FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Prof. Dipl.-Inf. Ingrid Scholl

Dipl.-Ing. Norbert Kutscher

Michel Erbach, B.Sc.

Kathrin Petters, B.Sc.

Marcel Stüttgen, M.Eng.

Algorithmen und Datenstrukturen Praktikum

Studiengänge Informatik und Wirtschaftsinformatik

SS 2017

Inhaltsverzeichnis

0 Einleitung

Willkommen zum neuen Praktikum „Algorithmen und Datenstrukturen". In diesem Praktikum werden eine Auswahl der theoretisch vermittelten Datenstrukturen und einige Algorithmen aus der Vorlesung mit der Programmiersprache C++ implementiert. Insgesamt werden fünf Praktika zu den folgenden Themen gestellt:

1. Verkettete Listen
2. Ringpuffer und Binärer Suchbaum
3. Performanz-Analyse von Sortierverfahren
4. Algorithmen zu Graphen
5. NN

Was ist neu? Sie erhalten schon jetzt die ersten 2 Praktika-Aufgaben zusammengefasst in diesem Buch, die folgenden Aufgaben werden nach Ostern veröffentlicht. Wir informieren Sie mit den organisatorischen Details und fassen auch zusammen, was zu einer erfolgreichen Abgabe führt - sprich unsere Spielregeln. Wir möchten Sie schon jetzt darauf hinweisen, dass jede Praktika-Aufgabe sehr umfangreich ist, so dass diese in Heimarbeit schon gut von Ihnen vorbereitet werden muss. Wir veranschlagen dazu unter der Berücksichtigung der ECTS-Belastung für dieses Modul (8 ECTS=240h) pro Praktikum ca. 10-16h in der Vorbereitungszeit und 4h im Praktikumstermin. Falls Sie während ihrer Programmierung zu Hause Fragen haben sollten, stehen Ihnen folgende Möglichkeiten ohne Wertung der Reihenfolge offen:

- Tutoren und Mitarbeiter während der anderen Praktikatermine aufsuchen oder zu Beginn des eigenen Praktikumtermins fragen,
- Lerngruppen bilden (wir sind dabei behilflich, Gruppen zu vernetzen),
- Fragen im ILIAS-Forum stellen und diskutieren,
- Kommilitonen fragen (selbst vernetzt oder via Forum),
- die Sprechstunde von Frau Prof. Scholl nutzen (Mittwoch von 11:00 - 12:00 Uhr, Raum G303).

Wir wünschen Ihnen ein erfolgreiches Semester und viel Spaß mit „Algorithmen und Datenstrukturen"!

0.1 Team

Für die Betreuung des Praktikums stehen Ihnen während des Semesters folgende Ansprechpartner zur Verfügung:

- Professorin

Prof. Ingrid Scholl

scholl@fh-aachen.de
+49 241 6009 52177
Raum G 303
Sprechstunde: Mittwoch 11:00-12:00 Uhr, Raum G303

- Mitarbeiter

Dipl.-Ing. Norbert Kutscher

kutscher@fh-aachen.de
+49 241 6009 52174
Raum E 151
Sprechstunde: nach Vereinbarung

Kathrin Petters, B.Sc.

petters@fh-aachen.de
+49 241 6009 52241
Raum G 309
Sprechstunde: nach Vereinbarung

Michel Erbach, B.Sc.

erbach@fh-aachen.de
+49 241 6009 52100
Raum H 215
Sprechstunde: nach Vereinbarung

Marcel Stüttgen, M.Eng.

stuettgen@fh-aachen.de
+49 241 6009 52206
Raum G 301
Sprechstunde: nach Vereinbarung

0.2 Zeitplan und Termine

Die Tabelle 1 zeigt alle Termine der jeweiligen Gruppen inklusive der Raumzuordnung. Beachten Sie bitte, dass aufgrund der Feiertage im Mai/Juni zwei Freitagstermine auf **Dienstag 15:45-18:45 Uhr** in den **Räumen E141, E143, E145 und E146** verschoben wurden.

	Gruppen 2a-e Dienstag	**Gruppen 1a-e** Dienstag	**Gruppen 3a-d** Freitag	**Gruppen 4a-d** Freitag
Uhrzeit	12:15-15:15Uhr	12:15-15:15Uhr	12:15-15:15Uhr	12:15-15:15Uhr
Raum	2a: F104 2b: F106 2c: F111 2d: E141 2e: E143	1a: F104 1b: F106 1c: F111 1d: E145 1e: E143	3a: F104 3b: F106 3c: F111 3d: E141	4a: F104 4b: F106 4c: F111 4d: E145
P1	18.04.	25.04.	21.04.	28.04.
P2	02.05.	09.05.	05.05.	12.05.
P3	16.05.	23.05.	19.05.	**Di. 23.05.**
P4	30.05.	06.06.	02.06.	09.06.
P5	13.06.	20.06.	**Di. 13.06.**	23.06.

Tabelle 1: Terminplanung

ACHTUNG: Durch die Terminvorgaben aus Campus starten wir mit Gruppe 2 und 4 statt den Gruppen 1 und 3!

0.3 Klausurtermin(e)

Die vorläufigen Termine für die ADS-Klausur sind:

- Montag, 17.07.2017 von 8:30-11:30 Uhr,
- Montag, 25.09.2017 von 8:30-11:30 Uhr.

0.4 „Spielregeln“

- Die Bearbeitung der Praktikumsaufgaben erfolgt in 2er-Gruppen, im Sonderfall, z.B. bei ungerader Anzahl an Studierenden innerhalb einer Praktikumsgruppe, auch in einer 3er-Gruppe.
- Um ein Testat für eine Praktikumsaufgabe zu erhalten, sind Sie in der Lage...
  - ... jede Zeile des Codes zu erklären. Lösungen, die nicht erklärt werden können, werden nicht akzeptiert.
  - ... die verwendeten Datenstrukturen und Algorithmen auf Papier mit neuen Testdaten zu skizzieren und Laufzeitkomplexitäten zu berechnen und zu begründen.
  - ... vor Ort das Programm so anzupassen, dass es mit neuen Testdaten on-the-fly getestet werden kann.
  - ... den Debugger sinnvoll einzusetzen und die Speicherbelegung der erzeugten Daten nachzuvollziehen.
  - ... pünktlich zu Beginn des Praktikumtermins zu erscheinen.
  - ... eine Entwicklungsumgebung Ihrer Wahl (z.B. VisualStudio oder Eclipse für Windows-Plattformen, Qt-Creator oder makefiles für Linux-Plattformen, o.Ä.) einzusetzen und ein Projekt 'ADS-Praktikum' zuzüglich Unterprojekten pro Aufgabe zu erstellen.
  - ... eine im Optimalfall vollständig funktionierende Lösung einen Tag vor dem Praktikumstermin per Ilias hochzuladen. Sollten noch kleinere Fehler vorhanden sein, können diese während des Praktikumtermins mit unserer Unterstützung behoben werden. Sie müssen nicht zwangsweise jede Aufgabe einzeln hochladen, sie können z.B. auch Ihre Quelldateien zippen und dann fortlaufend aktualisieren.
- Gelbe/Rote Karten - Regel: Jeder Teilnehmer darf sich im Laufe des Semester ein „Foul“ erlauben (z.B. unentschuldigtes Fehlen, Aufgabe nicht rechtzeitig fertig, etc), welches dann mit der gelben Karte geahndet wird. Jedes weitere „Foul“ führt zur roten Karte und damit zum Platzverweis (Ausschluss aus dem Praktikum). Ein unentschuldigter Fehltermin muss nachgeholt werden.
- Plagiat-Regel: Das Kopieren von nicht selbst programmierten Source-Code (auch in Subteilen) ist nicht erlaubt, denn **Kopieren ist nicht gleich Kapieren!** Ein offensichtlicher Täuschungsversuch führt zum Ausschluss aus dem Praktikum.
- Für alle Praktikumstermine Ihrer Gruppe besteht grundsätzlich **Anwesenheitspflicht** (Ausnahme: s.u.). Sollten Sie verhindert sein, z.B. aus schwerwiegenden Gründen oder Krankheit, so ist der zuständige Praktikumsbetreuer **vorher**, z.B. per Email, darüber in Kenntnis zu setzen. Im Krankheitsfall ist zusätzlich ein ärztliches Attest einzureichen bzw. nachzuzeigen. Sollte dies nicht erfolgen, handelt es sich um ein unentschuldigtes Fehlen (Ausnahmen: z.B. Schneechaos bei Bus/ Bahn). „Rückstände“ in der Bearbeitung der Aufgaben sind selbstständig bis zum nächsten Termin aufzuarbeiten und zu Begin des Praktikumtermins vorzuzeigen!

