SWE 4x Übung zu Softwareentwicklung mit modernen Plattformen 4 **SS 2022, Übung 2**

Abgabetermin: SA 08:00 in KW 12

☒ **Gr. 1**, Winkler, BSc MSc **Name** Michael Gruber **Aufwand in h** 12

☐ **Gr. 2**, Dr.Pitzer

Punkte _____ **Kurzzeichen Tutor / Übungsleiter** _____ / _____

Beispiel	L Lösungsidee	I Implementierung	T Tests	S = L+I+T	M Multiplikator	S*M
a	0..3	0..4	0..3		6	
b		0..5	0..5		4	
				Summe (Erfüllungsgrad) **Bitte auch online ausfüllen!**		100

Bitte erstellen Sie für diese Übung ein ZIP-Archiv in dem im Unterverzeichnis „`doc`“, die gesamte Dokumentation (inkl. Projektbeschreibung, Lösungsidee und formatiertem Quelltext) als PDF vorliegt, sowie der Quelltexte in den Angegebenen Unterverzeichnissen des Verzeichnis „`src`“.

Denken Sie bitte auch an die Angabe des Aufwandes auf dem Deckblatt und online!

Skip Lists

Zur Realisierung der abstrakten Datenstrukturen sortierte Menge *(set)* und sortiertes assoziatives Feld *(dictionary, map)* werden meistens binäre Suchbäume verwendet. Damit die Operationen auf diese Datenstrukturen (Finden, Einfügen und Löschen) in allen Fällen eine logarithmische Laufzeitkomplexität aufweisen, muss der Suchbaum bei jeder schreibenden Operation ausbalanciert werden. Die Implementierung von balancierten Bäumen (Rot-Schwarz-Baum) ist allerdings sehr aufwändig.

In einer schon relativ alten Arbeit aus dem Jahr 1990 stellt William Pugh sogenannte *Skip Lists* vor (das vollständige Paper finden Sie unter https://homepage.divms.uiowa.edu/~ghosh/skip.pdf), welche sich ebenfalls zur Implementierung der oben angesprochenen abstrakten Datenstrukturen eignen. Skip-Lists haben zwar im Durchschnitt dasselbe asymptotische Laufzeitverhalten wie balancierte Suchbäume, sind aber bei Weitem einfacher zu implementieren.

a) Studieren Sie das Paper von Pugh und erstellen Sie auf Basis der daraus gewonnenen Erkenntnisse eine C++-Schablone *skip_set*, welche die abstrakte Datenstruktur Menge implementiert, allerdings intern Skip Lists zur Repräsentation der Knoten verwendet. Die Schnittstelle von *skip_set* soll zunächst folgenermaßen aussehen:

```
template<typename T, const int MAXLEVEL=32>
class skip_set {
public:
  skip_set();
  ~skip_set();
  int size() const;
  bool find(T value);
  void insert(T value);
  bool erase(T value);
  friend std::ostream& operator << (std::ostream &os, const skip_set &s);
};
```

b) Führen Sie für die Mengen-Implementierung in der STL und Ihre Implementierung ausführliche Laufzeitanalysen durch. Übersetzen Sie dazu Ihre Quelltexte mit aktivierter Optimierungsoption. Überprüfen Sie zunächst, ob das erwartete asymptotische Laufzeitverhalten tatsächlich eintritt und belegen Sie Ihre Beobachtungen mit einer Grafik. Präzisieren Sie die Unterschiede im Laufzeitverhalten, indem Sie aus Ihrem Datenmaterial eine Regressionsfunktion für beide Implementierungen ermitteln.

