AARHUS UNIVERSITET
INGENIØRHØJSKOLEN

# DOA

# Hand-in 2
# Weeks 3-4: Elementary data structures and their processing

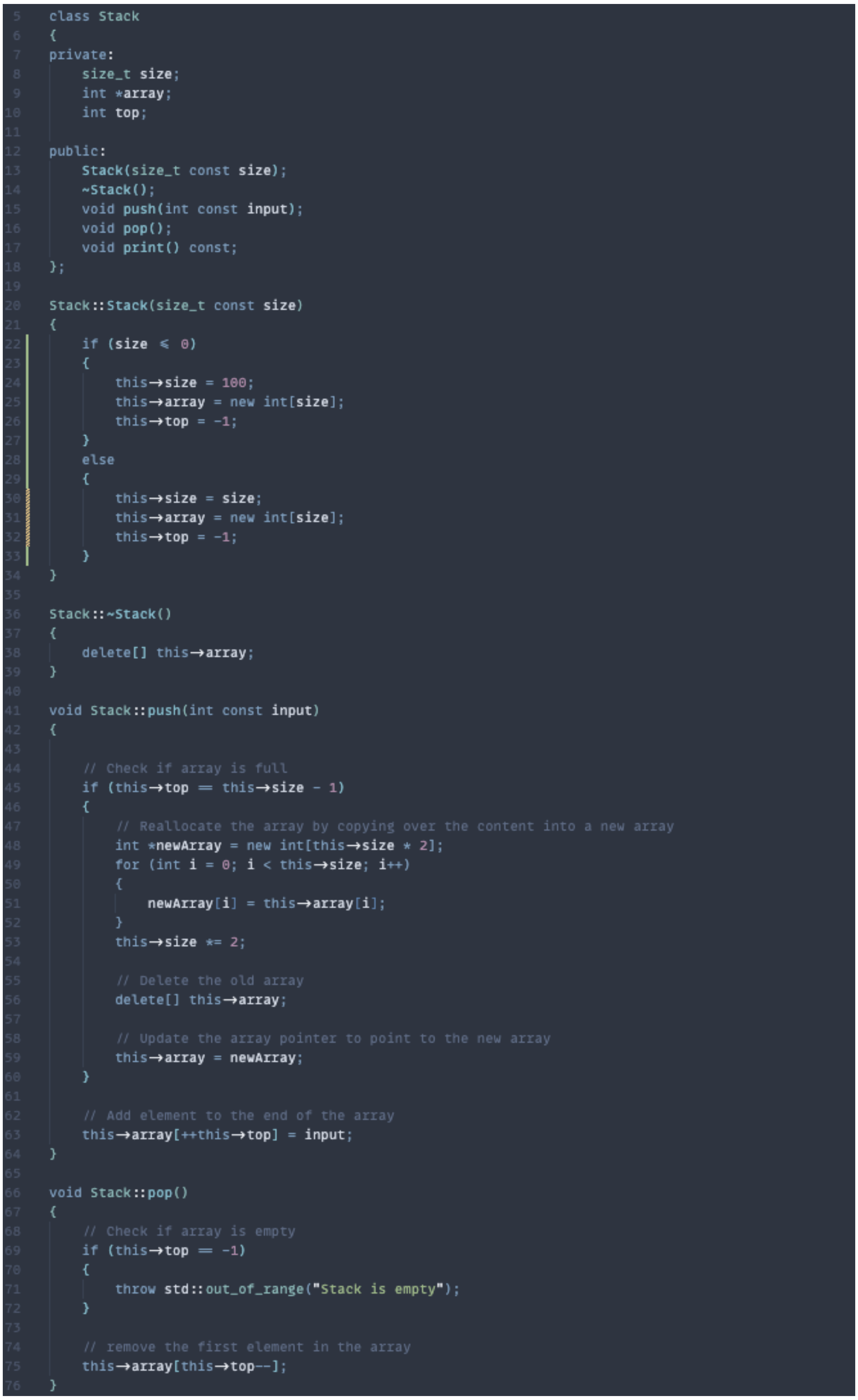
Group members:

| Studienummer | Navn | Studieretning |
|---|---|---|
| 201708558 | Mads Søborg | SW |
| 202007814 | Martin Dietz Vad | SW |

## Exercise 1:

The pop operation will always be $O(1)$ constant time because we are simply removing values from the array.

The pop operation will most of the time be $O(1)$ constant time because we are adding values to the array. However, when the size of the array is full, we are required to reallocate the entirety of the array which turns it into $O(N)$ linear time. If we preallocate a decent size this won't happen often, and we the operation can therefor still be considered constant.

