacterium** * `pHead`, **Bacterium** * & `pWorst`, **Bacterium** * & `pBest`);

   The function searches for the best and the worst bacterium. It uses the `evaluate` function. Its declaration is **double** `evaluate` (**Genome** * `pGenome`); Do not define the `evaluate` function!

4. **void** `crossGenomes` (**Bacterium** * `pStaphylococcus`, **Bacterium** * `pStreptococcus`);

   The function crosses genomes of two bacteria. First from genomes of both bacteria two sequences are cut. The lengths of the sequences are $\lfloor \frac{n}{2} \rfloor$, where n is a number of genes in a genome of a bacterium. Then the sequences are exchanged between bacteria. The lengths of the sequences are not necessary the same. The function call functions defined in pp. 1 and 2.

   Hint: The function **int** `length` (**Genome** * `pGenome`); is defined in the program. It returns a number of genes in a genome. Do not define this function!

   Hint: The *floor* function $\lfloor x \rfloor$ denotes the greatest integer not greater than x, ie. $\lfloor x \rfloor = \max\{k \in \mathbb{Z} : k \leqslant x\}$, eg. $\lfloor \pi \rfloor = 3$.

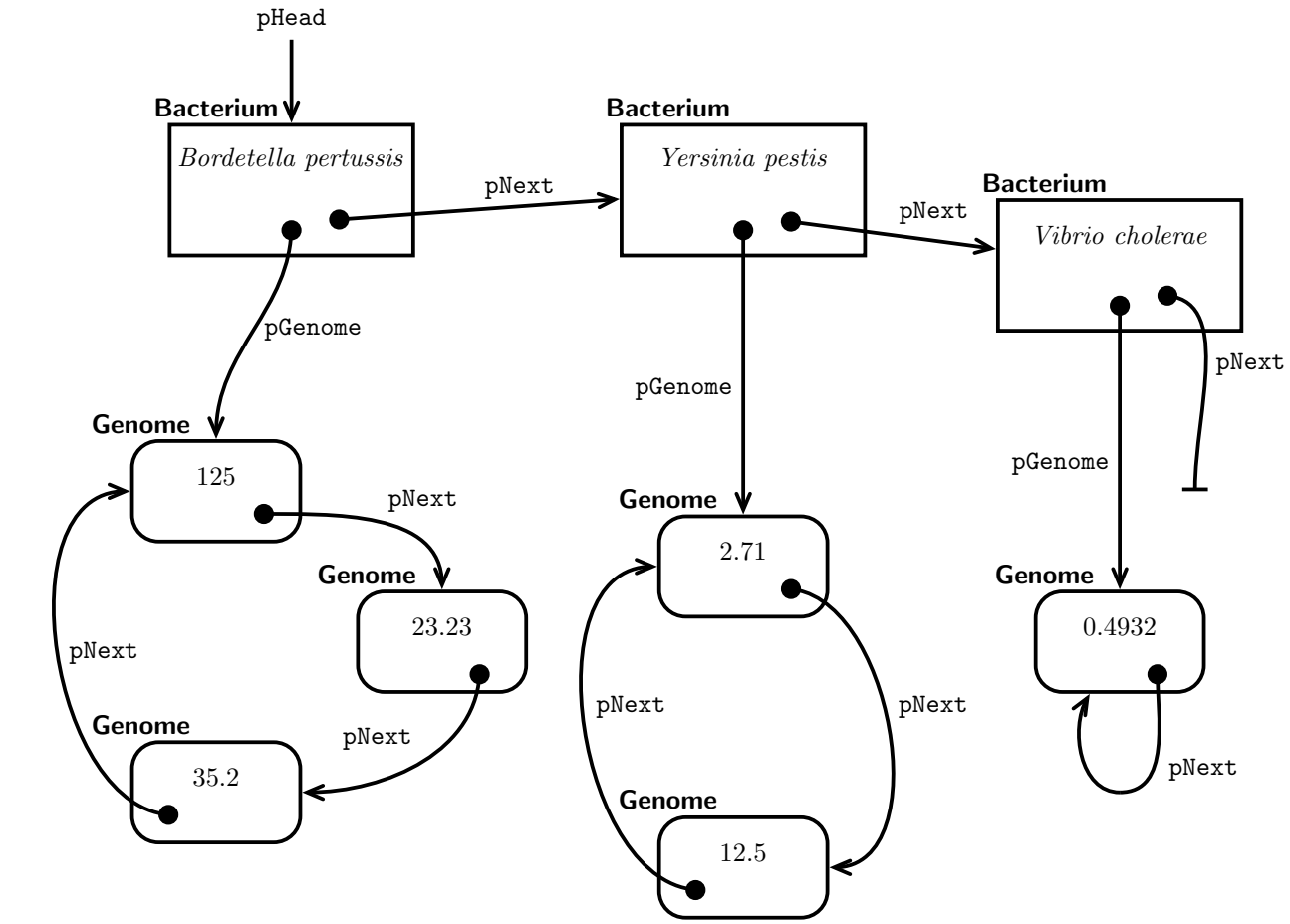


Figure 4: Example of bacteria and their genomes.