Lösungsidee und -beschreibung

Einleitung

In der letzten Übung mit der Programmiersprache C++ beschäftigen wir uns mit einer weiteren Datenstruktur – der sogenannten „Skip List“. Die Skip-Liste erweitert eine einfach verkettete Liste um zusätzliche Metainformationen und setzt auf ein „probabilistisches Balancing“, um eine bessere Zeitkomplexität zu erlangen. Bei korrekter Umsetzung verhält sie sich wie balancierte Bäume und weist eine logarithmische Laufzeitkomplexität für Suchen, Einfügen und Löschen auf: **O(log n)**.
Im Gegensatz zu balancierten Bäumen gestaltet sich die Implementierung zusätzlich einfacher, da keine streng erzwungene Balancierung erfolgen muss.

Da im Paper von William Pugh bereits Pseudocode enthalten ist, wird in der Lösungsbeschreibung noch einmal die Funktionsweise von Skip-Listen in eigenen Worten zusammengefasst und weniger auf die Implementierungsansätze eingegangen, da diese ohnehin eher „straight forward“ sein wird.

Aufbau einer Skip-Liste

Eine Skip-Liste ist in mehreren Schichten – sogenannten „Levels“ – aufgebaut. Die unterste Schicht repräsentiert eine gewöhnliche einfach-verkettete Liste, welche sortiert ist.

Darüber hinaus werden auf Basis dieser noch weitere Schichten aufgebaut, welche man sich jeweils wie bspw. zusätzliche Überholspuren auf einer Autobahn vorstellen kann: Mit einer bestimmten Wahrscheinlichkeit ist ein Element der darunterliegenden Schicht „i“ auch in der Ebene „i+1“ enthalten. Diese Wahrscheinlichkeit „p“ entspricht in der Regel ½ bzw. ¼.
Wichtig: Das erste Element der Skip-Liste ist in jedem Level enthalten!

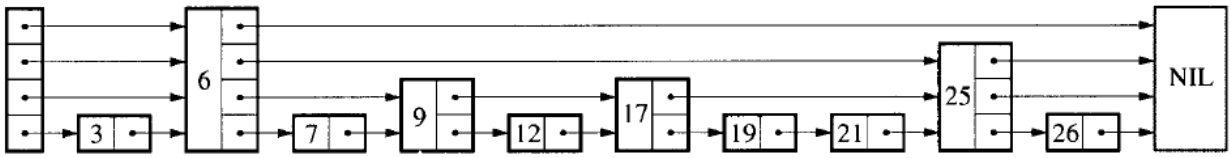


Abbildung 1: Darstellung der Skip-Liste von William Pugh in seinem Paper

In anderen Worten: Eine Skip-Liste ist also eine einfach-verkettete Liste mit ergänzenden „Forward-Pointer“, die auf Elemente zeigen, die nicht direkt nachfolgen. Bei Operationen werden somit Elemente übersprungen, deshalb auch die Namensgebung. Die Anzahl der „Forward-Pointer“ wird pro Knoten als „Level“ bzw. „Height“ bezeichnet.

Suchen

Die Suche in einer Skip-Liste startet beim obersten Level des ersten Knotens („head“). Innerhalb jedes Levels wird so lange zum nächsten referenzierten Element weitergegangen, bis der aktuelle Wert nicht mehr kleiner als der gesuchte Wert ist. Trifft dies zu, wird eine Ebene tiefer gegangen und der Algorithmus wiederholt, bis man im untersten Level angekommen ist.

Wurde der beschriebene Algorithmus ausgeführt, müsste man zu diesem Zeitpunkt vor dem gesuchten Element stehen, sodass man nur noch auf das nächste referenzierte Element weitergehen muss. Danach ist bekannt, ob das gesuchte Element auch tatsächlich existiert oder nicht.