- Sie möchten vorarbeiten? Gerne! Fleiß und Einsatz sollen belohnt werden! Sie können Aufgaben im Voraus bearbeiten und diese dann während des regulären Praktikums ihren Betreuern vorführen. Bitte haben Sie jedoch Verständnis, dass diese angehalten sind, zunächst alle „regulären“ Aufgaben zu testieren, so dass Sie ggf. etwas warten müssen, bis sich ein geeignetes Zeitfenster findet. Sollten Sie vorweg ein Testate erfolgreich erhalten, befreit Sie dies von der Anwesenheitpflicht für den entsprechenden regulären Termin.

Nach all diesen vielen Erstinformationen wünschen wir Ihnen ein erfolgreiches Semester und viel Spaß mit „Algorithmen und Datenstrukturen”!

1 Praktikum 1 : Erstellen Sie ein C++ Programm zur Speicherung von Tweets

Sie erstellen ein kleines Twitter-Programm. In diesem Programm lesen Sie kleine Tweets von der Tastatur ein. Diese werden mit Datum und Uhrzeit in einer verketteten Liste speichert. Alle bisherigen Tweet-Nachrichten sollen nach der Eingabe angezeigt werden.
In der Aufgabenstellung erhalten Sie Lösungshinweise und Testbeispiele. Vervollständigen Sie die gegebenen Programmteile und fügen Sie weitere Tests hinzu. Gehen Sie unbedingt in der Reihenfolge der Aufgabenstellung vor.
Die Aufgabenstellung gliedert sich in drei große Arbeitsschritte. Zu jedem Arbeitsschritt legen Sie bitte ein Projekt an, damit Sie die einzelnen Entwicklungsschritte im Praktikum erklären und zeigen können.

1.1 Aufgabenstellung

1. Der erste Teil der Aufgabe besteht darin, dass Sie eine Klasse bereit stellen, die die Datenstruktur einer dynamisch, doppelt verketteten Liste zur Verfügung stellt. Um die Aufgabe zunächst zu vereinfachen gehen wir von der Speicherung von integer-Werten (als key) aus.
   Zur Speicherung der Daten benötigen wir eine Klasse Node. Die Klasse List dient zur Verwaltung der Datenstruktur und die Klasse Node zu Speicherung der Inhalte.
   Wir benötigen folgende Dateien:

Node.h	Headerdatei der Klasse Node
Node.cpp	c++ Quelle der Klasse Node
List.h	Headerdatei der Klasse List
List.cpp	c++ Quelle der Klasse List
main.cpp	Testprogramm für die Klasse List

   Kopieren Sie die Dateien in ein neues C++ Projekt. Verändern Sie **nicht** die Headerdateien und auch nicht die Datei "Node.cpp". Fügen Sie den fehlenden Code in die Quelldateien ein (List.cpp oder main.cpp).

```
#ifndef _NODE_H
#define _NODE_H
class Node
{
public:
    int key;
    Node * next, * prev;
public:
    Node();
    Node(int key, Node * next = 0, Node * prev = 0);
    ~Node();
```

```
};
#endif
```

Listing 1: Node.h

```
#include "Node.h"
Node::Node()
{
    next = 0;
    prev = 0;
}
Node::Node(int key, Node * next, Node * prev)
{
    this->key = key;
    this->next = next;
    this->prev = prev;
}
Node::~Node()
{
}
```

Listing 2: Node.cpp

```
#ifndef _LIST_H
#define _LIST_H
#include "Node.h"
#include <string>
#include <iostream>

class List
{
    /*
    Die Klasse List dient zur Verwaltung von Knoten (Node). Mit Hilfe der Klasse List
    kann ein Stapel oder Warteschlange realisiert werden.
    */
private:
    struct form { std::string start = "<< "; std::string zwischen = ", "; std::string ende = " >>\n
    "; } _form;
    Node * head, *tail;          // Zeiger auf Kopf- und End-Element
    int _size;                   // Laenge der Kette
    bool temp;                   // normalerweise false; ist true, wenn es sich um eine temoraere
     Liste handelt
    // die innerhalb der ueberladenen Operatoren angelegt wird
public:
    List();
    List(const List & _List);    // Kopie Konstruktor
    List(const List * _List);    // Kopie Konstruktor
```

```
    ~List();
    void InsertFront(int key);        // Einfuegen eines Knotens am Anfang
    void InsertBack(int key);         // Einfuegen eines Knotesn am Ende
    bool getFront(int & key);         // Entnehmen eines Knoten am Anfang
    bool getBack(int & key);          // Entnehmen eines Knoten am Ende
    bool del(int key);                // loeschen eines Knotens [key]
    bool search(int key);             // Suchen eines Knoten
    bool swap(int key1, int key2);    // Knoten in der Liste vertauschen
    int size(void);                   // Groesse der Lise (Anzahl der Knoten)
    bool test(void);                  // Ueberpruefen der Zeigerstruktur der Liste
    void format(const std::string & start, const std::string & zwischen, const std::string & ende);
    // Mit der format-Methode kann die Ausgabe gesteuert werden: operator <<
    List & operator = (const List & _List);                    // Zuweisungsoperator definieren
    List & operator = (const List * _List);                    // Zuweisungsoperator definieren
    List & operator + (const List & List_Append);              // Listen zusammenfuehren zu einer
    Liste
    List & operator + (const List * List_Append);              // Listen zusammenfuehren zu einer
    Liste
    friend std::ostream & operator << (std::ostream & stream, List const & Liste);      //
    Ausgabeoperator ueberladen
    friend std::ostream & operator << (std::ostream & stream, List const * Liste);      //
    Ausgabeoperator ueberladen
};

#endif
```

