Algoritmer Og datastrukturer #2

Weeks 3-4: Elementary data structures and their processing

Algoritmer of Datastrukturer SW23
Simon Dybdahl Damulog Andersen - 202204902
Emil Lydersen - 202207059
Max Glintborg Dall - 202208488

Indhold

Exercise 1

To implement a Stack using an ordinary array, the following code snippets were implemented.

```
void Stack::resizeStack() {
    int newSize = size * 2;
    int* newStack = new int[newSize];
    for (int i = 0; i <= top; i++) {
        newStack[i] = stack[i];
    }
    delete[] stack;
    stack = newStack;
    size = newSize;
}
```

Figure 1: code snippet of the resizeStack function

```
Stack stack;
stack.push(1);
stack.push(2);
stack.push(3);
for (int i = 0; i < 101; i++)
{
    stack.push(69);
}
```

Figure 2: code snippet of main for filling out the matrix

If the code above is run and it is filled out with 100 of the value 69. The below output gets generated in the terminal. Where the values 3,2,1 is the point after the resize happens, so the array is now extended to *2 and the old elements are moved over in the intended order.

```
Stack: 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69
 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69
 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 3 2 1
```

Figure 3: Output from the terminal when the code is run

```
int myMethod(int N)
{
    Stack stack;
    for (int i = 0; i < N; i++)
    {
        stack.push(69);
    }
    std::cout << "Stack: ";
    while (!stack.isEmpty()) {
        std::cout << stack.pop() << " ";
    }
    std::cout << std::endl;
    return N;
}

int main(void) {
    int N, msec = 0;
    printf("Input N:");
    scanf_s("%d", &N);
    clock_t before = clock();
    myMethod(N);
    clock_t duration = clock() - before;
    msec = duration * 1000 / CLOCKS_PER_SEC;
    printf("duration in msec: %d", msec);
    return 0;
}
```

Figure 4: Code snippet of the running time method

For the analysis of the running time, the code above code shows how much time it takes to run the program. If these times are plotted into a graph for N values. One would get a graph that has O(N) running time. (Graph is on next page)

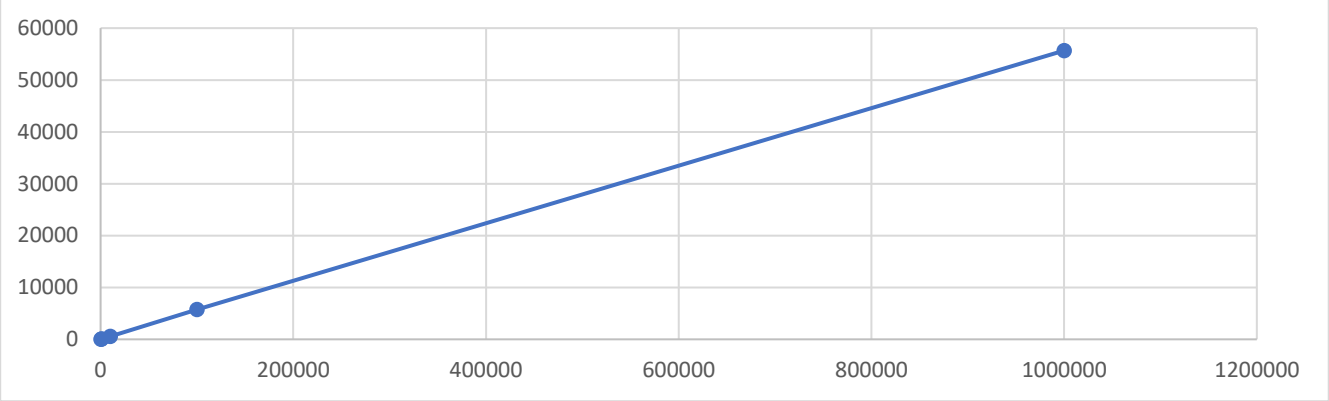


Figure 5: Graph showcasing the O(N) running time.