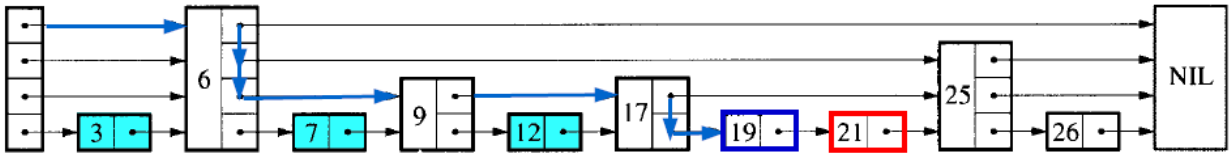


Abbildung 2: Ablauf der Suche in einer Skip-Liste

Der Ablauf der Suche des Elements „21“ in der gegebenen Skip-Liste ist zum besseren Verständnis visuell dargestellt. Wie man sieht, wird beim ersten Element beim obersten Level gestartet und so lange im selben Level weitergesucht, bis das aktuelle Element nicht mehr kleiner als das gesuchte Element ist, ehe auf eine Ebene tiefer weitergesucht wird. Diese Logik wird bis zum untersten Level wiederholt.

Auf unterster Ebene befindet man sich wieder bei der ursprünglichen einfach-verketteten Liste, sodass man bei Beendigung des Algorithmus direkt vor der gesuchten Stelle steht (siehe blaue Markierung). Deshalb ist es notwendig, noch einmal einen Schritt weiterzugehen, um zu ermitteln, ob das gesuchte Element enthalten ist (roten Markierung). Bei der Suche werden – wie in der Grafik hellblau markiert – Elemente übersprungen.

Einfügen

Für das Einfügen von Elementen ist grundsätzlich dasselbe Verfahren wie bei der Suche notwendig, da man anschließend bereits an der korrekten Einfügeposition steht. Der einzige Unterschied ist, dass zusätzlich beim Durchlaufen der verschiedenen Ebenen sich die „letzten“ Knoten eines jeden Levels vor dem angegebenen Wert gemerkt werden müssen.

Existiert ein Element bereits, ist nichts weiter zu tun, da es sich um ein „Set“ handelt und Duplikate nicht persistiert werden. Im Falle einer Einfügung wird dem neuen Knoten ein zufälliges „Level“ zugewiesen. Der neue Knoten kann unter Umständen zum aktuell „höchsten“ Knoten der Skip-Liste werden, sodass hier entsprechende Referenzen aktualisiert werden müssen. Zum Schluss müssen noch die vorhin bereits ermittelten „Forward Pointer“ bzgl. ihrer Referenzen auf den neuen Knoten aktualisiert werden.

Da beim Einfügen „nur“ Zeigerreferenzen von bestimmten Knoten aktualisiert werden, trägt dies enorm zu einer guten Laufzeiteffizient bei.

Löschen

Das Löschen ist dem Einfügen mit einer Ausnahme sehr ähnlich, nämlich dass nur mehr jene Referenzen aktualisiert werden müssen, welche auch tatsächlich auf den zu löschenden Knoten zeigen. Dies ist ähnlich zum Löschen in einer einfach verketteten Liste, wo auch die Referenzen auf den zu löschenden Knoten abgeändert werden müssen.

Beim Löschen des Knotens muss außerdem zusätzlich geprüft werden, ob dieser dem „höchsten“ Knoten (bezüglich des Levels) entspricht, da anschließend jene Levels wieder entfernt werden, welche keine Elemente haben. So werden unnötig gehaltene Levels vermieden.

Laufzeitanalyse

Für die Laufzeitanalyse wird im Main-Programm eine einfache Möglichkeit geschaffen, eine Zeitmessung für die Durchführung von Einfüge-Operationen auf der Skip-Liste sowie einem Set aus der Standardbibliothek vorzunehmen. Die Zeitmessung erfolgt mit „chrono“, welche bereits bei der letzten Übung mit einem Beispiel im Moodle zur Verfügung gestellt wurde.

super gut und ausführlich beschrieben. Hast dir echt die Arbeit gemacht, das Paper nochmal zusammenzufassen :) Chapeau!

Implementierung

Wie bereits in der Einleitung beschrieben, wurde auf eine ausführliche Dokumentation des Quellcodes verzichtet, da es sich hier um eine mehr oder weniger „vordefinierten Datenstruktur“ handelt.