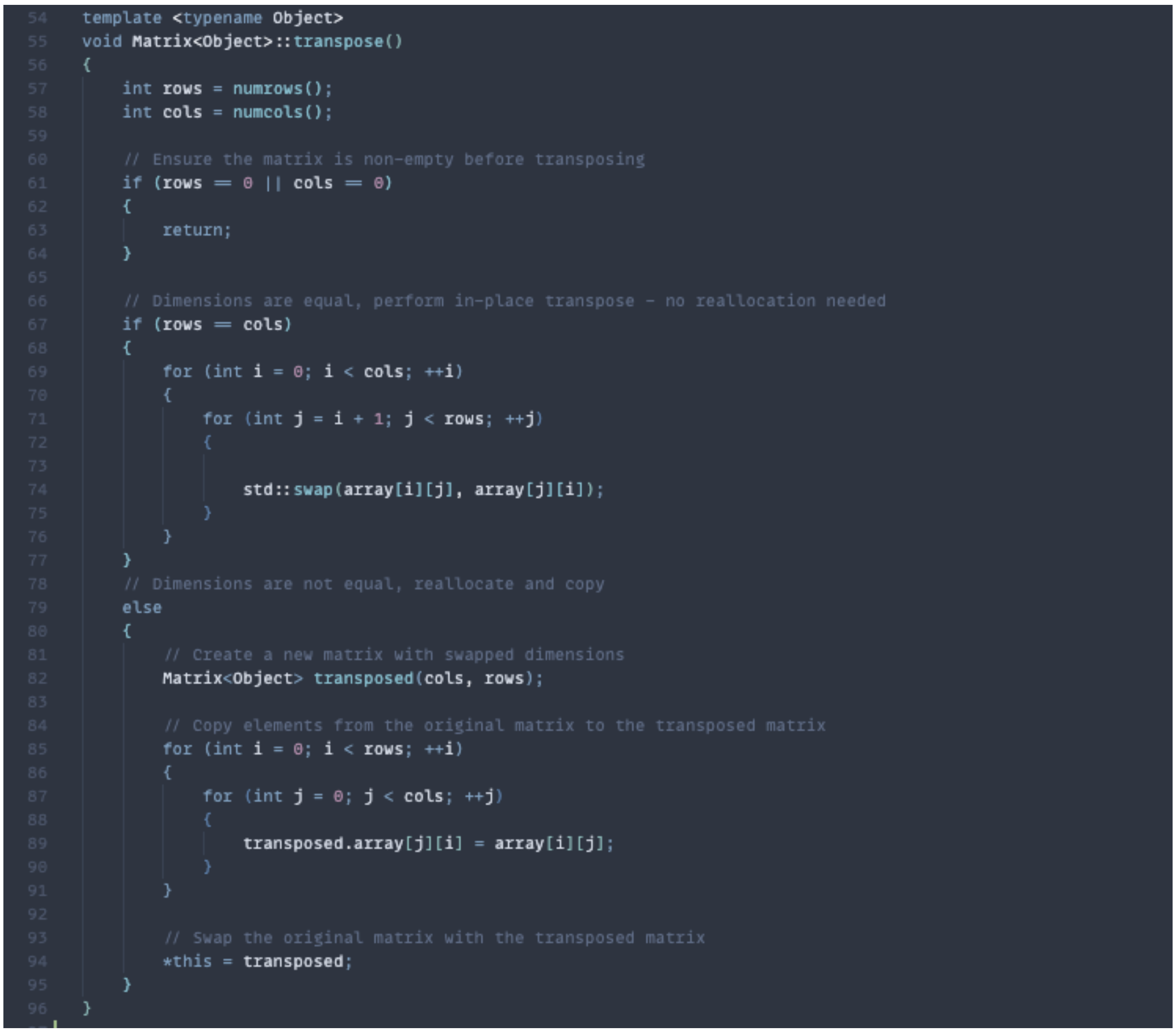


```cpp
class Stack
{
private:
    size_t size;
    int *array;
    int top;

public:
    Stack(size_t const size);
    ~Stack();
    void push(int const input);
    void pop();
    void print() const;
};

Stack::Stack(size_t const size)
{
    if (size <= 0)
    {
        this->size = 100;
        this->array = new int[size];
        this->top = -1;
    }
    else
    {
        this->size = size;
        this->array = new int[size];
        this->top = -1;
    }
}

Stack::~Stack()
{
    delete[] this->array;
}

void Stack::push(int const input)
{

    // Check if array is full
    if (this->top == this->size - 1)
    {
        // Reallocate the array by copying over the content into a new array
        int *newArray = new int[this->size * 2];
        for (int i = 0; i < this->size; i++)
        {
            newArray[i] = this->array[i];
        }
        this->size *= 2;

        // Delete the old array
        delete[] this->array;

        // Update the array pointer to point to the new array
        this->array = newArray;
    }

    // Add element to the end of the array
    this->array[++this->top] = input;
}

void Stack::pop()
{
    // Check if array is empty
    if (this->top == -1)
    {
        throw std::out_of_range("Stack is empty");
    }

    // remove the first element in the array
    this->array[this->top--];
}
```

