SWO 3x **Übung zu Softwareentwicklung mit klassischen Sprachen u. Bibliotheken 3** **SS 2022, Übung 02**

Abgabe elektronisch bis Sa 8 Uhr in der KW 12

☐ ~~Gr. 1,~~ Winkler, BSc Msc **Name: Roman Kofler-Hofer** **Aufwand in h: 15**

☐ **Gr. 1,** Dr. Pitzer

Punkte: _____ **Kurzzeichen Tutor / Übungsleiter ____/____**

Beispiel	L Lösungsidee	I Implementierung	T Tests	S = L+I+T	Multiplikator	S*M
1a	3	4	3	10	6	60
1b		5	5	10	4	40
					Summe	**100**

1. Skip Lists

1.1. Lösungsidee:

Ich habe den Pseudocode des Papers implementiert. Von daher war die Lösungsidee schon vorgeben und deshalb führe ich diese hier nicht noch einmal im Detail aus. Grundsätzlich geht es eben darum über die Layer mehrere "Express-Lanes" einzuführen, über die man schneller zur gesuchten Position kommt. Ich habe mich gegen Shared Pointer entschieden. Den einzigen Mehraufwand den ich dadurch hatte war die Implementierung des Destruktors – welcher aber mit einem Listenüberlauf einfach gelöst werden konnte.

1.2. Quellcode

```
#pragma once
#include "utils.h"
#include <iostream>

using namespace std;


template<typename T, const int MAXLEVEL=32>
class skip_set
{
    private:

            class node
            {
                public:
                    T m_value;

                    //array which contains shared pointers of type node*
                    node* *m_forwardArray;

                    node(T val, int level) : m_value(val), m_forwardArray(new node*[level+1]{}) {}

                    ~node () {
                        delete[] m_forwardArray;
                        //std::cout << "destructor : " << m_value << std::endl;
                    }
            };

        node* m_head;
        int m_level{0};
        double m_p;
        int m_size{0};

        node* newNode (T val, int level) {
            node* n = new node(val, level);
            return n;
        }

        int randomLevel () {
```

```
        int newLevel = 0;
        double random = randomDoubleWithin(0, 0.99);

        while(random < m_p && newLevel < MAXLEVEL) {
            newLevel += 1;
            random = randomDoubleWithin(0, 0.99);
        }

        return newLevel;

    }


public:

    //value must be >= x < 1
    skip_set(double p=0.5) : m_p(p), m_head(new node(T{}, MAXLEVEL)) { }

    ~skip_set () {
        clear();
    }

    void clear() {
        node* n = m_head->m_forwardArray[0];
        node* succ = nullptr;

        while (n != nullptr) {
            succ = n->m_forwardArray[0];
            delete n;
            n = succ;
            m_size--;
        }

        for (int i = 0; i <= m_level; i++) {
            m_head->m_forwardArray[i] = nullptr;
        }
    }

    int size() const {return m_size;}
```

```
bool find (T value) {
    node* x = m_head;

    for(int i = m_level; i >= 0; i--) {

        while(x->m_forwardArray[i] != nullptr && x->m_forwardArray[i]->m_value < value) {
            x = x->m_forwardArray[i];
        }
    }

    x = x->m_forwardArray[0];
    return x != nullptr && x->m_value == value;
}

void insert (T value) {
    node* x = m_head;
    node* update[MAXLEVEL+1]{nullptr};

    for(int i = m_level; i >= 0; i--) {
        while(x->m_forwardArray[i] != nullptr && x->m_forwardArray[i]->m_value < value) {
                    x = x->m_forwardArray[i];
        }
        update[i] = x;
    }

    x = x->m_forwardArray[0];

    if (x == nullptr || x->m_value != value)
    {
        int new_level = randomLevel();

        if(new_level > m_level) {
            for (int i = m_level+1; i <= new_level; i++)
            {
                update[i] = m_head;
             }
                m_level = new_level;
            }

            node* insertedNode = newNode(value, new_level);
```

```
            for(int i = 0; i <= new_level; i++) {
                insertedNode->m_forwardArray[i] = update[i]->m_forwardArray[i];
                update[i]->m_forwardArray[i] = insertedNode;
            }

        m_size++;
        //cout << "Inserted value " << value << endl;
    }
}

bool erase (T value) {
    node* x = m_head;
    node* update[MAXLEVEL+1]{nullptr};

    for(int i = m_level; i >= 0; i--) {
        while(x->m_forwardArray[i] != nullptr && x->m_forwardArray[i]->m_value < value) {
            x = x->m_forwardArray[i];
        }
        update[i] = x;
    }
    x = x->m_forwardArray[0];

    if(x != nullptr && x->m_value == value) {
        for(int i = 0; i <= m_level; i++) {
            if(update[i]->m_forwardArray[i] != x)
                break;

            update[i]->m_forwardArray[i] = x->m_forwardArray[i];
        }
        delete x;
        m_size--;
        while(m_level > 0 && m_head->m_forwardArray[m_level] == nullptr) {
            m_level--;
        }

        return true;
    }
    else {
        return false;
    }
}
```

```
        friend std::ostream& operator << (std::ostream& os, const skip_set& s) {
            for(int i = 0; i <= s.m_level; i++) {
                node* current = s.m_head->m_forwardArray[i];
                os << "Level " << i << ": ";
                while(current != nullptr) {
                    os << current->m_value << " ";
                    current = current->m_forwardArray[i];
                }
                os << endl;
            }
            return os;
        }
};
```