Bei der Implementierung des Main-Programmes könnte man noch etwas an Zeit investieren, um das Auslesen der Programmargumente sicherer zu gestalten. Weiters könnten auch noch Code-Duplizierungen vermieden werden, indem Sourcecode der Zeitmessung abstrahiert wird und die zu messenden Tätigkeiten direkt als Lambda-Ausdruck übergeben werden – da es sich aber um eine Laufzeitmessung handelt, wollte ich die Aufrufe nicht unnötig in weitere Objekte und Methoden verpacken.

src/utils.h

```
#pragma once
#include <stdlib.h>

#define MIN_RANDOM_NUMBER 0
#define MAX_RANDOM_NUMBER 2000000

/// <summary>
/// Gets a random integer number between a given limit.
/// </summary>
inline int get_random_number(int min, int max) {
  return (rand() % (max - min + 1) + min);
}

/// <summary>
/// Gets a random integer number between RAND_MIN and RAND_MAX.
/// </summary>
inline int get_random_number() {
  return get_random_number(MIN_RANDOM_NUMBER, MAX_RANDOM_NUMBER);
}
```

src/skip_set.h

```
#pragma once
#include <iostream>

template <typename T, const int MAXLEVEL = 32>
class skip_set {
private:
  class skip_node {
  public:
    T value;
    skip_node** forward;

    skip_node(T value, int level)
      : value(value),
        forward(new skip_node*[level+1]()){
    }

    ~skip_node() {
      delete[] forward;
    }
  };

private:
  const float PROBABILITY{ 0.5f };

  int level;
  int size;
  skip_node* head;

  inline int get_random_level() {
    int lvl = 1;

    while (((float)std::rand() / RAND_MAX) < PROBABILITY && lvl < MAXLEVEL) {
      lvl++;
    }

    return lvl;
  }
```

```
public:
  skip_set() : level(0), size(0), head(nullptr) {
    head = new skip_node(-1, MAXLEVEL);
  }

  ~skip_set() {
    skip_node* x = head->forward[0];

    while (x != nullptr) {
      skip_node* tmp = x->forward[0];

      delete x;
      x = tmp;
    }

    delete head;
  }

  inline int get_size() const {
    return size;
  }

  inline bool find(T value) {
    skip_node* x = head;

    for (int i = level; i >= 0; i--) {
      while (x->forward[i] != nullptr && x->forward[i]->value < value) {
        x = x->forward[i];
      }
    }
    x = x->forward[0];

    return x != nullptr && x->value == value;
  }
```

Destruktor hätte man sich mit Smart Pointern gespart aber hab's auch ohne gemacht ;)

```
inline void insert(T value) {
  skip_node* x = head;
  skip_node* update[MAXLEVEL + 1] = { nullptr };

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

  x = x->forward[0];
  if (x == nullptr || x->value != value) {
    int new_level = get_random_level();

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

    x = new skip_node(value, new_level);
    size++;

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

```
inline bool erase(T value) {
  skip_node* x = head;
  skip_node* update[MAXLEVEL + 1];

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

  x = x->forward[0];
  if (x != nullptr && x->value == value) {
    for (int i = 0; i <= level; i++) {
      if (update[i]->forward[i] != x) {
        break;
      }
      update[i]->forward[i] = x->forward[i];
    }

    delete x;
    size--;

    while (level > 0 && head->forward[level] == nullptr) {
      level--;
    }
    return true;
  }

  return false;
}
```

```
  inline void print_level(std::ostream& os, int lvl) const {
    skip_node* x = head->forward[lvl];
    os << "[LVL " << lvl << "]: ";

    while (x != nullptr) {
      os << x->value << " ";
      x = x->forward[lvl];
    }
    os << std::endl;
  }

  inline void print(std::ostream& os) const {
    print_level(os, 0);
  }

  inline void print_all(std::ostream& os) const {
    os << "[SIZE]: " << size << std::endl;
    for (int i = 0; i <= level; i++) {
      print_level(os, i);
    }
  }

  inline friend std::ostream& operator<<(std::ostream& os, const skip_set& s) {
    return os << s.print(os);
  }
};
```

schön abstrahiert!

schwer sonst was zu sagen, nachdem es ja "eh" die Standardimplementierung gibt, an die du dich auch gehalten hast.

src/main.cpp