For N. the different values have been chosen to be:

100, 1000, 10000, 100000, 1000000. Which result in the above graph an goes to show that the running time is consistent with O(N).

Exercise 2

The program was written with the use of the Matrix class given by the professors. The following transpose () function was added as per the description of the assignment…

```
template <typename Object>
void Matrix<Object>::transpose() {
    int numRows = numrows();
    int numCols = numcols();

    // Create a new matrix to store the transposed data.
    Matrix<Object> transposed(numCols, numRows);

    for (int i = 0; i < numRows; ++i) {
        for (int j = 0; j < numCols; ++j) {
            transposed[j][i] = array[i][j]; // Transpose by copying elements
        }
    }

    // Replace the original matrix with the transposed matrix.
    array = transposed.array;
}
```

Figure 6: code snippet for the transpose function

The main is consistent with the test_matrix.cpp file given by the professors. The transpose was applied to the ‘mat’ matrix.

```
transpose=
| 141.08  154.62   75.56  141.81   59.73  190.26   64.99  142.06   51.70   68.27 |
|  77.93   83.44  117.44  184.95  155.49   92.72   29.62   62.82  151.60   73.90 |
|  85.68   94.74   58.72   17.51  138.77   40.56  170.30   11.00  151.90   79.81 |
|  75.75  179.94   20.36   79.92  100.26   54.70   72.98   29.46  103.70  102.95 |
| 160.73  129.78  120.33  174.41   12.89   69.20    5.49  101.77  132.29    0.00 |
```

Figure 7: terminal output for the transposed matrix

The terminal is included to confirm the functionality of the transpose function. The worst-case complexity can be concluded as O(n^2), since there is a nested for-loop, which is the functions upper bound. Therefore, there is no need to discuss the lower-bound complexities.

Exercise 3

The program was written building onto the code used in exercise 2, builds upon the way a matrix is made, and then iterates through the matrix with O(N) worst-case complexity when looking for the value x.

```
35  template <typename Object>
36  bool Matrix<Object>::isNumberInMatrix(Object x) {
37      int row = 0;  // Start from the top-left corner (INDEX)
38      int col = 0;  // Start from the left-most column (INDEX)
39      int valueData = 0; // Value of elements
40      // Fill NxN matrix with "n+1"
41      while (row < numrows() && col < numcols()) {
42          array[row][col] = valueData++;
43          if(col == numcols() -1 ){
44              row++;
45              col = 0;
46          }
47          else
48              col++;
49      }
50
51      // reset indecies
52      row = 0;
53      col = 0;
54
55      while (row < numrows() && col < numcols()) {
56          if (array[row][col] == x) {
57              return true;  // Found the number
58          }
59          else if(col==numcols() -1) {
60              row++;
61              col = 0;
62          }
63          else
64             col++;
65      }
66      return false;  // 'x' not found in the matrix
67  }
```

Figure 8: code snippet for finding 'x'

Disregard the filling algorithm only the bottom part after 'reset indecies' is applicable for this assignment.

Exercise 4

To write an implementation of Queue using Stack(s), start of by using the given code templates for double_list.h, double_list.tpp, queue_class.h, stack_class.h and test_queue_clas.cpp. The queue_class.h is the class that is going to be edited.

New functions and variables are added to the queue_class.h in the form of the following code:

Week_3-4 > Exercise_4 > C queue_class.h

```
#ifndef _QUEUE_H_
#define _QUEUE_H_

#include "stack_class.h"
#include <stdexcept>
using namespace std;

template <typename Object>
class Queue {
  private:
    Stack<Object> *stack1, *stack2; //enqueue & dequeue


  public:
    Queue() {
        stack1 = new Stack<Object>();
        stack2 = new Stack<Object>();
    }

    ~Queue() {
        delete stack1, stack2;
    }

    bool empty() { return (stack1->empty() && stack2->empty()); }

    Object front() {                        // front function
        if (empty()) {
            throw runtime_error("queue is empty.");
        }

        if (stack2->empty()) {              //if stack2 is empty, move all elements from stack1 to stack2
            while(!stack1->empty()) {
                stack2->push(stack1->pop()); //push the top element of stack1 to stack2
            }
        }
        return stack2->top(); //return the top element of stack2
    }

    Object get(){                               // dequeue function
        if (empty()) {
            throw runtime_error("queue is empty.");
        }

        if (stack2->empty()){                   //if stack2 is empty, move all elements from stack1 to stack2
            while(!stack1->empty()){            //while stack1 is not empty
                stack2->push(stack1->pop());    //push the top element of stack1 to stack2
            }
        }
        return stack2->pop(); //pop the top element of stack2
    }

    void put(const Object x) { // enqueue function
        stack1->push(x);
    };
};

#endif
```

Figure 9: Exercise 4 code

Here there is made use of two stacks (Stack1, Stack2). These two stacks will get enqueued and dequeued. Essentially the stacks are being used to go through the queue, by making use of ‘if’ and

while statements, and most importantly pop, push and top. These are used to move the values around in the queue.

For example, in the front function - Object front ():

```
Object front() {                        // front function
    if (empty()) {
        throw runtime_error("queue is empty.");
    }

    if (stack2->empty()) {              //if stack2 is empty, move all elements from stack1 to stack2
        while(!stack1->empty()) {
            stack2->push(stack1->pop()); //push the top element of stack1 to stack2
        }
    }
    return stack2->top(); //return the top element of stack2
}
```

Figure 10: Code snippet - Object front()

If stack2 is empty, then move all elements from stack1 to stack2 after which the top element from stack1 will be pushed to stack2. The return function then returns the top element of stack2.

The function – "Object get()":

```
Object get(){                           // dequeue function
    if (empty()) {
        throw runtime_error("queue is empty.");
    }

    if (stack2->empty()){               //if stack2 is empty, move all elements from stack1 to stack2
        while(!stack1->empty()){        //while stack1 is not empty
            stack2->push(stack1->pop());    //push the top element of stack1 to stack2
        }
    }
    return stack2->pop(); //pop the top element of stack2
}
```

Figure 11: Code snippet – "Object get()"

This function is used to pop the top element of stack2. if stack2 is empty already, then all elements will be moved from stack1 to stack2, whereafter the top element of stack1 is pushed to stack2. It then pops the top element of stack2, so effectively the order of the stack is reversed and emulating the functionality of Queue.

The queue and the stacks make use of two different implementations. The stack uses LIFO (last-in-first-out) while the queue makes use of the FIFO (first-in-first-out) implementation.

Exercise 5

When writing down the entries into a hash table something one must watch out for when inserting the keys is the load factor λ. The load factor λ is defined as the ratio of elements in the table to the table size. If the load factor is exceeded one need to rehash the old hash tables like so:

Capacity = nextPrime(2*oldCapacity)

Where “nextPrime” means next prime number. And then the whole hash table must be reorganized compared to the new capacity.

The capacity is defined by the type of method one uses. There are as described different ways to do the calculations but some of them are the ones below.

- Chaining
- Linear Probing
- Quadratic probing

Each of these different methods require different approaches. And has a different threshold for the load factor. Typically, one would keep the threshold for the load factor under 0,5 for quadratic probing and under 0,7 for chaining and linear probing.

For a hash table T with the size 7 and the hash function h(x) = x mod 7. With the keys 5, 28, 19, 15, 20, 33, 12, 17, 33 and 10 which need to be inserted.

