## 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.

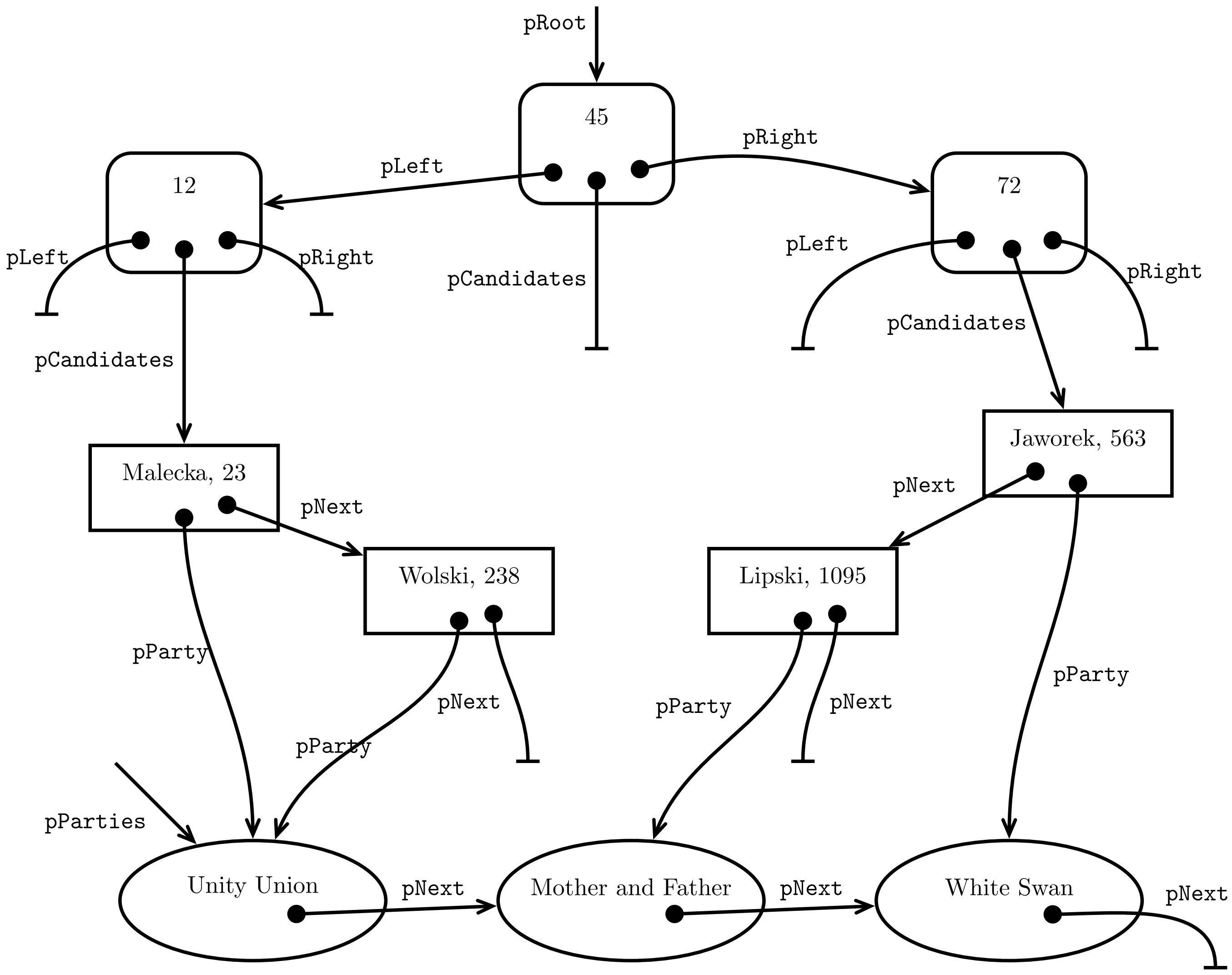

Figure 6: Examples of candidates, constituencies, and parties.