```cpp
#include <iostream>
#include <time.h>
#include <string>
#include <chrono>
#include <vector>
#include <set>
#include "utils.h"
#include "skip_set.h"

using namespace std;
using namespace std::chrono;

vector<int> get_data(int n) {
  vector<int> numbers;
  for (int i = 0; i < n; i++) {
    numbers.push_back(get_random_number());
  }

  return numbers;
}

void basic_skip_set_tests() {
  int n = 15;
  int non_existing_number = -24;
  skip_set<int> set;
  vector<int> v = get_data(n);

  cout << ">> inserting " << n << " elements..." << endl;
  for (int i = 0; i < n; i++) {
    set.insert(v[i]);
  }

  cout << ">> printing set..." << endl;
  set.print_all(cout);

  cout << ">> searching elements..." << endl;
  for (int i = 0; i < n; i++) {
    cout << "> found element " << v[i] << ": " << set.find(v[i]) << endl;
  }

  cout << ">> deleting elements..." << endl;
  for (int i = 0; i < n; i += 2) {
    cout << "deleted element " << v[i] << ": " << set.erase(v[i]) << endl;
  }
```

```
  cout << ">> printing set..." << endl;
  set.print_all(cout);

  cout << "> found element " << non_existing_number << ": " <<
set.find(non_existing_number) << endl;
  cout << "> delete element " << non_existing_number << ": " <<
set.erase(non_existing_number) << endl;
}

long long get_set_execution_time(int n) {
  vector<int> v = get_data(n);
  set<int> set;

  auto start = high_resolution_clock::now();
  for (int i = 0; i < n; i++) {

    set.insert(v[i]);
  }
  auto end = high_resolution_clock::now();

  duration<double> diff = end - start;
  return duration_cast<milliseconds>(diff).count();
}

long long get_skip_set_execution_time(int n) {
  vector<int> v = get_data(n);
  skip_set<int> set;

  auto start = high_resolution_clock::now();
  for (int i = 0; i < n; i++) {

    set.insert(v[i]);
  }
  auto end = high_resolution_clock::now();

  duration<double> diff = end - start;
  return duration_cast<milliseconds>(diff).count();
}

void test_avg_set_runtime(int n, int retries) {
  long long avg_execution_time = 0;
  long long execution_time = 0;

  for (int i = 0; i < retries; i++) {
```

```
    execution_time += get_set_execution_time(n);
  }

  avg_execution_time = execution_time / retries;
  cout << "avg set execution time: " << avg_execution_time << "ms" << endl;
}

void test_avg_skip_set_runtime(int n, int retries) {
  long long avg_execution_time = 0;
  long long execution_time = 0;

  for (int i = 0; i < retries; i++) {
    execution_time += get_skip_set_execution_time(n);
  }

  avg_execution_time = execution_time / retries;
  cout << "avg skip set execution time: " << avg_execution_time << "ms" << endl;
}

int main(int argc, char* argv[]) {
  if (argc != 2) {
    cout << "Usage: " << argv[0] << " <n>" << endl;
    return -1;
  }

  srand(time(NULL));
  cout << boolalpha;

  basic_skip_set_tests();

  int n = std::stoi(argv[1]);
  int retries = 15;
  cout << "N: " << n << ", RETRIES: " << retries << endl;

  test_avg_set_runtime(n, retries);
  test_avg_skip_set_runtime(n, retries);

  return 0;
}
```

Testfälle

Basistests

Vor der Laufzeitanalyse wurden die grundlegenden Operationen der Datenstruktur „skip_set“ auf Funktionalität geprüft.