Listing 3: List.h

In der C++ Quelle der Klasse List sind einige Methoden nur allgemein sprachlich in der zu erfüllenden Funktion beschrieben. Erzeugen Sie den notwendigen C++ Quellcode.
Die main.cpp enthält ein Testprogramm mit dem die Methoden der Klasse geprüft werden können. Dort können Sie auch weitere Testbeispiele einfügen.

```
#include "List.h"

List::List()
{
    head = new Node;
    tail = new Node;
    _size = 0;
    temp = false;
    head->next = tail;
    tail->prev = head;
}

List::List(const List & _List)
{
```

```
    // in dem Objekt _List sind die Knoten enthalten, die Kopiert werden sollen.
    // Kopiert wird in das Objekt "this"
    _form = _List._form;
    head = new Node;
    tail = new Node;
    _size = 0;
    temp = false;
    head->next = tail;
    tail->prev = head;
    Node * tmp_node;
    tmp_node = _List.head->next;
    while (tmp_node != _List.tail)
    {
        InsertBack(tmp_node->key);
        tmp_node = tmp_node->next;
    }
    if (_List.temp) delete & _List;      // ist die Uebergebene Liste eine temporaere Liste? -> aus
    Operator +
}

List::List(const List * _List)
{
    // in dem Objekt _List sind die Knoten enthalten, die Kopiert werden sollen.
    // Kopiert wird in das Objekt "this"
    _form = _List->_form;
    head = new Node;
    tail = new Node;
    _size = 0;
    temp = false;
    head->next = tail;
    tail->prev = head;
    Node * tmp_node;
    tmp_node = _List->head->next;
    while (tmp_node != _List->tail)
    {
        InsertBack(tmp_node->key);
        tmp_node = tmp_node->next;
    }
    if (_List->temp) delete _List;      // ist die Uebergebene Liste eine temporaere Liste? -> aus
    Operator +
}

List::~List()
{
    ( ... loeschen Sie alle noch vorhandenen Knoten Node dieser Instanz
        Denken Sie auch den die Knoten head und tail.)
```

```
}

void List::InsertFront(int key)
{
    ( ... Erweitern Sie die Methode so, dass ein neuer Knoten Node vorne
        (hinter dem Knoten head) in die Liste eingefuegt wird. )
}
void List::InsertBack(int key)
{
    ( ... Erweitern Sie die Methode so, dass ein neuer Knoten Node hinten
        (vor dem Knoten tail) in die Liste eingefuegt wird. )
}
bool List::getFront(int & key)
{
    ( ... Erweitern Sie die Methode so, dass der erste Knoten der Liste
        (hinter head) zurueckgegeben und dieser Knoten dann geloescht wird.
        Im Erfolgsfall geben Sie true zurueck, sonst false. )
}
bool List::getBack(int & key)
{
    (... Erweitern Sie die Methode so, dass der letzte Knoten der Liste
        (vor tail) zurueckgegeben und dieser Knoten dann geloescht wird.
        Im Erfolgsfall geben Sie true zurueck, sonst false. )
}
bool List::del(int key)
{
    (... Die Methode del sucht den Knoten mit dem Wert Key und loescht diesen
        im Erfolgsfall aus der Liste.
        Im Erfolgsfall geben Sie true zurueck, sonst false. )
}
bool List::search(int key)
{
    (... Die Methode search sucht den Knoten mit dem Wert key
        Im Erfolgsfall geben Sie true zurueck, sonst false. )
}
bool List::swap(int key1, int key2)
{
    (... Die Methode swap sucht den Knoten mit dem Wert key1,
        dann den Knoten mit dem Wert key2. Diese Knoten werden dann
        getauscht, indem die Zeiger der Knoten entsprechend geaendert
        werden. )
}
int List::size(void)
{
    (... Die Methode git den Wert von size (Anzahl der Knoten in der Liste) zurueck. )
}
```

```
bool List::test(void)
{
    (... Die Methode ueberprueft die Pointer der Liste. Gestartet wird mit head. Es werden alle
        Knoten bis zum tail durchlaufen von dort aus dann die prev-Zeiger bis zum head.
        Bei Erfolg wird true zurueck gegeben. )
}

void List::format(const std::string & start, const std::string & zwischen, const std::string & ende)
{
    _form.start = start;
    _form.zwischen = zwischen;
    _form.ende = ende;
}

List & List::operator = (const List & _List)
{
    // in dem Objekt _List sind die Knoten enthalten, die Kopiert werden sollen.
    // Kopiert wird in das Objekt "this"
    if (this == &_List) return *this;   //  !! keine Aktion notwendig
    _form = _List._form;
    Node * tmp_node;
    if (_size)
    {
        Node * tmp_del;
        tmp_node = head->next;
        while (tmp_node != tail)          // Alle eventuell vorhandenen Knoten in this loeschen
        {
            tmp_del = tmp_node;
            tmp_node = tmp_node->next;
            delete tmp_del;
        }
        _size = 0;
        head->next = tail;
        tail->prev = head;
    }
    tmp_node = _List.head->next;
    while (tmp_node != _List.tail)
    {
        InsertBack(tmp_node->key);
        tmp_node = tmp_node->next;
    }
    if (_List.temp) delete & _List;       // ist die Uebergebene Liste eine temporaere Liste? -> aus
    Operator +
    return *this;
}
```

```
List & List::operator = (const List * _List)
{
    // in dem Objekt _List sind die Knoten enthalten, die Kopiert werden sollen.
    // Kopiert wird in das Objekt "this"
    if (this == _List) return *this;      //  !! keine Aktion notwendig
    _form = _List->_form;
    Node * tmp_node;
    if ( _size )
    {
        Node * tmp_del;
        tmp_node = head->next;
        while (tmp_node != tail)          // Alle eventuell vorhandenen Knoten in this loeschen
        {
            tmp_del = tmp_node;
            tmp_node = tmp_node->next;
            delete tmp_del;
        }
        _size = 0;
        head->next = tail;
        tail->prev = head;
    }
    tmp_node = _List->head->next;
    while (tmp_node != _List->tail)
    {
        InsertBack(tmp_node->key);
        tmp_node = tmp_node->next;
    }
    if ( _List->temp) delete _List;       // ist die Uebergebene Liste eine temporaere Liste? -> aus
    Operator +
    return *this;
}

List & List::operator + (const List & List_Append)
{
    Node * tmp_node;
    List * tmp;
    if (temp){                            // this ist eine temporaere Liste und kann veraendert
    werden
        tmp = this;
    }
    else {
        tmp = new List(this);             // this ist keine temporaere Liste -> Kopie erzeugen
        tmp->temp = true;                 // Merker setzten, dass es sich um eine temporaere Liste
     handelt
    }
    if (List_Append._size) {              // anhaengen der uebergebenen Liste an tmp
```

```
        tmp_node = List_Append.head->next;
        while (tmp_node != List_Append.tail){
            tmp->InsertBack(tmp_node->key);
            tmp_node = tmp_node->next;
        }
    }
    if (List_Append.temp) delete & List_Append;     // wurde eine temporaere Liste uebergeben, dann
     wird diese geloescht
    return *tmp;
}

List & List::operator + (const List * List_Append)
{
    Node * tmp_node;
    List * tmp;
    if (temp){                                  // this ist eine temporaere Liste und kann veraendert
    werden
        tmp = this;
    }
    else {
        tmp = new List(this);                   // this ist keine temporaere Liste -> Kopie erzeugen
        tmp->temp = true;                       // Merker setzten, dass es sich um eine temporaere Liste
     handelt
    }
    if (List_Append->_size) {                   // anhaengen der uebergebenen Liste an tmp
        tmp_node = List_Append->head->next;
        while (tmp_node != List_Append->tail){
            tmp->InsertBack(tmp_node->key);
            tmp_node = tmp_node->next;
        }
    }
    if (List_Append->temp) delete List_Append;      // wurde eine temporaere Liste uebergeben,
    dann wird diese geloescht
    return *tmp;
}

std::ostream & operator<<(std::ostream  & stream, List const & Liste)
{
    stream << Liste._form.start;
    for (Node * tmp = Liste.head->next; tmp != Liste.tail; tmp = tmp->next)
        stream << tmp->key << (tmp->next == Liste.tail ? Liste._form.ende : Liste._form.zwischen
    );
    if (Liste.temp) delete & Liste;             // wurde eine temporaere Liste uebergeben, dann wird
    diese geloescht
    return stream;
}
```

```
std::ostream & operator << (std::ostream & stream, List const * Liste)
{
    stream << Liste->_form.start;
    for (Node * tmp = Liste->head->next; tmp != Liste->tail; tmp = tmp->next)
        stream << tmp->key << (tmp->next == Liste->tail ? Liste->_form.ende : Liste->_form.
    zwischen);
    if (Liste->temp) delete Liste;          // wurde eine temporaere Liste uebergeben, dann wird
    diese geloescht
    return stream;
}
```

Listing 4: List.cpp

```
#include <iostream>
#include "List.h"
using namespace std;

int main(void)
{
    int i;
    List MyList;

    for (i = 0; i < 10; i++){
        MyList.InsertFront(i + 1);
        MyList.InsertBack(i + 100);
    }

    cout << MyList;

    cout << "100: " << (MyList.search(100) ? "gefunden" : "nicht gefunden") << endl;
    cout << "99: " << (MyList.search(99) ? "gefunden" : "nicht gefunden") << endl << endl;

    while (MyList.getBack(i))
        cout << i << " ";
    cout << endl << endl;

    List MyList1, MyList2, MyList3;
    List * MyList_dyn = new List;

    for (i = 0; i < 10; i++){
        MyList1.InsertFront(i + 1);
        MyList2.InsertBack(i + 100);
    }

    MyList1.format("MyList1\n<<", ", ", ">>\n\n");
    cout << MyList1;
```

```
    MyList_dyn->format("MyList_dyn\n<<", ", ", ">>\n\n");
    MyList_dyn->InsertFront(-111);
    cout << MyList_dyn;

    MyList2.format("MyList2\n<<", ", ", ">>\n\n");
    cout << MyList2;

    MyList3 = MyList1 + MyList_dyn + MyList2;

    delete MyList_dyn;

    cout << "Groesse von MyList3: " << MyList3.size() << endl << endl;

    MyList3.format("MyList3\n<<", ", ", ">>\n\n");
    cout << MyList3 << endl;

    cout << "tauschen von 8 mit 103\n\n";
    MyList3.swap(8, 103);

    cout << MyList3;

    if (MyList3.test())
        cout << "MyList3: Zeiger OK\n";
    else
        cout << "MyList3: Zeiger ******Error\n";

    return 0;
}
```

Listing 5: main.cpp

2. Der zweite Teil der Aufgabe besteht darin, dass Sie die Klasse List und die Klasse Node je zu einer Template Klasse umbauen.
   Gehen Sie wie folgt vor:
   Kopieren Sie alle Dateien in ein neues Projekt. Anschließend wird der Programmteil von Node.cpp in die Datei Node.h kopiert (unterhalb der Klassendefinition). Anschließen kopieren Sie den Programmteil von List.cpp in List.h und löschen dann die Dateien Node.cpp und List.cpp.

