# Exercise Sheet #8

*Deadline: 05.06.23, 12:00*

**Problem 1** (*Hash Table*) (10 points)

*This problem is a lot of text, but it is not hard to do – I promise!* ☺
In many applications, the problem arises to store a pair of data, which we will from now on denote *key* and *value*. Consider for example a phone book software, where we want to associate a name (key) with a phone number (value). For this purpose, an often encountered and very useful data structure are *hash tables*[1]. The idea of behind hash tables is, to input the key into a function, called *hash function*, that calculates an index for the key and then stores the associated value at the respective index in an array underlying the hash table. The great thing about hash tables from a programmers point of view is that the lookup (i.e. finding the value for a given key or in our example the phone number for a given name) is very fast, since the index, at which the corresponding value is stored, can quickly be calculated from the key.[2] Note, that it is required that the keys are unique (on the other hand, the values don't have to be unique).
The goal of this problem is to implement a very simple (and mostly educational) hash table in `C++` that stores `int` values associated with `int` keys.

(a) We first build the bare-bones structure of our hash table and implement the functionality in the following parts. Write a class `HashTable` with two private member variables `int t_size` and `int* t_values` that are the size of the hash table, i.e. the number of values one can store, and a pointer to the array, where the values will be stored, respectively. Add a constructor, that takes an `int size` as an argument and initializes `t_size` with `size` and `t_values` with an integer array of size `size`. Furthermore, implement a destructor that deletes the `t_values` array.

(b) Next, we implement a hash function. In the calculation, that the hash function performs to produce an index for a given key, it can happen that two different keys produce the same index. This is referred to as a *hash collision* and the quality of a good hash function is to have as

[1] Sometimes they are referred to by *hash maps* or (e.g. in Python) *dictionaries*.

[2] On the flip side, imagine you would store pairs of names and phone numbers in an ordinary array: Finding the phone number of John Doe would require going through (potentially the whole) array until we encounter John and his phone number!

little collisions as possible. Usually, if a collision happens, two different values (with their associated keys) are stored at the same index. The terminology here is that we have *buckets* at every index. Then, at lookup, we still have to go through the bucket to find the correct key, but the number of items will be substantially smaller than the size of the hash table. Since we do not care about quality here, we just use a very simple hash function and only store a single value at a given index. If a collision does happen, we just output a warning to the user (we will implement that later). For your implementation we will use the following hash function:

```
int hash(int key) const {
    return key % m_size;
}
```

Implement that function as a private method in the `HashTable` class (yes, you just have to copy-paste that function into your program).

(c) Next, implement a function
`void insert(int key, int value){...}`
that inserts `value` into the hash table. Use `key` and the hash function to determine the index in `t_values`, where `value` must be stored. If you detect a hash collision, simply output a warning to the user and do absolutely nothing.
**Hint:** To easily detect a hash collision, it helps to make sure that `t_values` is initialized with zeros.

(d) Further, implement a function
`int get(int key)const {...}`
that retrieves the value associated to `key` and returns it.

(e) Finally, test your implementation with a meaningful example. Try to produce a hash collision.

**Problem 2** (*Perceptron Training*) (10 points)

The perceptron is a basic algorithm capable of performing linear classification. It takes the product of an input vector with a fixed vector of weights and adds a bias to create a single number, the activation, which is used to predict the label of the input. In short, a perceptron is a single-layer neural network.
Implement a training function and teach a perceptron model to tell the difference between sonar readings of rocks and mines.

(a) Create a prediction function that receives an input vector $\boldsymbol{x}$ and a weights vector $\boldsymbol{w}$ and calculates the activation

$$y' = \boldsymbol{w} \cdot \boldsymbol{x} + b.$$

The bias $b$ can be stored in the weights vector. Return the activation after applying a step function

$$\Theta(y) = \begin{cases} 1, & \text{if } y \geq 0, \\ 0, & \text{else.} \end{cases}$$

(b) Create a training function that receives a dataset and trains the perceptron to predict its labels. Start with all weights set to zero and calculate the prediction error of each data point, $l = (y - y')^2$. The true label $y$ should be the last element of each data vector. Use this error to update the weights after every data point with the update rules

$$\boldsymbol{w} \to \boldsymbol{w} + \eta l \boldsymbol{x},$$
$$b \to b + \eta l,$$

with the learning rate $\eta = 0.01$. Run through the whole dataset and print out the average prediction error.

(c) Test the power of your algorithm by learning on real data. Download the dataset `sonar.all-data` from this here (click on 'Data Folder'). Load the dataset and change all $R$ labels to 0 and $M$ labels to 1. Notice that the data is ordered, shuffle the order randomly to prevent overfitting. Call the training function and train your model for 30 steps (epochs) over the entire dataset. Can you see an improvement in classification error?

**Hint:** You don't have to solve this problem in `C++`. It might be easier to do in Python.