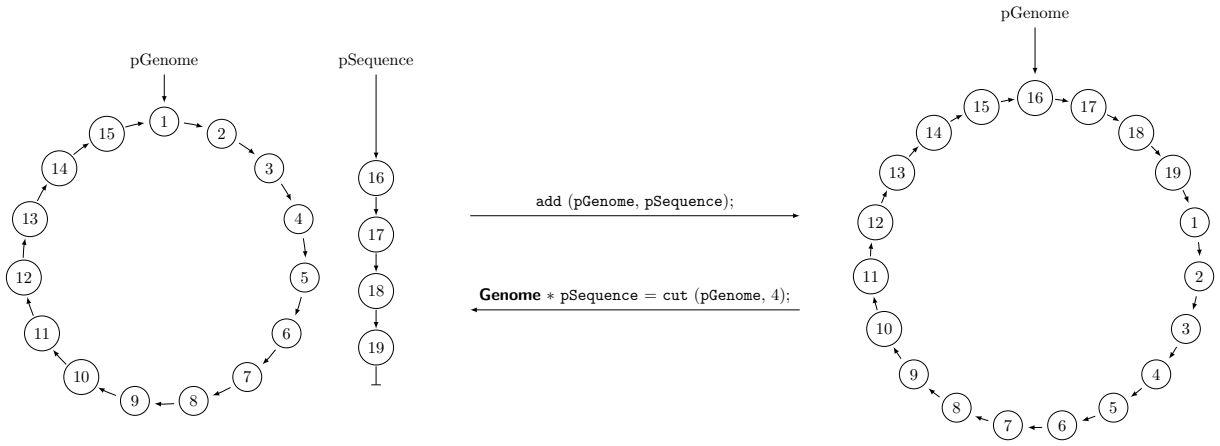


Figure 5: Adding and cutting of a sequence.

Election

```
struct Party
{
   string name;
   int nVotes;     // number of votes
   Party * pNext;
};

struct Candidate
{
   string surname;
   int nVotes;
   Party * pParty;
   Candidate * pNext;
};

struct Constituency
{
   int number;
   Candidate * pCandidates;
   Constituency * pLeft, * pRight;
};
```

The structures are used to build a complicated data structure presented in Fig. 6. Constituencies build a binary search tree ordered by constituency numbers. Each constituency has a singly linked list of candidates. Each candidate has a pointer (`pParty`) to a party they represent. All parties build a singly linked list.

We assume that each candidate has a unique surname. Also party names and constituency numbers are unique.

One function has been defined in the programme:

- **Party** * `findParty` (**Party** * & `pHead`, **const std**::**string** & `name`);

  The function returns a party in a list of parties `pHead`. If the party is missing, the function adds it and returns its address.

Attention: Do not modify definitions of structures **Party**, **Candidate**, and **Constituency**.

Define functions:

1. ```
   Constituency * findConstituency (Constituency * & pRoot, int number);
   ```

   The function returns an address of a constituency searched by its `number` in a tree `pRoot`. If a constituency is missing, the function adds a new constituency in correct location in the tree and returns an address of a newly added item.

2. ```
   void addCandidateToConstituency (Constituency * & pConstituencies, Party
        * & pParties, int constituency_number, const std::string &
      candidate_surname, const std::string & party_name, int
      number_of_votes);
   ```

   The function add a candidate (named `candidate_surname`) from `party_name` party to a list in a constituency numbered `constituency_number`. The function finds a constituency in a tree and a party in a list with functions defined above. A candidate in placed in any location in a list of candidates.

3.
```
int count_votes (Constituency * pConstituencies);
```

The function counts number of votes for each party from votes gained by party candidates. The function sets the field `nVotes` for each party. The function returns a sum of all votes of all candidates of all parties.