The final hash tables will look like this respectively for Chaining, Linear probing and Quadratic probing (which uses the formula (((x mod 7) +i^2)mod 7))

0	28
1	14
2	
3	17 -> 10
4	
5	5 -> 19 -> 33 -> 13 -> 33
6	20

Figure 12: hash table showcasing chaining

0	17
1	33
2	19
3	20
4	
5	5
6	
7	
8	
9	
10	10
11	28
12	12
13	
14	
15	15
16	33

Figure 13: Hash table showcasing probing

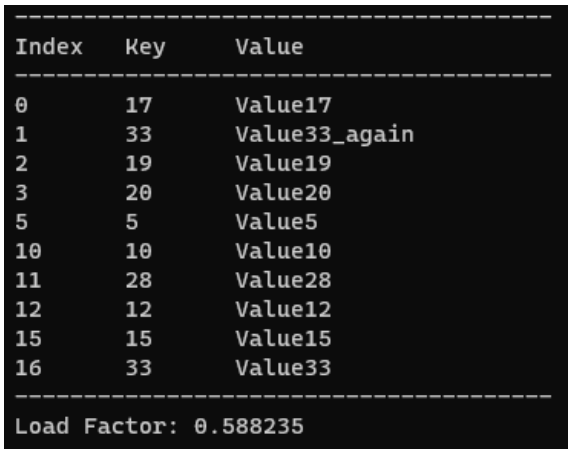
5	5
10	10
12	12
15	15
17	17
19	19
20	20
28	28
33	33
34	33

Figure 14: showcase a quadratic probing

As a little extra, we implemented this functionality in a little program. And these are the outputs

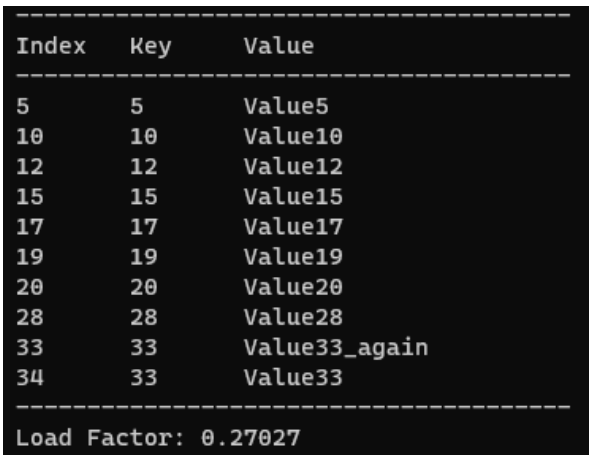
```
---------------------------------------
Index   Key     Value
---------------------------------------
0       28      Value28
1       15      Value15
3       17      Value17
3       10      Value10
5       5       Value5
5       19      Value19
5       33      Value33
5       12      Value12
5       33      Value33_again
6       20      Value20
---------------------------------------
Load Factor: 1.42857
```

Figure 15: Chaining output



```
-------------------------------------
Index   Key     Value
-------------------------------------
0       17      Value17
1       33      Value33_again
2       19      Value19
3       20      Value20
5       5       Value5
10      10      Value10
11      28      Value28
12      12      Value12
15      15      Value15
16      33      Value33
-------------------------------------
Load Factor: 0.588235
```

Figure 16 Linear probing output



```
-------------------------------------
Index   Key     Value
-------------------------------------
5       5       Value5
10      10      Value10
12      12      Value12
15      15      Value15
17      17      Value17
19      19      Value19
20      20      Value20
28      28      Value28
33      33      Value33_again
34      33      Value33
-------------------------------------
Load Factor: 0.27027
```

Figure 17 quadratic output

Exercise 6

For this assignment the objective was to make a Set with one of ADTs. For this purpose, a list was chosen, since it is possible to look through the contents and decide where to cut the list using a while loop and two conditionals, when implementing the sorting-like capability of a Set. The following is a screenshot of the final implementation of insert, other functions like print, insert, remove & contains were also added and available inside of the source code:

```
void insert(Object x) {
    List<Object>* temp = new List<Object>; // Temporary list to hold elements to keep
    int i = 0;
    if (!contains(x)) { // Check if the element is not already in the set
        while (!list->empty() && list->find_kth(i) <= x) {
            temp->push_front(list->pop_front()); // pop the lowest elemetns of list and store them in tem
        }

        list->push_front(x); // push element into where it belongs

        while (!temp->empty()) {
            list->push_front(temp->pop_back()); // push the highest to lowest element on to the Set
        }
        delete temp;
    }
    else if (list->empty())
    {
        list->push_front(x);
    }
}
```

Figure 18: Insert function made with list ADT.

```
Set elements:
1
2
3
4
Set elements after removing 3:
1
2
4
Is 4 in the set? Yes
```

Figure 19: Output from program, insert, remove and contains.