```
>> inserting 15 elements...
>> printing set...
[LVL 0]: 2256 3352 3368 6239 7997 13581 13758 13953 19094 19509 19871 21083 25392 28878 32757
[LVL 1]: 2256 7997 13758 13953 19509 19871 21083 28878
[LVL 2]: 2256 19509
[LVL 3]: 2256
[LVL 4]:
>> searching elements...
> found element 28878: true
> found element 6239: true
> found element 2256: true
> found element 7997: true
> found element 3368: true
> found element 13953: true
> found element 3352: true
> found element 19871: true
> found element 19094: true
> found element 13758: true
> found element 13581: true
> found element 32757: true
> found element 25392: true
> found element 21083: true
> found element 19509: true
>> deleting elements...
deleted element 28878: true
deleted element 2256: true
deleted element 3368: true
deleted element 3352: true
deleted element 19094: true
deleted element 13581: true
deleted element 25392: true
deleted element 19509: true
>> printing set...
[LVL 0]: 6239 7997 13758 13953 19871 21083 32757
[LVL 1]: 7997 13758 13953 19871 21083
> found element -24: false
> delete element -24: false
```

Laufzeitanalyse

Setup

Für die Durchführung der Laufzeitanalyse wurde im Visual Studio für das Projekt zuvor die Optimierung aktiviert und infolgedessen die Basic Runtime Checks auf „Default“ gesetzt, da die dafür vordefinierte Einstellung nicht kompatibel zur Optimierungseinstellung /O2 ist.

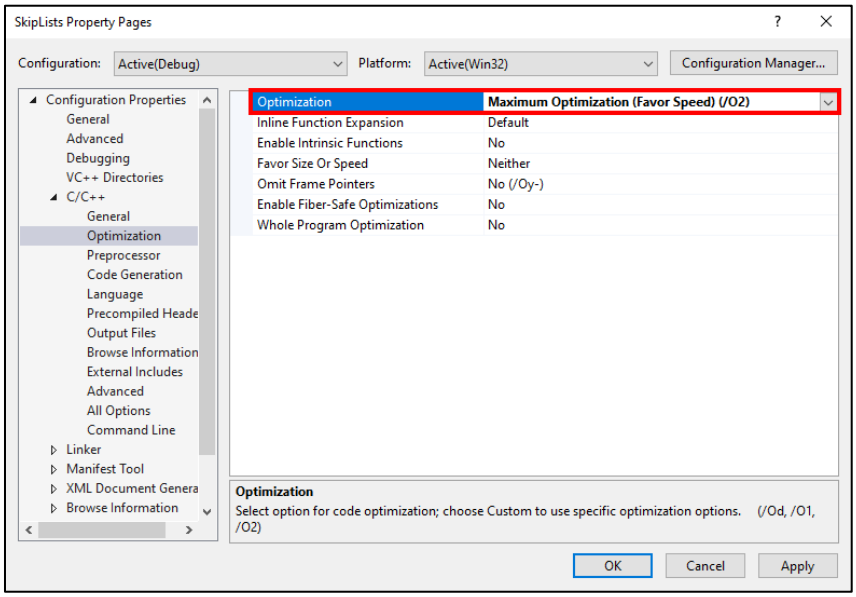