   Die Überladenen Operatoren werden wie normale Methoden zu Template-Methoden umgesetzt. Eine Besonderheit sind die *friend*-Operatoren. Hier ist eiene zusätzliche Template-Deklarierung notwendig. Aus diesem Grund sind diese Teile des Quellcodes nochmals vorgegeben.

```
#ifndef _LIST_H
#define _LIST_H
#include "Node.h"
```

```
#include <string>
#include <iostream>

template <class T>
class  List
{

    .
    .
    .

    void format(const std::string & start, const std::string & zwischen, const std::string & ende);
    // Mit der format-Methode kann die Ausgabe gesteuert werden: operator <<
    List<T> & operator = (const List<T> & _List);                    // Zuweisungsoperator
    definieren
    List<T> & operator = (const List<T> * _List);                    // Zuweisungsoperator
    definieren
    List<T> & operator + (const List<T> & List_Append);              // Listen zusammenfuehren
    zu einer Liste
    List<T> & operator + (const List<T> * List_Append);              // Listen zusammenfuehren
    zu einer Liste
    template <typename Tf>
    friend std::ostream & operator << (std::ostream & stream, const List<Tf> & Liste);      //
    Ausgabeoperator ueberladen
    template <typename Tf>
    friend std::ostream & operator << (std::ostream & stream, const List<Tf> * Liste);      //
    Ausgabeoperator ueberladen
};

.
.
.

template <class T>
void List<T>::format(const std::string & start, const std::string & zwischen, const std::string &
    ende)
{
    _form.start = start;
    _form.zwischen = zwischen;
    _form.ende = ende;
}

template <class T>
List<T> & List<T>::operator = (const List<T> & _List)
{
    // in dem Objekt _List sind die Knoten enthalten, die Kopiert werden sollen.
```

```
    // Kopiert wird in das Objekt "this"
    if (this == &_List) return *this;   //  !! keine Aktion notwendig
    Node<T> * tmp_node;
    if (_size)
    {
        Node<T> * tmp_del;
        tmp_node = head->next;
        while (tmp_node != tail)        // Alle eventuell vorhandenen Knoten in this loeschen
        {
            tmp_del = tmp_node;
            tmp_node = tmp_node->next;
            delete tmp_del;
        }
        _size = 0;
        head->next = tail;
        tail->prev = head;
    }
    tmp_node = _List.head->next;
    while (tmp_node != _List.tail)
    {
        InsertBack(tmp_node->key);
        tmp_node = tmp_node->next;
    }
    if (_List.temp) delete & _List;     // ist die Uebergebene Liste eine temporaere Liste? -> aus
    Operator +
    return *this;
}

template <class T>
List<T> & List<T>::operator = (const List<T> * _List)
{
    // in dem Objekt _List sind die Knoten enthalten, die Kopiert werden sollen.
    // Kopiert wird in das Objekt "this"
    if (this == _List) return *this;    //  !! keine Aktion notwendig
    Node<T> * tmp_node;
    if (_size)
    {
        Node<T> * tmp_del;
        tmp_node = head->next;
        while (tmp_node != tail)        // Alle eventuell vorhandenen Knoten in this loeschen
        {
            tmp_del = tmp_node;
            tmp_node = tmp_node->next;
            delete tmp_del;
        }
        _size = 0;
```

```
        head->next = tail;
        tail->prev = head;
    }
    tmp_node = _List->head->next;
    while (tmp_node != _List->tail)
    {
        InsertBack(tmp_node->key);
        tmp_node = tmp_node->next;
    }
    if (_List->temp) delete _List;      // ist die Uebergebene Liste eine temporaere Liste? -> aus
    Operator +
    return *this;
}

template <class T>
List<T> & List<T>::operator + (const List<T> & List_Append)
{
    Node<T> * tmp_node;
    List<T> * tmp;
    if (temp){                          // this ist eine temporaere Liste und kann veraendert
    werden
        tmp = this;
    }
    else {
        tmp = new List(this);           // this ist keine temporaere Liste -> Kopie erzeugen
        tmp->temp = true;               // Merker setzten, dass es sich um eine temporaere Liste
     handelt
    }
    if (List_Append._size) {            // anhaengen der uebergebenen Liste an tmp
        tmp_node = List_Append.head->next;
        while (tmp_node != List_Append.tail){
            tmp->InsertBack(tmp_node->key);
            tmp_node = tmp_node->next;
        }
    }
    if (List_Append.temp) delete & List_Append;     // wurde eine temporaere Liste uebergeben, dann
     wird diese geloescht
    return *tmp;
}

template <class T>
List<T> & List<T>::operator + (List<T> const * List_Append)
{
    Node<T> * tmp_node;
    List<T> * tmp;
    if (temp){                          // this ist eine temporaere Liste und kann veraendert
```

```
    werden
        tmp = this;
    }
    else {
        tmp = new List(this);                // this ist keine temporaere Liste -> Kopie erzeugen
        tmp->temp = true;                    // Merker setzten, dass es sich um eine temporaere Liste
     handelt
    }
    if (List_Append->_size) {                // anhaengen der uebergebenen Liste an tmp
        tmp_node = List_Append->head->next;
        while (tmp_node != List_Append->tail){
            tmp->InsertBack(tmp_node->key);
            tmp_node = tmp_node->next;
        }
    }
    if (List_Append->temp) delete List_Append;      // wurde eine temporaere Liste uebergeben,
    dann wird diese geloescht
    return *tmp;
}


template <class Tf>
std::ostream & operator << (std::ostream  & stream, const List<Tf> & Liste)
{
    stream << Liste._form.start;
    for (Node<Tf> * tmp = Liste.head->next; tmp != Liste.tail; tmp = tmp->next)
        stream << tmp->key << (tmp->next == Liste.tail ? Liste._form.ende : Liste._form.zwischen
    );
    if (Liste.temp) delete & Liste;          // wurde eine temporaere Liste uebergeben, dann wird
    diese geloescht
    return stream;
}

template <class Tf>
std::ostream & operator << (std::ostream  & stream, const List<Tf> * Liste)
{
    stream << Liste->_form.start;
    for (Node<Tf> * tmp = Liste->head->next; tmp != Liste->tail; tmp = tmp->next)
        stream << tmp->key << (tmp->next == Liste->tail ? Liste->_form.ende : Liste->_form.
    zwischen);
    if (Liste->temp) delete Liste;           // wurde eine temporaere Liste uebergeben, dann wird
    diese geloescht
    return stream;
}

#endif
```

Listing 6: List.h

Das Programm sollte jetzt noch immer lauffähig sein.
Jetzt bauen Sie beide Klassen zu Template Klassen um.

3. Der letzte Teil der Aufgabe besteht nun darin, die ursprüngliche Aufgabe zu realisieren. Kopieren Sie dazu wieder alle Datein vom 2. Projekt in ein neues Projekt außer die Datei main.cpp. Dafür fügen Sie die Datei tweet.h und main.cpp ein.

```
#ifndef _TWEET_H
#define _TWEET_H

#include <string>
#include <ctime>
#include <iomanip>

class tweet
{
private:
    std::string text;
    int hour, min, sec;
public:
    tweet();
    tweet(std::string text);
    ~tweet();
    friend std::ostream & operator << (std::ostream & stream, const tweet & _tweet);        //
     Ausgabeoperator ueberladen
};

tweet::tweet()
{
}

tweet::tweet(std::string text)
{
    time_t now = time(0);
    tm *ltm = localtime(&now);
    hour = ltm->tm_hour;
    min = ltm->tm_min;
    sec = ltm->tm_sec;
    this->text = text;
}

tweet::~tweet()
{
}
```

```
std::ostream & operator << (std::ostream  & stream, const tweet & _tweet)
{
    stream << std::setw(2) << std::setfill('0') << _tweet.hour << ":" <<_tweet.min << ":" <<
     _tweet.sec << " -> " << _tweet.text;
    return stream;
}

#endif
```

Listing 7: tweet.h

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include "List.h"
#include "tweet.h"
using namespace std;

int main(void)
{
    List<tweet> Tweets;
    string Text;

    Tweets.format("Tweet\n--->\n", "\n", "\n--->\n");
    while (true)
    {
        cout << "\nNachricht: ";
        getline(cin, Text);
        if (Text == "quit") break;
        Tweets.InsertFront(tweet(Text));