4.
```
void remove_threshold(Constituency * pConstituencies, int nVotes, double
    threshold);
```

The function removes candidates which parties have less votes than `threshold`. The threshold is a number from interval $[0, 1]$. The parameter `nVotes` denotes a total number of votes gained by all candidates. Let p be a threshold, n_i number of votes gained by the i-th party, and N number of all parties. The fraction of votes gained by the i-th party is calculated with formula

$$p_i = \frac{n_i}{\sum_{k=1}^{N} n_k}.$$

If $p_i < p$ then candidates of the i-th party are removed from the list of candidates.

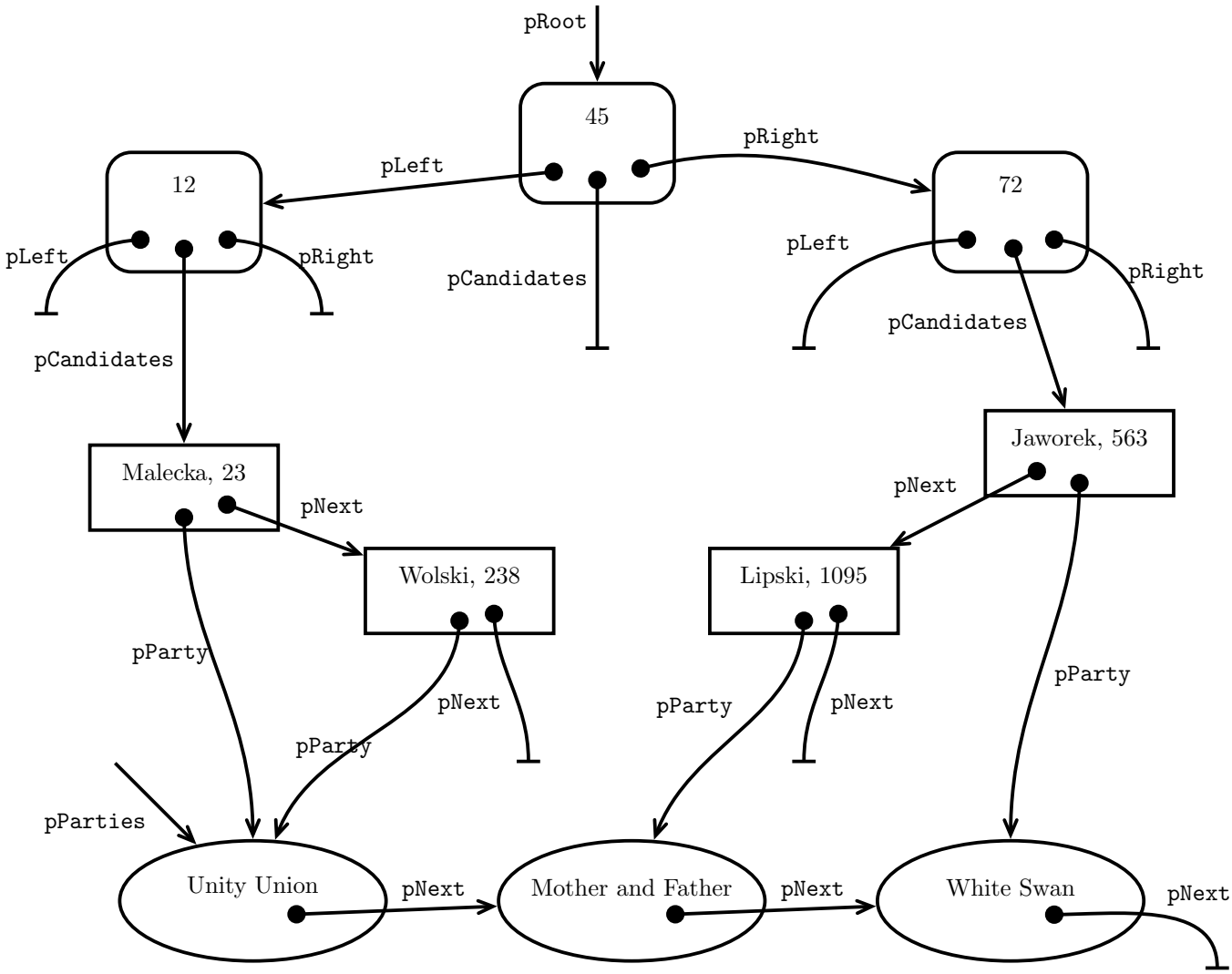


Figure 6: Examples of candidates, constituencies, and parties.