Abbildung 3: Aktivierung der Optimierung im VS-Projekt

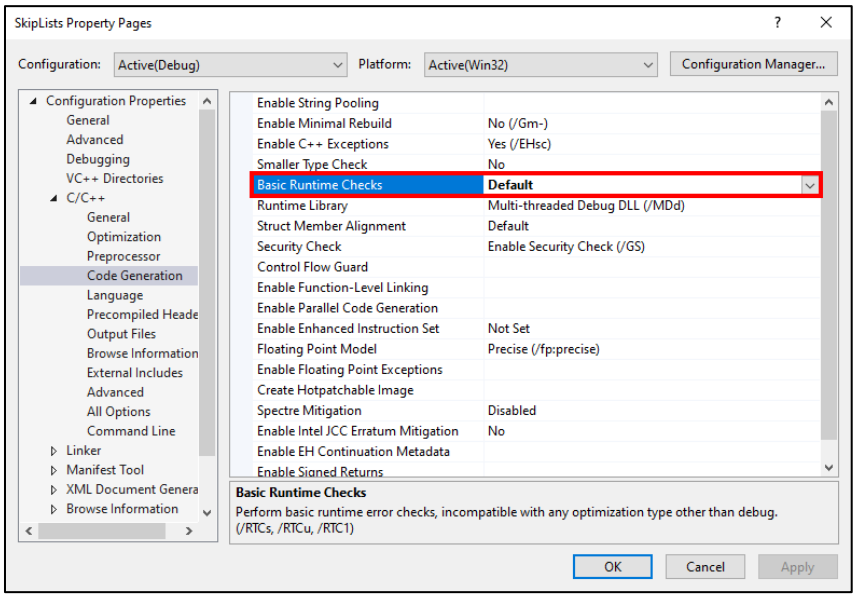


Abbildung 4: Abänderung der Einstellung für Basic Runtime Checks

Zu guter Letzt wurde das Programm im „Release“-Modus kompiliert, sodass die Laufzeittests mit dem Executable durchgeführt werden können.

Durchführung

Die Main-Methode wurde so konzipiert, dass über die Kommandozeilenparameter der Wert für „n“ eingelesen wird und so die Anzahl der Elemente gesteuert werden kann. Die Durchführung der Operation mit „n“ Elementen wird darüber hinaus wiederholt (standardmäßig 15x), sodass daraus ein Durchschnittswert ermittelt wird, welcher für die Interpretation der Laufzeit herangezogen wird.
Die Laufzeitanalyse wurde sowohl für „set“ aus der Standardbibliothek als auch „skip_set“ durchgeführt.

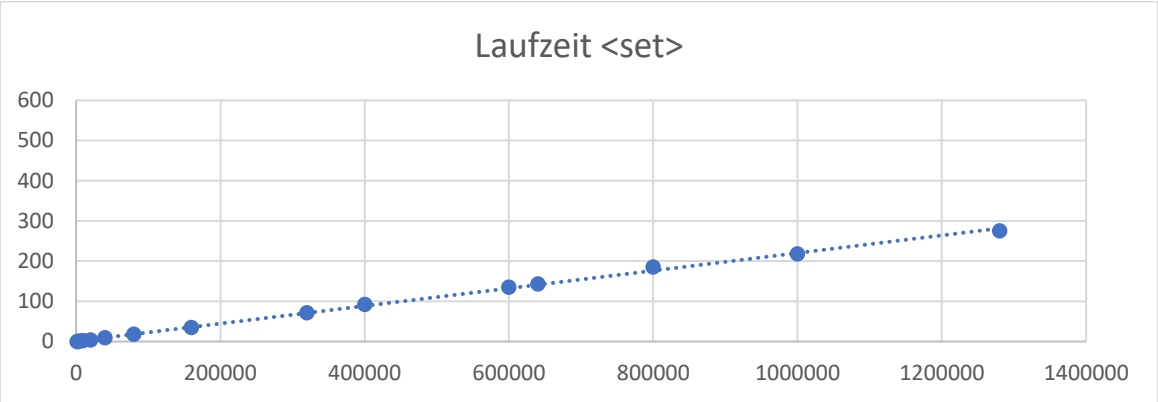
Aus der Analyse ergaben sich folgende Werte:

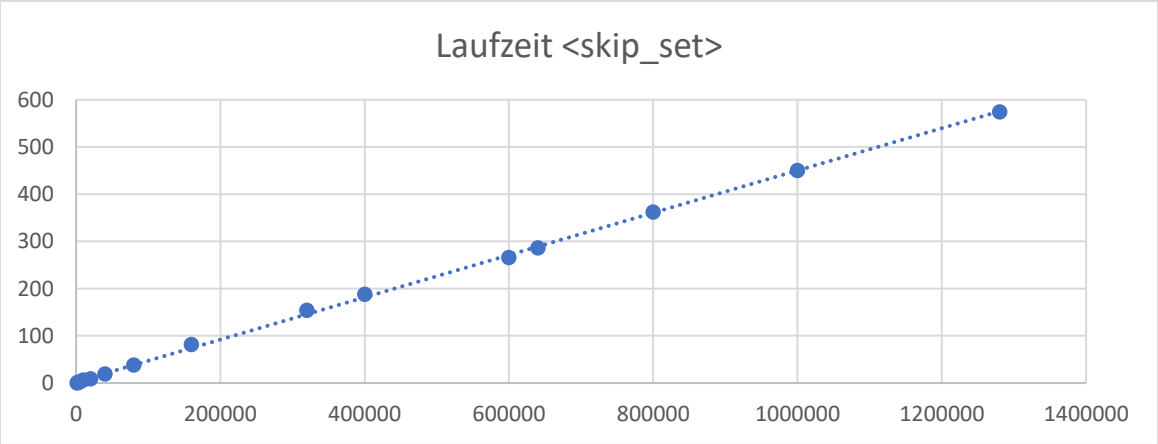
n (Anzahl Elemente)	<set>-Laufzeit in ms	<skip_set>-Laufzeit in ms
1.250	0	0
2.500	0	1
5.000	1	2
10.000	2	6
20.000	4	9
40.000	9	19
80.000	18	38
160.000	35	81
320.000	71	154
400.000	92	188
600.000	135	266
640.000	143	286
800.000	185	362
1.000.000	218	450
1.280.000	275	574

Erwartet wird eine **logarithmische Laufzeitkomplexität**. Wird die **Datenmenge jeweils verdoppelt**, steigt die **Laufzeit linear** an, wie es bspw. bei klassischen Suchbäumen der Fall ist.