        cout << Tweets;
    }

    return 0;
}
```

Listing 8: main.cpp

2 Praktikum 2: Ringpuffer und Binärbaum

2.1 Aufgabenstellung

Herzlichen Glückwunsch. Sie haben die Stelle als Systemarchitekt bei „Data Fuse Inc.“, einem führenden Anbieter von BigData- und Swarm-Informationslösungen, erhalten. In der Abteilung „advanced distributed systems“ sind Sie nun für die Konzeption und Realisierung von experimentellen Lösung zuständig, die die Handhabung von großen, elastischen Datenmengen vereinfachen sollen. Zeigen Sie, dass Sie kreative Lösungen finden und Algorithmen in ein neues Zusammenspiel bringen können.

- Teil 1 - Backup-Rotation mittels Ringpuffer
- Teil 2 - Mikro-Data, Datenhaltung und Operationen auf Binärbäumen
- Teil 3 - Backup-Rotation mit Ringpuffer und Binärbäumen (Zusatzaufgabe)

2.2 Backup mittels Ringpuffer

Backups sind wichtig - jedoch nur solange sie aktuell und vollständig sind. Im industriellen Umfeld kann eine einzelne Sicherung schnell mehrere Terabyte umfassen und eine aufwändige Archivierung ist selten erforderlich (Buchhaltung und Forschung mal außen vor). Man bedient sich hier dem Modell der Ring-Sicherung, bei der nur ein begrenzter Horizont von Backupsätzen vorgehalten wird und ältere Sicherungen wieder gelöscht oder überschrieben werden - dies spart Speicher und Geld.
Schreiben Sie ein Programm zur Erstellung und Verwaltung von Backups mittels Ringpuffer. Erstellen Sie eine Hauptklasse Ringpuffer, die die Verwaltung der verknüpften Backups ermöglicht und die alle nötigen Operationen auf der Datenstruktur erledigt (z.B. Neue Elemente hinzufügen, Zeiger umbiegen, Inhalte ausgeben, Suchen, usw.). Der verknüpfte Ring soll aus insgesamt 6 Knoten (RingNode) bestehen, in die die Sicherungssätze fortlaufend gespeichert werden. Um die Alterung des Datensatzes zu simulieren/kennzeichnen (siehe Bild 1), besitzt jeder Node das Attribut OldAge (Aktuellste: ’0’, der Älteste ’5’). Mit der Speicherung eines neuen Nodes ersetzen Sie immer den Ältesten. Damit der spätere Ausbau funktioniert, verwenden Sie bitte die im Bild (1) vorgegebene Klasse und erweitern Sie sie nach Bedarf.
Der Benutzer soll über die Konsole folgende Möglichkeiten haben:

- Neuen Datensatz eingeben. Dieser besteht aus Daten für das Backup (SymbolicData:string) sowie einer Beschreibung (Description:string)
- Suchen nach Backup-Daten. Auf der Konsole sollen die Informationen OldAge, Beschreibungs- und Datentext des betreffenden Nodes ausgegeben werden. Sonst eine Fehlermeldung.
- Alle Backup-Informationen ausgeben. Liste aller Backups, Format siehe Beispiel

```
==================================
 OneRingToRuleThemAll v0.1, by Sauron Schmidt
==================================
1) Backup einfuegen
2) Backup suchen
3) Alle Backups ausgeben
?> 1                                // Beispiel: neuer Datensatz
+Neuen Datensatz einfuegen
Beschreibung ?>  erstes Backup
Daten ?> echtWichtig1
+Ihr Datensatz wurde hinzugefuegt.
[...]
?> 2                                // Beispiel: suche Datensatz
+Nach welchen Daten soll gesucht werden?
?> echtWichtig1
+ Gefunden in Backup: OldAge 0, Beschreibung: erstes Backup, Daten: echtWichtig1
[...]
?> 3                    // Beispiel: Ausgabe aller Backups nach weiteren Operationen
OldAge: 0, Descr: sechstes Backup, Data: 0118999
--------------------------
OldAge: 5, Descr: erstes Backup, Data: echtWichtig1
--------------------------
OldAge: 4, Descr: zweites Backup, Data: 456
--------------------------
OldAge: 3, Descr: drittes Backup, Data: 789
--------------------------
OldAge: 2, Descr: viertes Backup, Data: 007
--------------------------
OldAge: 1, Descr: fuenftes Backup, Data: 1337
--------------------------
```

Beantworten Sie folgende Fragen:

- Welche andere Anwendung von Ringpuffern oder ähnlichen Strukturen kennen Sie?
- Wäre eine Sicherung dieser Art in einem Rechenzentrum sinnvoll?
- Können wir als Techniker die Dauer der Aufbewahrung frei bestimmen oder gibt es gesetzliche Regelungen (Bitte keine Gesetzestexte lesen, nur kurz googeln)
- Nennen Sie 3 Punkte, die Sie im Rahmen des Backup-Szenarios verbessern würden.

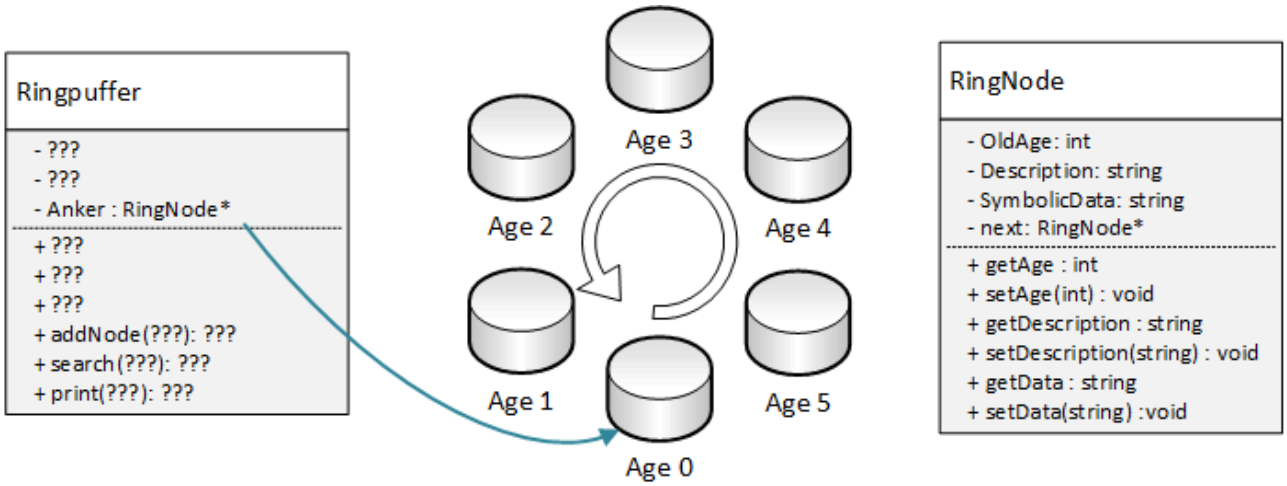


Abbildung 1: Datensicherung mittels Ringpuffer

2.2.1 Lösungshinweise

- Erstellen Sie die Klasse Ringpuffer als übergeordnete (keine Vererbung) Klasse für die Kontrolle über den eigentlichen Ring.
- Methoden der Klasse Ringpuffer sollen sich nur um das Handling der Datennodes kümmern und keine Nutzereingaben fürs Menü verarbeiten (z.B. Menü ausgeben, Menüauswahl einlesen, etc.). Schreiben Sie für die Darstellung des Menüs eigene Methoden oder realisieren Sie dies in der main() selbst.
- Der Datenring besteht aus 6 Nodes der Klasse RingNode und ist von außen nicht direkt zu erreichen, sondern nur über die Methoden der Ringpuffer-Klasse.
- Nutzen Sie Ihr Wissen über die verkettet Liste, da sich die Datenstrukturen stark ähneln.
- Achten Sie darauf, dass Sie die Reihenfolge der Operationen nach einer Nutzereingabe richtig vornehmen.
- Sie müssen die OldAge-Informationen der Nodes aktualisieren und den neuen Node richtig positionieren.
- Denken Sie über die Folgen einer Rotation nach, wenn Sie die Referenzen umstecken.
- Erweitern Sie die Klassen mit Attributen und Methoden nur, wenn es wirklich erforderlich ist.

2.3 Datenhaltung und Operationen auf Binärbäumen

Ihnen liegt ein großer Datensatz aus der letzten Zielgruppenanalyse vor. Um diesen mit Ihrem Mikro-Data System nutzen zu können, müssen Sie ihn zunächst in einer entsprechenden Datenstruktur aufbereiten. Sie haben sich für die einfache Variante eines Binär-Bäums entschieden, der die Platzierungsentscheidung der Nodes anhand eines Schlüsselwertes trifft.
Entwickeln Sie daher einen Binärbaum „Tree“, dessen „TreeNodes“ der Vorgabe aus Bild 2 entsprechen. Der Tree ist dabei die übergeordnete Klasse (keine Vererbung), die die Organisation und Operation auf den TreeNodes steuert. Die TreeNodes sind nur vom Tree ansprechbar und besitzen die angegebenen Datenattribute (Bild 2) sowie Referenzen auf den links/rechts folgenden TreeNode oder Nullpointer. Programmieren Sie für die Attribute Setter/Getter-Methoden wenn es sinnvoll erscheint. Die Position des TreeNodes im Tree wird aus dem Alter, der PLZ und dem Einkommen der Person errechnet und als Schlüsselattribut „NodePosID“ gespeichert.

$$Alter(int) + PLZ(int) + Einkommen(double) = Positionsindikator(int)$$

Beachten Sie, dass die NodeID eine fortlaufende Seriennummer ist und nicht mit der NodePosID zu verwechseln ist. Sie müssen den Baum nach Operationen **nicht** ausbalancieren, aber nach z.B. Löschvorgängen wieder korrekt funktionsfähig machen. Beachten Sie hier die Lösungshinweise.

- Die NodeID eignet sich nicht als Positionsindikator. Warum? Wie würde sich das auf den Baum auswirken?
- Was würde passieren, wenn die NodePosID mehrfach vergeben wird?
- Welche Interpretation lassen die sortierten Daten später zu? Was könnten Sie aus den Zusammenhängen schließen?
- Kennen Sie eine Datenstruktur, die sich für eine solche Aufgabe besser eignen würde?

Der Benutzer soll über ein Menü mit der Datenstruktur arbeiten können und folgende Möglichkeiten haben:

- Hinzufügen neuer Datensätze als Benutzereingabe
- Hinzufügen neuer Datensätze aus einer CSV Datei (Beachten Sie die Lösungshinweise)
- Löschen eines vorhandenen Datensatzes anhand der PositionsID.
- Suchen eines Datensatzes anhand des Personennamens.
- Anzeige des vollständigen Baums nach ’Preorder’.

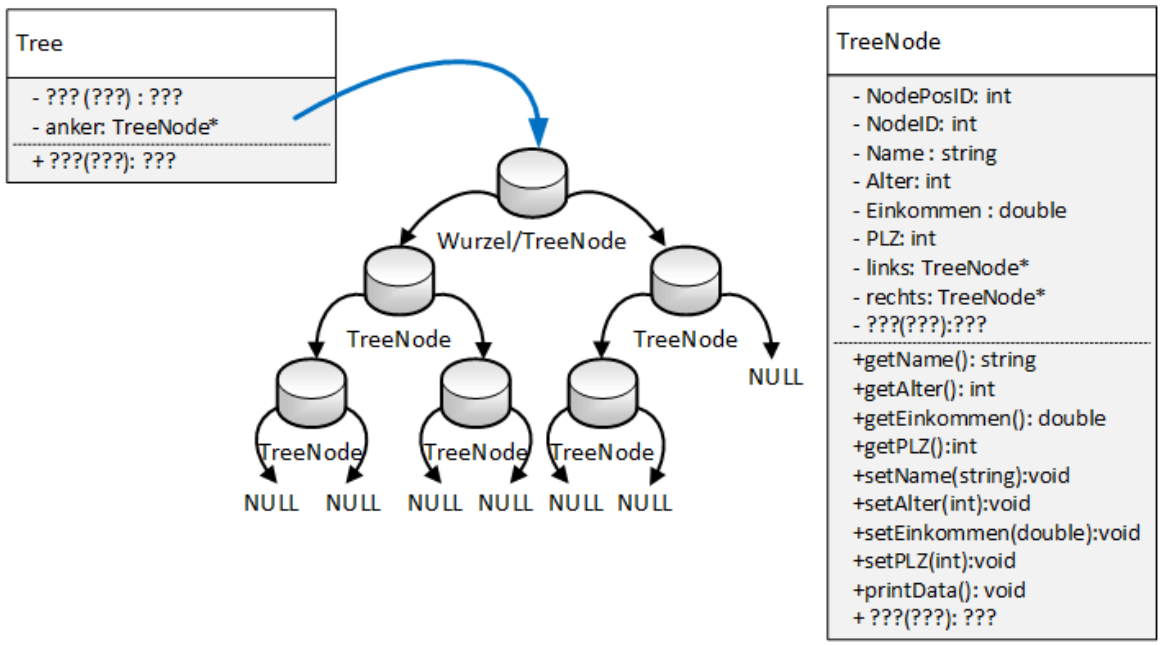


Abbildung 2: Beispiel, Aufbau der Baumstruktur und Klassen