1.3. Tests:

Insert:

- Skip Set zuerst size = 0, nach dem Einfügen 4 Elemente
- Ausgabe zeigt die eingefügten Knoten und deren Level

```
~~~~~~~~~~testInsert~~~~~~~~~~
mySet->size() -> 0
mySet->size() -> 4
Level 0: 5 8 10 67
Level 1: 5 8 10 67
Level 2: 10 67
Level 3: 10 67
Level 4: 10 67
```

Find:

```
~~~~~~~~~~testFind~~~~~~~~~~
mySet->size() -> 0
mySet->find(10) -> false
Inserted value 5
Inserted value 67
Inserted value 8
Inserted value 10
Level 0: 5 8 10 67
Level 1: 5 8
Level 2: 5
mySet->find(10) -> true
mySet->find(3) -> false
mySet->find(67) -> true
mySet->find(5) -> true
mySet->find(8) -> true
```

Erase

```
~~~~~~~~~~testErase~~~~~~~~~~
mySet->size() -> 0
mySet->erase(8) -> false
Inserted value 5
Inserted value 67
Inserted value 8
Inserted value 10
mySet->erase(8) -> true
Level 0: 5 10 67
Level 1: 5 10 67
Level 2: 10
Level 3: 10
Level 4: 10
mySet->erase(5) -> true
mySet->erase(67) -> true
mySet->erase(8) -> false
mySet->erase(10) -> true
Level 0:
mySet->size() -> 0
```

2. Laufzeitanalyse

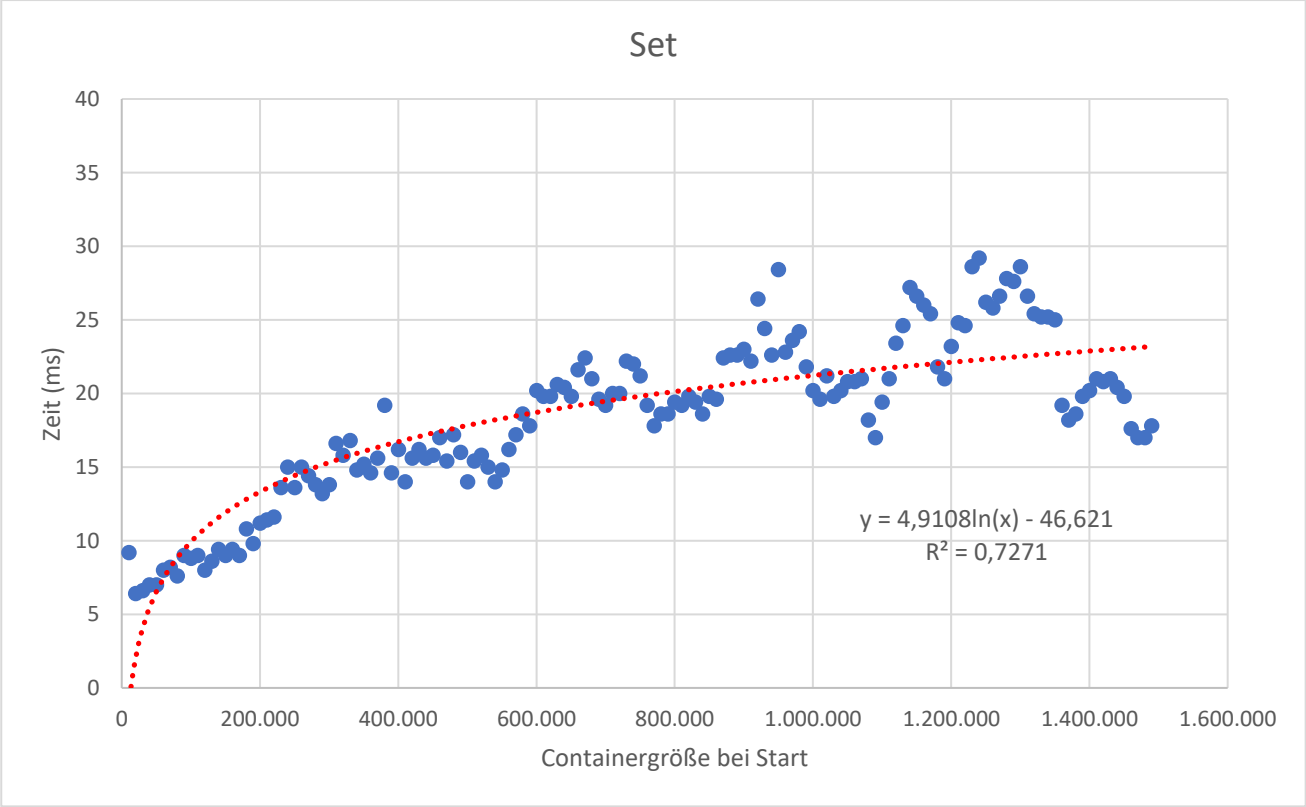