*Figure 1: Exercise 1 stack implementation.*

## Exercise 2:

The worst-case big O notation for the transpose method is $O(M \cdot N)$ where M is columns and N is rows. This is due to the two for loops needed to swap the indexes of the arrays. If rows are equal to columns then no reallocation is needed and no temporary matrix is needed, however if they are not equal it is needed to reallocate into a new matrix and return that due to the new size requirement.

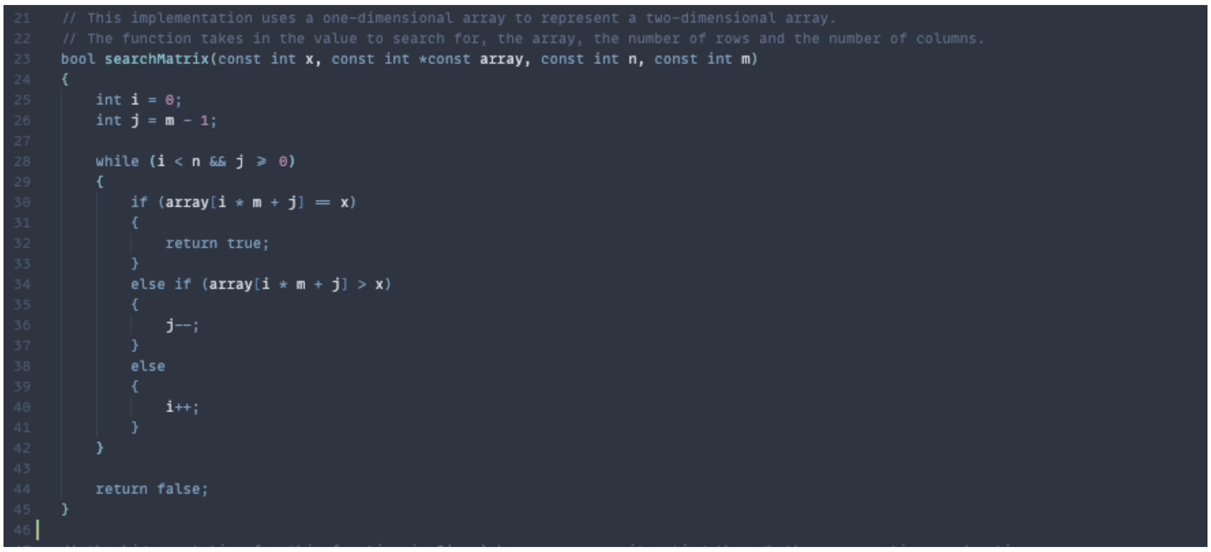


```cpp
template <typename Object>
void Matrix<Object>::transpose()
{
    int rows = numrows();
    int cols = numcols();

    // Ensure the matrix is non-empty before transposing
    if (rows == 0 || cols == 0)
    {
        return;
    }

    // Dimensions are equal, perform in-place transpose - no reallocation needed
    if (rows == cols)
    {
        for (int i = 0; i < cols; ++i)
        {
            for (int j = i + 1; j < rows; ++j)
            {

                std::swap(array[i][j], array[j][i]);
            }
        }
    }
    // Dimensions are not equal, reallocate and copy
    else
    {
        // Create a new matrix with swapped dimensions
        Matrix<Object> transposed(cols, rows);

        // Copy elements from the original matrix to the transposed matrix
        for (int i = 0; i < rows; ++i)
        {
            for (int j = 0; j < cols; ++j)
            {
                transposed.array[j][i] = array[i][j];
            }
        }

        // Swap the original matrix with the transposed matrix
        *this = transposed;
    }
}
```