```
==================================
Person Analyzer v19.84, by George Orwell
==================================
1) Datensatz einfuegen, manuell
2) Datensatz einfuegen, CSV Datei
3) Datensatz löschen
4) Suchen
5) Datenstruktur anzeigen
?> 1
+ Bitte geben Sie die den Datensatz ein
Name ?> Mustermann
Alter ?> 1
Einkommen ?> 1000.00
PLZ ?> 1
+ Ihr Datensatz wurde eingefögt
[...]                                // ... mehrere Datensötze hinzugefögt
5) Datenstruktur anzeigen
?> 4
ID | Name       | Alter | Einkommen |  PLZ   | Pos
---+------------+-------+-----------+--------+-------
0  |  Mustermann|      1|       1000|       1|1002
3  |        Hans|      1|        500|       1|502
5  |     Schmitz|      1|        400|       2|403
```

```
4  |     Schmitt|      1|        500|      2|503
1  |      Ritter|      1|       2000|      1|2002
2  |      Kaiser|      1|       3000|      1|3002
[...]
?> 4                                          // Datensatz suchen
+ Bitte geben Sie den zu suchenden Datensatz an
Name ?> Schmitt
+ Fundstellen:
NodeID: 4, Name: Schmitt, Alter: 1, Einkommen: 500, PLZ: 2, PosID: 503
[...]
?> 3                                          // Datensatz löschen
+ Bitte geben Sie den zu löschenden Datensatz an
PosID: 502
+ Datensatz wurde gelöscht.
[...]
?> 5                              // zum Vergleich nochmal Struktur angezeigt
ID | Name       | Alter | Einkommen |  PLZ  | Pos
---+------------+-------+-----------+-------+-------
0  |  Mustermann|      1|       1000|      1|1002
4  |     Schmitt|      1|        500|      2|503
5  |     Schmitz|      1|        400|      2|403
1  |      Ritter|      1|       2000|      1|2002
2  |      Kaiser|      1|       3000|      1|3002
```

2.3.1 Lösungshinweise

2.3.2 Klassen und Attribute

- Der Baum „Tree“ und die Knoten/Blätter „TreeNodes“ sind je eigenständige Klassen, die nicht untereinander vererben.
- Die NodeID ist eine fortlaufende Seriennummer für jeden Node, die beim Anlegen eines neuen Datensatzes einmalig vergeben wird und die Einfüge-Reihenfolge nachvollziehbar macht. Die NodePosID ist damit nicht zu verwechseln, denn diese wird aus dem Alter, dem Einkommen und der PLZ des Datensatzes errechnet und dann für die Platzierung im Baum genutzt.
- Nutzen Sie die Datenkapselung und schützen Sie alle Attribute vor Zugriff. Erstellen Sie nur dann Setter/Getter, wenn es sinnvoll ist.
- Alle TreeNodes haben zwei Zeiger vom Typ TreeNode, die auf die nachfolgenden rechten/linken TreeNodes verweisen. Gibt es keine Nachfolger, so müssen die Referenzen auf Null oder einen festgelegten Nullpointer verweisen.

2.3.3 Tree und Tree-Operationen

- Das Einfügen eines neuen Nodes erfolgt anhand der errechneten PositionsID. Laufen Sie ab Wurzel die bestehenden Nodes ab und vergleichen Sie die PositionsID, in deren Abhängigkeit dann in den linken oder rechten Teilbaum gewechselt wird. Sie können davon ausgehen, dass der Benutzer nur gültige Werte eingibt, die nicht zu einer Mehrfachvergabe der PositionsID führen.
- Bei der Suchfunktion soll nach dem Namen einer Person gesucht werden und die Position sowie die vorhandenen Daten ausgegeben werden. Beachten Sie hierbei, dass „Name“ kein Schlüsselelement ist und es mehrere Datensätze mit dem Personennamen z.B. „Schmitt“ geben kann. Sie müssen alle passenden Einträge finden und ausgeben. Überlegen Sie, ob hier eine rekursive oder iterativer Vorgehensweise besser ist.
- Löschen eines vorhandenen Datensatzes ist die anspruchsvollste Operation im Binärbaum. Haben Sie die Position des TreeNodes im Tree ausgemacht, müssen Sie auf folgende 4 Fälle richtig reagieren. Der zu löschende TreeNode...

  1. ... ist die Wurzel.
  2. ... hat keine Nachfolger.
  3. ... hat nur einen Nachfolger (rechts oder links).
  4. ... hat zwei Nachfolger.

  Denken Sie daran, dass Sie bei einer Löschoperation auch immer die Referenz des Vorgängers auf die neue Situation umstellen müssen.
- Die erwartete Funktionalität der Methoden aus Bild 2 ergibt sich aus der Benennung. Erweitern Sie die Klassen um eigene Methoden und Attribute wenn es sinnvoll ist.
- Sie müssen den Baum nicht ausbalancieren.

2.3.4 Eingabe der Daten

- Manuelle Eingabe - Wie im Schaubild oben zu erkennen soll der Benutzer über die Konsole einen neuen Datensatz anlegen können. Dabei werden die benötigten Positionen der Reihe nach als Benutzereingabe aufgenommen.
- CSV-Datei - Um einen schnelleren Import zu ermöglichen, soll der Benutzer auch eine CSV Datei einlesen können.
  - Die CSV Datei liegt im gleichen Verzeichnis wie das Programm und heißt immer “ExportZielanalyse.csv“. Sie müssen dem Benutzer keine andere Datei zur Auswahl geben oder eine Eingabe für den Dateinamen realisieren. Fragen Sie lediglich ob die Datei mit dem Namen wirklich importiert werden soll.

- Jede Zeile der Datei stellt einen Datensatz dar, der seine Elemente mittels Semikolon trennt. (Hilfestellung: `https://www.c-plusplus.net/forum/281529-full` und `https://de.wikipedia.org/wiki/CSV_(Dateiformat)`)
- Beachten Sie, dass Sie die gelesenen Werte in den richtigen Zieldatentyp übertragen müssen.
- Die Reihenfolge der Spalten in der CSV Datei, entspricht der manuellen Eingabe (siehe Beispiel oben).
- Der CSV Import darf die bereits vorhandenen, manuell eingetragenen, Datensätze nicht überschreiben, sondern nur den Tree damit erweitern.

2.3.5 Ausgaben und Benutzerinteraktion

- Übernehmen Sie das Menü aus dem Beispiel oben.
- Fehlerhafte Eingaben im Menü müssen abgefangen werden.
- Bei fehlgeschlagenen Operationen wird eine Fehlermeldung erwartet.
- Die Ausgabe des gesamten Baums (alle TreeNode Daten, siehe Beispiel) soll nach der 'Pre-Order' Reihenfolge erfolgen.
- Formatieren Sie die Ausgaben sinnvoll (siehe Beispiel).

2.4 Backup in hochverfügbaren Datenstrukturen (Zusatzaufgabe)

Situation: Das von Ihnen geschaffene, fragile Mass-Data-System (MDS) ist das Herz, die Seele und die Einnahmequelle des Unternehmens. Laut internem Service-Level-Agreement, muss das System extrem hochverfügbar sein und soll trotzdem in die Rotation der Datensicherung aufgenommen werden. Durch die geforderte Verfügbarkeit und der hohen Datenänderungsrate, ist ein "cold" Backup (abschalten und sichern) oder ein "hot/online" Backup (Stück für Stück sichern, wenn TreeNodes ohne Zugriff sind) nicht möglich. Sie müssen daher einen Snapshot (alles sofort) realisieren.

- Überlegen Sie: Warum ist keine der anderen Sicherungsarten passend? Warum passt das hot/online Backup auch nicht?

Kombinieren Sie jetzt die beiden oberen Teilaufgaben zu einem Gesamtsystem, welches den Binärbaum auf Befehl in einem Ringpuffer „backupt". Die Funktionalität des Binärbaums soll wie in Teilaufgabe 2 erhalten bleiben. Die Abbildung 3 zeigt den gedachten Aufbau. Vereinen Sie die Menüstruktur der bisherigen beiden Teilaufgaben sinnvoll. Beispiel:

```
1  ==================================
2  HPDS v0.1, by Speedy Gonzales
3  ==================================
4  Backupsteuerung:
5  1) Backup einfuegen
6  2) Backup suchen
7  3) Alle Backups ausgeben
8  ----------------------------
9  Aktueller Baum:
10 4) Datensatz einfuegen
11 5) Datensatz löschen
12 6) Suchen
13 7) Datenstruktur anzeigen
14 ?>
```

2.4.1 Lösungshinweise

2.4.2 Anpassen des Ringpuffers

- Nutzen Sie jetzt das alte RingNode „SymbolicData" Attribut für die Referenz auf den zu speichernden Tree. Somit entfällt dieses Attribut bei der Nutzereingabe sowie Ausgabe.
- Die Suche nach Daten soll sich über alle Trees erstrecken. Wie bei den Trees, ist das zu suchende Attribut jetzt der Personenname. Geben Sie das Suchergebnis aus allen Trees, unter Angabe des Fundortes in welchem Backup, auf der Konsole aus.
- Passen Sie die Ringklassen für Operationen und Parameter vom Typ „Tree" bzw. „Tree*" an.

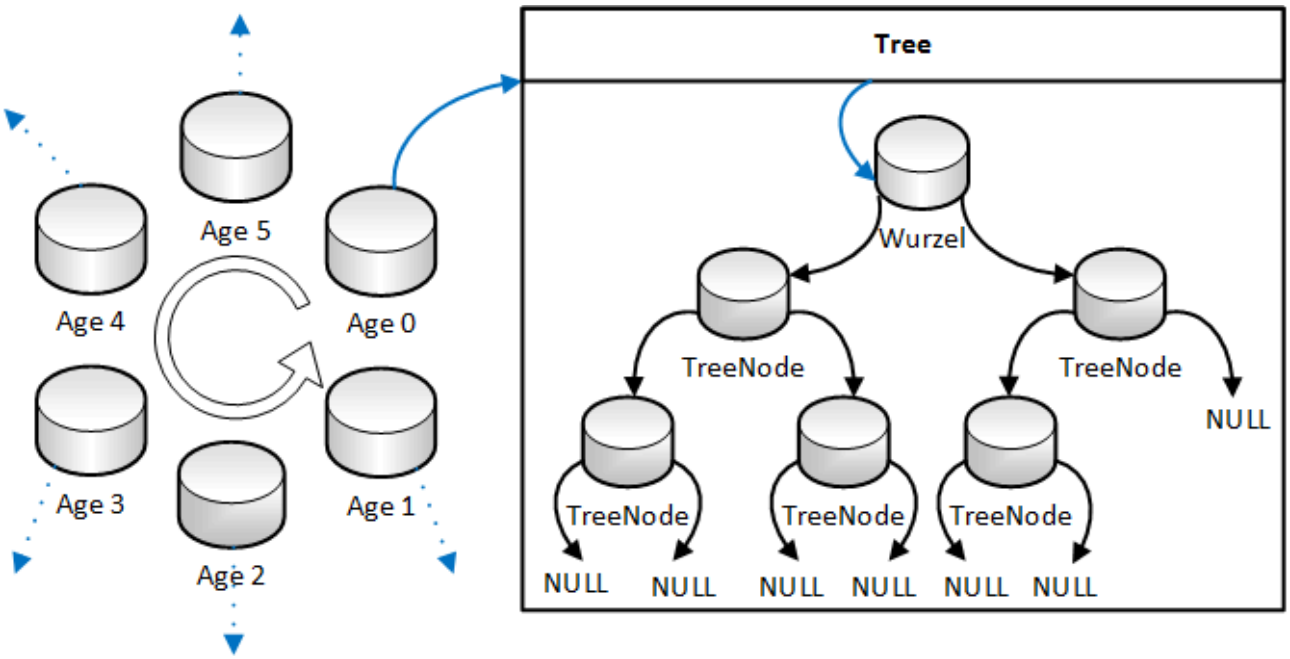


Abbildung 3: Im Ring gesicherter Snapshot der Baumstruktur

2.4.3 Sicherung/Kopie des Trees

- Beachten Sie, dass wenn ein Objekt Referenzen auf Andere als Attribute besitzt, diese nicht einfach kopieren/zuweisen können, da der Verweis auf die Speicherstelle bleibt. Ändern Sie Dinge im aktuellen Baum, hätte das Auswirkung auf die Bäume im Backup. Sie müssen die Bäume jeweils in neuen Speicher überführen (Stichwort: CopyConstructor).
- Überlegen Sie, ob die Überführung/Kopie der DatenNodes mit einer rekursiven oder iterativen Methode bewerkstelligt werden soll.

3 Praktikum 3: Sortierverfahren / Open Multi-Processing (OpenMP)

Ihre Vorgesetzten bei der Firma "Data Fuse Inc." sind begeistert von Ihren Fähigkeiten! Da die Verarbeitungsgeschwindigkeit der enormen Datenmengen weiter optimiert werden muss wurden Sie beauftragt, ein Framework zur Messung von Laufzeiten zu entwickeln. Mit Hilfe dieses Programms sollen Sie anschließend die Ausführungsdauer verschiedener Algorithmen sowie die Laufzeit von Matrixmultiplikationen quadratischer Matrizen in Abhängigkeit einer Problemgröße n untersuchen und auswerten. Die Ergebnisse sollen dann im Praktikum präsentiert und diskutiert werden. Mit Hilfe von OpenMP sollen die Zeiten gemessen und die Matrixmultiplikationen parallelisiert werden.

3.1 Aufgabenstellung

1. Erstellen Sie dazu eine Header-Datei *MyAlgorithms.h* und implementieren Sie die folgenden Sortierverfahren in der zugehörigen cpp-Datei *MyAlgorithms.cpp* sowie in einem eigenen Namespace:

   - Heapsort
   - Mergesort
   - Quicksort
   - Shellsort mit der **Hibbard Folge** ($H_i = 2H_{i-1} + 1$)

   Testen Sie die Sortierverfahren zunächst mit folgenden Zahlen: 98 44 30 22 64 63 11 23 8 18. Messen Sie anschließend für jedes Sortierverfahren die Laufzeit, die zur Sortierung eines Arrays der Größe n benötigt wird! Zur Durchführung der Messung erzeugen Sie Zufallszahlen (Integer) und lassen die Problemgröße wie folgt wachsen: $n = 1000 : 1000 : 1000000$

2. Erweitern Sie ihre Algorithmensammlung indem Sie die schulbuchmäßige Matrixmultiplikation für quadratische Matrizen implementieren (Testlauf: siehe (3.5)). Beim letzten Plausch in der Kaffeeküche entbrannte eine heiße Diskussion, ob hier eine spalten- oder zweilenweise Speicherung der Matrizen effizienter sei. Ihre Vorgesetzten waren sich uneinig, implementieren Sie die Multiplikation in beiden Varianten und messen Sie die Laufzeit! Welche Art der Speicherung ist günstiger für die Verarbeitung? Die Matritzen sollen dabei mit Zufallszahlen (Double) gefüllt werden und die Problemgröße soll wie folgt wachsen: $n = 2 : 1 : 799$, danach $n = 800 : 11 : 2000$.

3. Parallelisieren Sie die Berechnung der Matrizen durch hinzufügen der passenden OpenMP-Präprozessor-Direktive (siehe Abb. 13) und vergleichen Sie die Ergebnisse mit der "single-thread" Variante!

4. Stellen Sie ihre Messergebnisse unter Zuhilfennahme von MATLAB, Octave oder GNUPLOT grafisch dar! Entsprechen die Messergebnisse den Erwartungen (z.B. bzgl. O-Notation)? Achten Sie bei den Plots auf aussagekräftige Achsenbeschriftungen und eine vernünftige Legende!

5. Beachten Sie unbedingt die Lösungshinweise (3.2) sowie die Testläufe (3.3) und **planen Sie genügend Zeit für die Messungen ein**!

3.2 Lösungshinweise

3.2.1 main.cpp

Die Laufzeit eines Programms / Algorithmus exakt zu messen ist grundsätzlich nicht trivial. Im Rahmen dieser Aufgabe begnügen wir uns mit Bordmitteln aus der OpenMP-Library. Benutzen Sie die folgende Struktur als Grundgerüst für ihre Messungen und erweitern Sie sie nach Ihren Bedürfnissen:

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <vector>
#include <omp.h>
#include "MyAlgorithms.h"

using namespace std;

int main(int argc, char** argv) {
  ofstream file;
  file.open("example.txt");
  double dtime;
  int n_start = 1000;
  int n_step = 1000;
  int n_end = 1000000;

  for (int n = n_start; n<n_end; n+=n_step) {
    cout << "n: " << n << endl;
    /**********************/
    // init data here //
    /**********************/
    dtime = omp_get_wtime();
    /**********************/
    // run algorithm here //
    /**********************/
    dtime = omp_get_wtime() - dtime;
    file << n << "\t" << scientific << setprecision(10) << dtime << endl;
  }
  file.close();
}
```