Exercise 7

To implement a dictionary using an STL vector, where the element of the vector is a pair that represents a key and a value, a dynamic vector can be used:

```
template <typename Key, typename Value> // Key and Value are template parameters

class Dictionary {
private:
    vector<pair<Key, Value>> data; // vector of pairs for storing the key-value pairs
```

Figure 20: STL vector with pairs

This makes it possible to insert both a key and value in the same vector as a pair. Pairs are used to combine two values that are of different data types. This could for example be a string and an integer or something else. In order to use pairs, the <utility> library is included.

```
public:
    void insert(const Key& key, const Value& value) // insert a new pair
    {
        data.push_back(make_pair(key, value)); // make_pair creates a pair object if it doesn't exist already
    }


    bool contains(const Key& key) const // check if the dictionary contains a key
    {
        for (const auto& pair : data)
        {
            if (pair.first == key) // if the key is found, return true
            {
                return true;
            }
        }
        return false;
    }

    Key& operator[](const Value& value) // return a reference to the value associated with the key
    {
        for (auto& pair : data) // iterate through the vector
        {
            if (pair.second == value)
            {
                return pair.first; // if the key is found, return the reference to the value
            }
        }
        return data.back().first; // return the reference to the value

    }

    bool empty() const // check if the dictionary is empty
    {
        return data.empty();
    }

    size_t size() const { // return the number of key-value pairs in the dictionary
        return data.size();
    }

};
```

Figure 21: Code snippet

In the code snippet above the functions for dictionary can be found. Full code can be found in the zip file under “Exercise_7”.

Here is the terminal when running the code:

```
The value of the key is:  4
The value of the key is:  7
The value of the key is:  10
The size of the dictionary is 3
Does the dictionary contain a key? 1
Dictionary is not empty
```

Figure 22: Dictionary terminal

Exercise 8

The hash table presented is correct it follows the formula (((x mod 7) +i^2)mod 7) which is what causes the value 27 to end up in index 9. And causes value 16 to end up in index 6:

```
-------------------------------------
Index   Key     Value
-------------------------------------
0       22      Value22
5       5       Value5
6       16      Value16
9       27      Value27
-------------------------------------
Load Factor: 0.363636
```

Figure 23 terminal output of the original hash table

Index	Value
0	22
1	
2	
3	
4	
5	5
6	16
7	
8	
9	27
10	

Figure 24 hash table from exercise

Down below is the hash table for the next part of this question after having added the new values and done a rehashing of the original hash table.

```
---------------------------------------
Index   Key     Value
---------------------------------------
0       23      Value23
1       1       Value1
4       27      Value27
5       5       Value5
9       202208488       studiernr
12      12      Value12
16      16      Value16
22      22      Value22
---------------------------------------
Load Factor: 0.347826
```

Figure 25 output from the terminal of the new hash table

index	Value
0	23
1	1
2	
3	
4	27
5	5
6	
7	
8	
9	202208488
10	
11	
12	12
13	
14	
15	
16	16
17	
18	
19	
20	
21	
22	22

Figure 26 The newly created hash table

When the hash table gets rehashed one must take the new size into account for both the new and old entries of values. Hence the reason the value 22 ends up at index 22 and some of the other values gets new index positions.

Age: 23

Student nr: 202208488

Bilag

Source Code:

Week_3-4_ElementaryDataStructsAndTheirProccessing.zip – Kan også finds på GitHub!

(DOA-Exercises/Week_3-4_ElementaryDataStructsAndTheirProccessing/Week_3-4 at main · SjioGG/DOA-Exercises (github.com))