*Figure 2: Exercise 2 transpose method implementation.*

## Exercise 3:

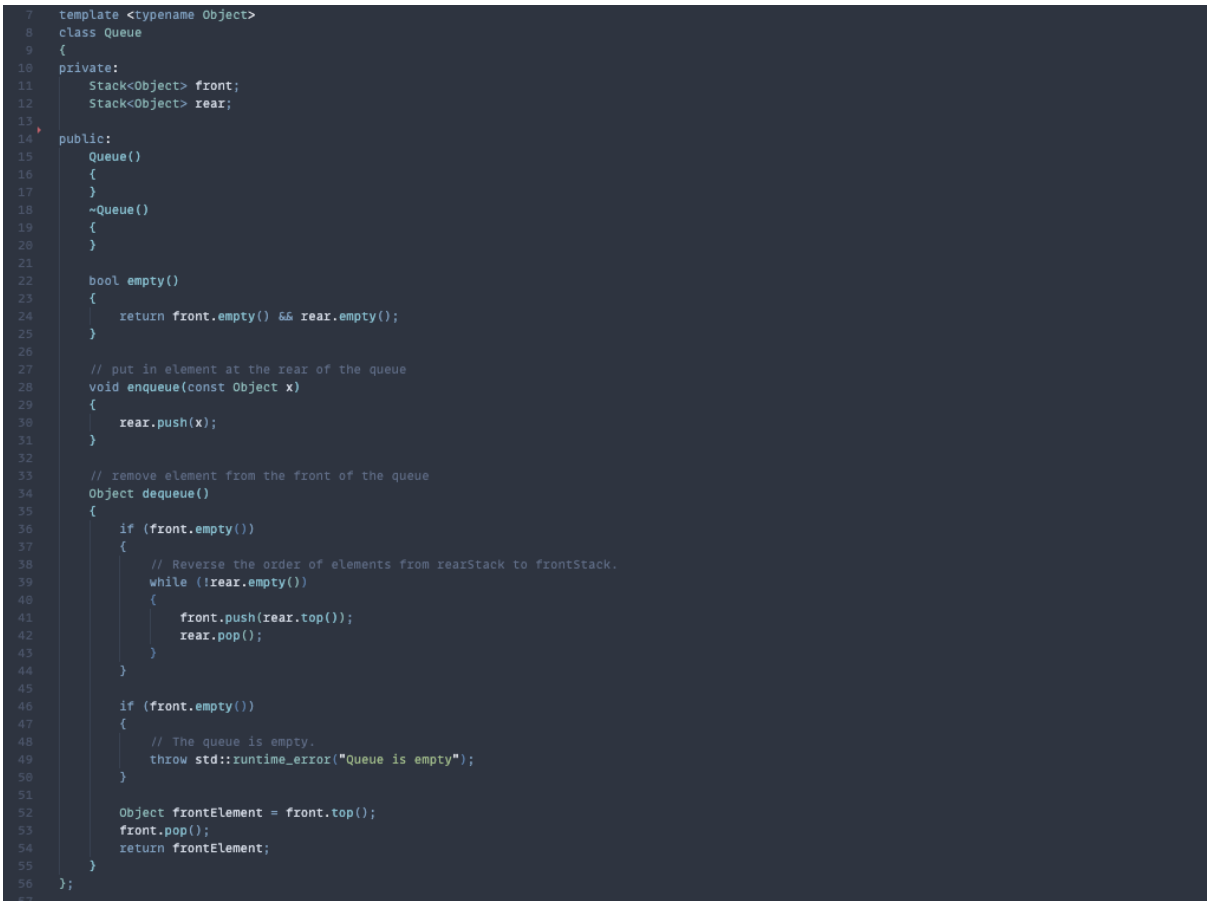
The worst-case time complexity for this function is $O(M + N)$, where M is columns and N is rows. This is the due to the worst case being a traversal of the entirety of the array to find the value.



```cpp
// This implementation uses a one-dimensional array to represent a two-dimensional array.
// The function takes in the value to search for, the array, the number of rows and the number of columns.
bool searchMatrix(const int x, const int *const array, const int n, const int m)
{
    int i = 0;
    int j = m - 1;

    while (i < n && j >= 0)
    {
        if (array[i * m + j] == x)
        {
            return true;
        }
        else if (array[i * m + j] > x)
        {
            j--;
        }
        else
        {
            i++;
        }
    }

    return false;
}
```

*Figure 3: Exercise 3 searchMatrix implementation using flat array as a matrix represenation.*

## Exercise 4:



```cpp
template <typename Object>
class Queue
{
private:
    Stack<Object> front;
    Stack<Object> rear;

public:
    Queue()
    {
    }
    ~Queue()
    {
    }

    bool empty()
    {
        return front.empty() && rear.empty();
    }

    // put in element at the rear of the queue
    void enqueue(const Object x)
    {
        rear.push(x);
    }

    // remove element from the front of the queue
    Object dequeue()
    {
        if (front.empty())
        {
            // Reverse the order of elements from rearStack to frontStack.
            while (!rear.empty())
            {
                front.push(rear.top());
                rear.pop();
            }
        }

        if (front.empty())
        {
            // The queue is empty.
            throw std::runtime_error("Queue is empty");
        }

        Object frontElement = front.top();
        front.pop();
        return frontElement;
    }
};
```