Ich habe eine Testklasse implementiert der ich mittels Template den Datentyp des Containers übergebe. Die Testklasse ist dafür ausgelegt entweder ein set<int> oder ein skip_set<int> aufzunehmen.

Getestet habe ich somit "nur" mit Int Werten. Außerdem habe ich mich auf die Insert-Methode fokussiert. Ich gehe davon aus, dass das Ergebnis für Find und Erase ganz ähnlich sein müsste, weil im Endeffekt ja auch wieder die Levels durchsucht werden, bis der Knoten gefunden wurde (und dieser eventuell gelöscht wird).

Teststrategie:

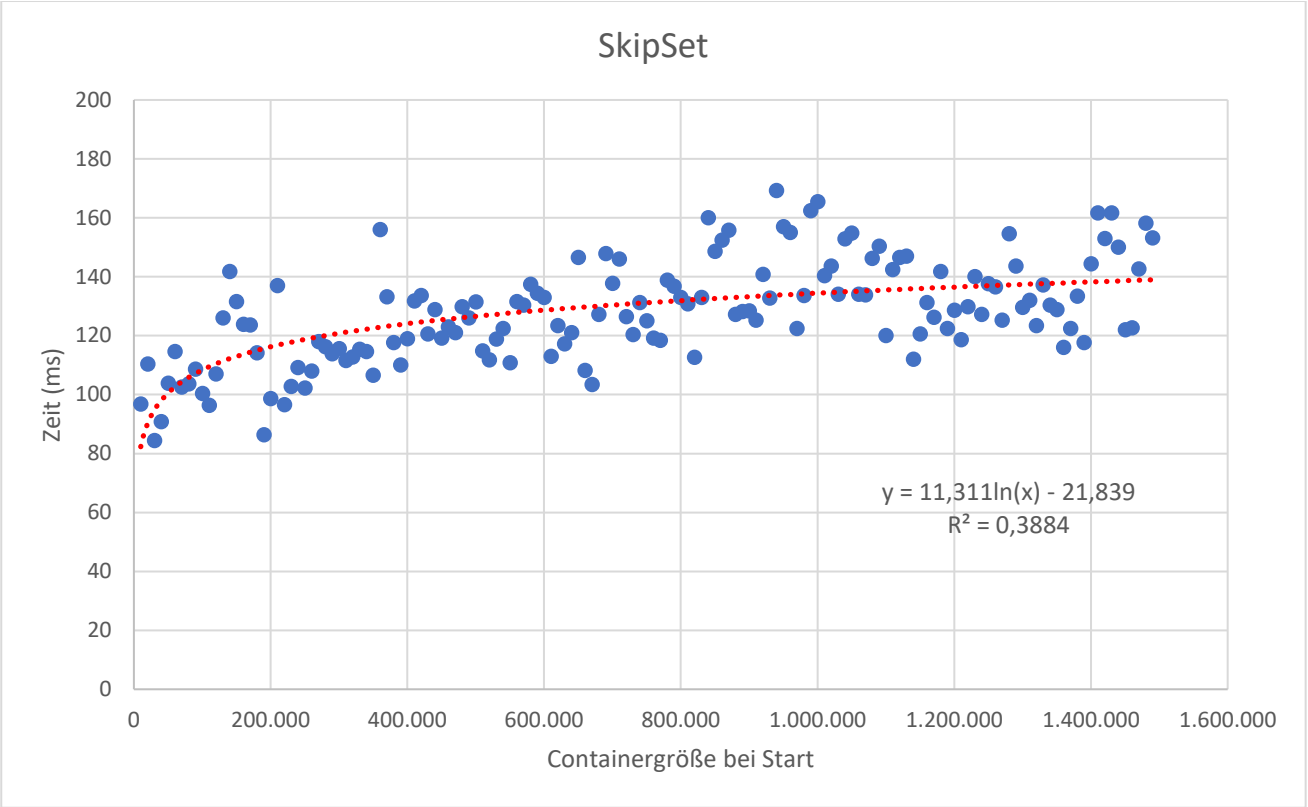
- Ich habe jeweils 10.000 zufällige (aber eindeutige) Zahlen in den Container eingefügt und dabei die Zeit gemessen.
- Das habe ich so lange wiederholt bis insgesamt 1.5 Mio Zahlen eingefügt wurden.
- Diesen Vorgang habe ich 5x wiederholt und den Durchschnitt der 5 Messungen gebildet.
- Somit gibt es 150 Datenpunkte (1.5 Mio / 10.k).
- Sowohl für das Set als das Skip_Set würde ich eine Komplexität von O(log n) erwarten.

Ergebnis Set:



Grafisch sieht man bereits einen Zusammenhang zwischen Containergröße und benötigter Zeit zum Einfügen von 10k Elementen. Mit zunehmender Größe steigt die Zeit aber nicht mehr ganz so stark an. Daher entspricht das Ergebnis meinen Erwartungen. Das Zusammenhangsmaß R^2 der Regressions-Analyse ergibt einen Wert von 0,72. D.h. 72% der Varianz werden durch das Datenmodell erklärt (der Zusammenhang zwischen Containergröße und Zeit ist stark).

Ergebnis Skip_Set:



Als erstes fällt auch, dass das Skip_Set grundsätzlich langsamer ist. Die schnellsten Einfüge-Vorgänge (von 1.000 Elementen) dauern 80 Millisekunden. Beim Set waren die langsamsten 1.000 Inserts bei 30 ms. Jedoch sieht man auch, dass die Kurve etwas weniger stark steigt, was auch das Zusammenhangsmaß R^2 zeigt. Dieses liegt bei 0,38. Somit ist der Zusammenhang zwischen Containergröße und Einfügezeitpunkt geringer als beim Set.

Fazit:

Was mich nicht überrascht ist, dass das Set grundsätzlich schneller ist, als das Skip_Set. Jedoch kommt mir der große Zusammenhang zwischen Containergröße und gemessener Zeit beim Set nicht ganz richtig vor. Eventuell liegt es daran, dass die Zeiten beim Set so gering sind und so Messungenauigkeiten stärker ins Gewicht fallen.

2.1. Quellcode:

```
#pragma once
#include "utils.h"
#include <chrono>
#include <iostream>
#include <fstream>
#include <string>
#include <map>
#include <iterator>
#include <algorithm>

using namespace std::chrono;
using namespace std;

template <typename T>
class container_test_kit
{
    using container_type = T;

    private:
        container_type data;
        std::string m_type;

    public:
        container_test_kit(std::string type) : m_type(type), data{} {}

        void insertPerformance() {

            //prepare random (unique) numbers that will be inserted afterwards
            int countRandomNumbers = 2000000;
            int* randomNumbersArray = new int[countRandomNumbers];
            for (int i = 0; i < countRandomNumbers; i++) {
                randomNumbersArray[i] = i;
            }
            shuffle(randomNumbersArray, randomNumbersArray + countRandomNumbers, default_random_engine(rand()));

            //test is 5 times repeated (average will be used)
```

```
int repetitions = 5;

//time is always measured for 10k inserts
int steps = 10000;

//until 1.5 Mio inserted values are reached
int insertedValues = 1500000;

//data structur where time for each 10k inserts per repetition is saved
multimap <int, double> measuredValues;

for (int i = 0; i < repetitions; i++) {
    while (data.size() < insertedValues) {
        size_t currentSize = data.size();

        auto start = high_resolution_clock::now();
        for (int insertCalls = 0; insertCalls < steps; insertCalls++) {
            data.insert(randomNumbersArray[currentSize+insertCalls]);
        }
        auto end = high_resolution_clock::now();
        duration<double, std::milli> timePassed = end - start;
        //cout << "measured time starting at: " << currentSize << " = " << timePassed.count() << endl;
        measuredValues.insert(make_pair(currentSize, timePassed.count()));
    }

    data.clear();
}

//prepare output files
std::string fileName;
if (m_type == "set") {
    fileName = "set_insert_performanceV2.txt";
}
else if (m_type == "skipSet") {
    fileName = "skipSet_insert_performanceV2.txt";
}

std::ofstream outfile{ fileName };
outfile << "size of dataStructure (before inserting 10k elements); milliseconds" << std::endl;

//sum up values for each key (0, 10k, 20k, 30,...) and calculate average
```

```
            for (int key = 0; key < insertedValues; key+=steps) {
                int sum = 0;
                double avg = 0;
                for (auto it = measuredValues.begin(); it != measuredValues.end(); it++) {
                    if (it->first == key) {
                        sum += it->second;
                    }
                }

                avg = (double)sum / repetitions;

                outfile << key << ";";
                outfile << avg << std::endl;
            }

            outfile.close();
            delete[] randomNumbersArray;
        }
};
```

3. Testprogramm (main)

```
#include "skip_set.h"
#include "container_test_kit.h"
#include <iostream>
#include <set>
#include <random>

using std::cout;
using std::endl;

#define DEBUG(X) std::cout << (#X) << " -> " << (X) << std::endl
#define LOG(X) std::cout << (#X) << std::endl; (X)

void testInsert () {

    cout << endl << "~~~~~~~~~~testInsert~~~~~~~~~~" << endl;
    skip_set<int>* mySet = new skip_set<int>(0.5);

    DEBUG(mySet->size());
    mySet->insert(5);
    mySet->insert(67);
    mySet->insert(8);
    mySet->insert(10);
    DEBUG(mySet->size());

    cout << *mySet;
    delete mySet;
}

void testFind () {

    cout << endl << "~~~~~~~~~~testFind~~~~~~~~~~" << endl;
    skip_set<int>* mySet = new skip_set<int>(0.5);
    cout << boolalpha;

    DEBUG(mySet->size());
    DEBUG(mySet->find(10));

    mySet->insert(5);
    mySet->insert(67);
    mySet->insert(8);
    mySet->insert(10);

    cout << *mySet;

    DEBUG(mySet->find(10));
    DEBUG(mySet->find(3));
    DEBUG(mySet->find(67));
    DEBUG(mySet->find(5));
    DEBUG(mySet->find(8));

    delete mySet;
}

void testErase () {
    cout << endl << "~~~~~~~~~~testErase~~~~~~~~~~" << endl;
    skip_set<int>* mySet = new skip_set<int>(0.5);

    DEBUG(mySet->size());
    DEBUG(mySet->erase(8));
```

```
    mySet->insert(5);
    mySet->insert(67);
    mySet->insert(8);
    mySet->insert(10);

    DEBUG(mySet->erase(8));
    cout << *mySet;

    DEBUG(mySet->erase(5));
    DEBUG(mySet->erase(67));
    DEBUG(mySet->erase(8));
    DEBUG(mySet->erase(10));
    cout << *mySet;
    DEBUG(mySet->size());

    delete mySet;
}

int main () {
    srand(time  (0));
    testInsert();
    testFind();
    testErase();

    container_test_kit<set<int>> setTest{"set"};
    //setTest.insertPerformance();

    container_test_kit<skip_set<int>> skipSetTest{"skipSet"};
    //skipSetTest.insertPerformance();

    return 0;
}
```