Bei einer quadratischen Komplexität würde eine doppelte Datenmenge zu einer Vervierfachung der Laufzeit führen, wie es bspw. beim Bubble-Sort der Fall ist.

Regressionsgeraden

Die Regressionsgeraden für den Behälter aus der Standardbibliothek sowie der Implementierung von „skip_set“ unter Verwendung einer Skip-Liste sehen wie folgt aus:





Auch wenn man für die Anzahl der Elemente noch weit mehr Stichproben hätte nehmen können, so ist hier deutlich bei beiden Implementierungen ein linearer Anstieg der Laufzeit zu sehen. Die Analyse wurde auch mit ein paar Referenzwerten für das Löschen und Suchen von Elementen durchgeführt, wobei sich ein ähnliches Bild ergab.

Die Grafik belegt also die ursprünglich angenommene logarithmische Laufzeitkomplexität von Skip-Listen.

Ein erkennbarer Unterschied he szeit
im direkten Vergleich der beider u
erkennen. Das Einfügen von me n
Implementierung schon doppelt noch
einiges an Verbesserungspotent glich
noch weiter zu optimieren.

mich wundert gerade, dass das Set auch eine lineare Laufzeit aufweist. Hab für beide Implementierungen eigentlich einen logarithmischen Anstieg erwartet.

Ich hab so getestet: bei einer Container-Size von s=0 dauert das Einfügen von 10k Elementen x ms. Bei einer Size von s=10k Elementen dauert das Einfügen von weiteren 10k Elementen y ms usw. Also immer dieses Verhältnis aktuelle Containergröße zum Einfügen von 10k Elementen. Bin so auch auf einen eher logarithmischen Zusammenhang gekommen.