*Figure 4: Exercise 4 Queue implementation.*

## Exercise 5:

### A: Chaining

$$\lambda = \frac{9}{7} = 1{,}28$$

The recommended maximum value for chaining is 0.7 therefor we expand the total size of the array by doubling its capacity and choosing the next prime as the new capacity.

$$\lambda = \frac{9}{17} = 0.52$$

We are now within the limits of a valid lambda.

| Index | Value |
|---|---|
| 0 | 17 |
| 2 | 19 |
| 3 | 20 |
| 5 | 5 |
| 10 | 10 |
| 11 | 28 |
| 12 | 12 |
| 15 | 15 |
| 16 | 33 |

### B: Linear proping

$$\lambda = \frac{9}{7} = 1{,}28$$

The recommended maximum value for linear proping is 0.7 therefor we expand the total size of the array by doubling its capacity and choosing the next prime as the new capacity.

$$\lambda = \frac{9}{17} = 0.52$$

We are now within the limits of a valid lambda.

| Index | Value |
|---|---|
| 0 | 17 |
| 2 | 19 |
| 3 | 20 |
| 5 | 5 |
| 10 | 10 |
| 11 | 28 |
| 12 | 12 |
| 15 | 15 |
| 16 | 33 |

## C: Quadratic proping

The required lambda value for quadratic proping is 0.5, therefor we expand the total size of the array by doubling its capacity and choosing the next prime as the new capacity, until we reach a lambda value which is within the limits.

$$\lambda = \frac{9}{7} = 1{,}28$$

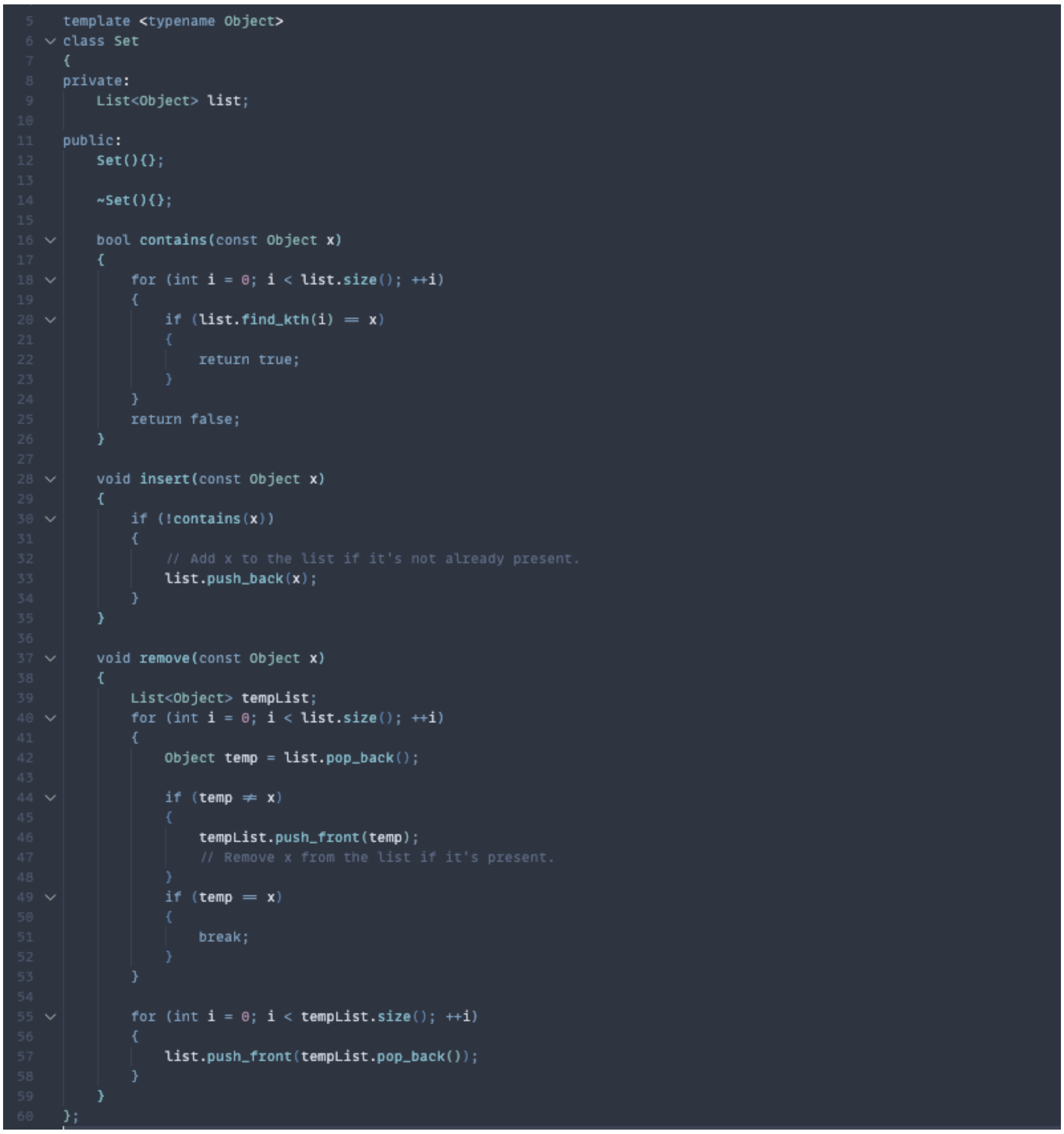
$$\lambda = \frac{9}{17} = 0.52$$

$$\lambda = \frac{9}{37} = 0.24$$

We are now within the limits of a valid lambda.

| Index | Value |
|---|---|
| 5 | 5 |
| 10 | 10 |
| 12 | 12 |
| 15 | 15 |
| 17 | 17 |
| 19 | 19 |
| 20 | 20 |
| 28 | 28 |
| 33 | 33 |

## Exercise 6:

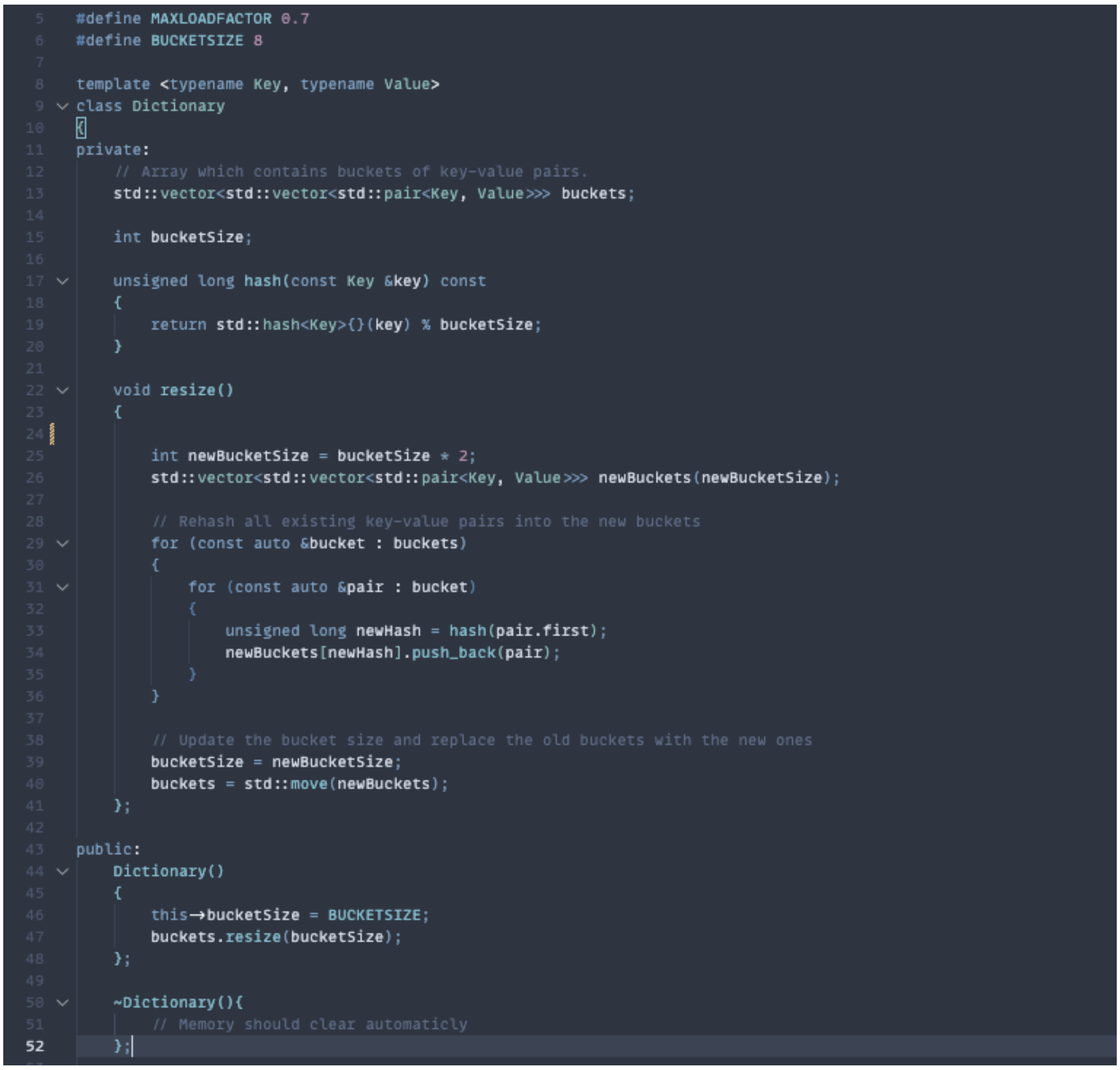


```cpp
template <typename Object>
class Set
{
private:
    List<Object> list;

public:
    Set(){};

    ~Set(){};

    bool contains(const Object x)
    {
        for (int i = 0; i < list.size(); ++i)
        {
            if (list.find_kth(i) == x)
            {
                return true;
            }
        }
        return false;
    }

    void insert(const Object x)
    {
        if (!contains(x))
        {
            // Add x to the list if it's not already present.
            list.push_back(x);
        }
    }

    void remove(const Object x)
    {
        List<Object> tempList;
        for (int i = 0; i < list.size(); ++i)
        {
            Object temp = list.pop_back();

            if (temp != x)
            {
                tempList.push_front(temp);
                // Remove x from the list if it's present.
            }
            if (temp == x)
            {
                break;
            }
        }

        for (int i = 0; i < tempList.size(); ++i)
        {
            list.push_front(tempList.pop_back());
        }
    }
};
```

*Figure 5: Exercise 6 Set implementation using a list and brute force.*

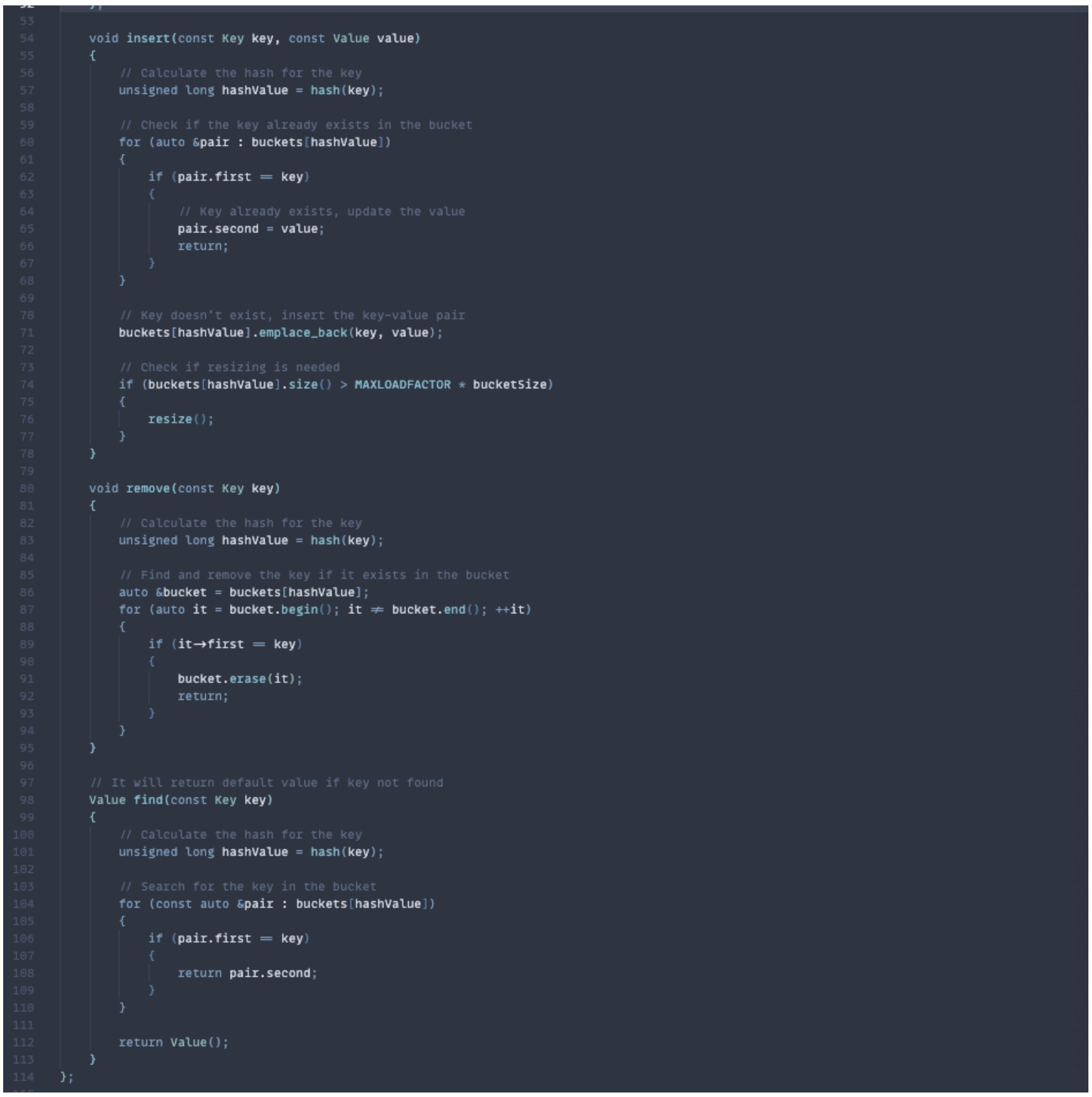
## Exercise 7:

A full hashmap implementation using arrays instead of linked lists to optimize for performance.



```cpp
#define MAXLOADFACTOR 0.7
#define BUCKETSIZE 8

template <typename Key, typename Value>
class Dictionary
{
private:
    // Array which contains buckets of key-value pairs.
    std::vector<std::vector<std::pair<Key, Value>>> buckets;

    int bucketSize;

    unsigned long hash(const Key &key) const
    {
        return std::hash<Key>{}(key) % bucketSize;
    }

    void resize()
    {

        int newBucketSize = bucketSize * 2;
        std::vector<std::vector<std::pair<Key, Value>>> newBuckets(newBucketSize);

        // Rehash all existing key-value pairs into the new buckets
        for (const auto &bucket : buckets)
        {
            for (const auto &pair : bucket)
            {
                unsigned long newHash = hash(pair.first);
                newBuckets[newHash].push_back(pair);
            }
        }

        // Update the bucket size and replace the old buckets with the new ones
        bucketSize = newBucketSize;
        buckets = std::move(newBuckets);
    };

public:
    Dictionary()
    {
        this->bucketSize = BUCKETSIZE;
        buckets.resize(bucketSize);
    };

    ~Dictionary(){
        // Memory should clear automaticly
    };
```

*Figure 6: Exercise 7 hashmap/dictionary implementation using array buckets.*



```cpp

    void insert(const Key key, const Value value)
    {
        // Calculate the hash for the key
        unsigned long hashValue = hash(key);

        // Check if the key already exists in the bucket
        for (auto &pair : buckets[hashValue])
        {
            if (pair.first == key)
            {
                // Key already exists, update the value
                pair.second = value;
                return;
            }
        }

        // Key doesn't exist, insert the key-value pair
        buckets[hashValue].emplace_back(key, value);

        // Check if resizing is needed
        if (buckets[hashValue].size() > MAXLOADFACTOR * bucketSize)
        {
            resize();
        }
    }

    void remove(const Key key)
    {
        // Calculate the hash for the key
        unsigned long hashValue = hash(key);

        // Find and remove the key if it exists in the bucket
        auto &bucket = buckets[hashValue];
        for (auto it = bucket.begin(); it != bucket.end(); ++it)
        {
            if (it->first == key)
            {
                bucket.erase(it);
                return;
            }
        }
    }

    // It will return default value if key not found
    Value find(const Key key)
    {
        // Calculate the hash for the key
        unsigned long hashValue = hash(key);

        // Search for the key in the bucket
        for (const auto &pair : buckets[hashValue])
        {
            if (pair.first == key)
            {
                return pair.second;
            }
        }

        return Value();
    }
};
```

*Figure 7: Exercise 7 hashmap/dictionary implementation using array buckets.*

## Exercise 8:

The hash values are placed based on the original placement order. The hash function calculates an index based on the modulus of the key and this value takes up the according index in the array with the principle of first come first served. In case there is a collision where the index is already taken, we use quadratic probing where we attempt to place it in the next index K squared. Which means the first attempt at relocating would be 1 index after and after 2 attempts it would be 3 indexes away.

Age: 25
Student number: 202006814

| Number: | Placement based on mod | Actual placement |
|---|---|---|
| 22 | 22 mod 11 = 0 | 0 |
| 5 | 5 mod 11 = 5 | 5 |
| 16 | 16 mod 11 = 5 | $5 + 1^2 = 6$ |
| 27 | 27 mod 11 = 5 | $5 + 2^2 = 9$ |
| 1 | 1 mod 11 = 1 | 1 |
| 12 | 12 mod 11 = 1 | $1 + 1^2 = 2$ |
| 25 | 25 mod 11 = 3 | 3 |
| 202006814 | 202006814 mod 11 = 8 | 8 |