Listing 9: main.cpp

3.2.2 MyAlgorithms.h

Implementieren Sie die folgende Header-Datei in der zugehörigen cpp-Datei *MyAlgorithms.cpp*. Beachten Sie, dass für ihre eigenen Funktionen ein eigener Namespace verwendet werden soll.

```
#ifndef _MYALGORITHMS_H_
#define _MYALGORITHMS_H_
#include <vector>
using namespace std;

namespace MyAlgorithms {

  //Heapsort
  void max_heapify(vector<int> &a, int i, int  n);
  void HeapSort(vector<int> &a, int n);

  //MergeSort
  void Merge(vector<int> &a, vector<int> &b,int low, int pivot, int high);
  void MergeSort(vector<int> &a, vector<int> &b, int low, int high);

  //Quicksort
  void QuickSort(vector<int> &arr, int left,  int  right);

  //Shellsort

  //Matrix Multiplikation
  void MatrixMul_ColMajor(vector<double> &A,
                vector<double> &B,
                vector<double> &C,
                int  n);
  void MatrixMul_RowMajor(vector<double> &A,
                vector<double> &B,
                vector<double> &C,
                int  n);
} //end namespace

#endif //_MYALGORITHMS_H_
```

Listing 10: MyAlgorithms.h

3.2.3 Matrixmultiplikation: Aufbau der Matrizen und Allokation im Speicher

Gegeben sei eine quadratische Matrix $A \in \mathbb{R}^{n \times n}$ mit n Zeilen und n Spalten. Ihre Dimension in y-Richtung, bzw. die Anzahl der Zeilen, wird als "leading dimension y (ldy)" bezeichnet, die Dimension in x-Richtung, bzw die Anzahl der Spalten, als "leading dimension x (ldx)". Da wir nur quadratische Matrizen betrachten gilt $ldx = ldy = ld = n$. Zusätzlich „denken" wir in C/C++ und indizieren die Matrixelemente ab 0 (Abb. 4).

$$A = \underbrace{\left.\begin{bmatrix} a_{0,0} & \dots & a_{0,n-1} \\ \vdots & \vdots & \vdots \\ a_{n-1,0} & \dots & a_{n-1,n-1} \end{bmatrix}\right\} n,\ ldy}_{n,ldx}$$

Abbildung 4: Quadratische Matrix der Größe n

Um diese zweidimensionale Datenstruktur möglichst effizient in C/C++ speichern und verarbeiten zu können sollten die Daten unbedingt hintereinander und zusammenhängend im Speicher abgelegt werden. Die zweidimensionale $(n \times n)$-Matrix soll deswegen auf ein eindimensionales Array L mit $n * n$ Elementen abgebildet werden, wobei entweder alle Spalten oder alle Zeilen der Matrix der Reihe nach aneinander gehängt werden können (Abb. 5):

$$L = \begin{bmatrix} a_{0,0} \\ \vdots \\ a_{n-1,0} \\ \vdots \\ \vdots \\ a_{0,n-1} \\ \vdots \\ a_{n-1,n-1} \end{bmatrix}$$

$$A_{i,j} \rightarrow L[i + j * ld]$$

(a) spaltenorientierte Speicherung

$$L = \begin{bmatrix} a_{0,0} \\ \vdots \\ a_{0,n-1} \\ \vdots \\ \vdots \\ a_{n-1,0} \\ \vdots \\ a_{n-1,n-1} \end{bmatrix}$$

$$A_{i,j} \rightarrow L[i * ld + j]$$

(b) zeilenorienterte Speicherung

Abbildung 5: eindimensionales Array L

Mit diesem Array L soll im Programm gearbeitet werden. Dazu muss die zweidimensionale Indizierung der Matrix $A \in \mathbb{R}^{n \times n}$ in die passende eindimensionale Indizierung des Arrays L übersetzt werden. Ein Indexpaar (i, j) wird dabei wie in Abb. 5 übersetzt. Nutzen Sie die C++-Klasse ***vector*** zur Realisierung des linearen Arrays. Machen Sie sich mit den Methoden dieser Klasse vertraut (*size()*, *resize()*, etc ...) !

3.2.4 Matrixmultiplikation : Formel / Codebeispiel

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} * b_{kj} \quad mit\ i = 0\ ...\ n-1,\ j = 0\ ...\ n-1 \tag{1}$$

$$\underbrace{\begin{bmatrix} c_{0,0} & c_{0,1} & c_{0,2} & c_{0,3} \\ c_{1,0} & c_{1,1} & c_{1,2} & c_{1,3} \\ c_{2,0} & c_{2,1} & c_{2,2} & c_{2,3} \\ c_{3,0} & c_{3,1} & c_{3,2} & c_{3,3} \end{bmatrix}}_{\text{C}} = \underbrace{\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix}}_{\text{A}} \times \underbrace{\begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{bmatrix}}_{\text{B}}$$

Abbildung 6: Matrixmultiplikation $C = A \times B$ für $n = 4$

```
void MatrixMul_ColMajor(vector<double> &A, vector<double> &B, vector<double> &C,
int n) {
  //lda, ldb, ldc are leading dimensions of matrices A,B and C
  int lda = n;
  int ldb = n;
  int ldc = n;
  double s = 0.0;

  for (int i=0; i<n ; i++) {
    for (int j=0; j<n ; j++) {
      s = 0.0;
      for (int k=0; k<n; k++) {
    s = s + A[i+k*lda] * B[k+j*ldb];
      }
      C[i+j*ldc]=s;
    }
  }
}

void MatrixMul_RowMajor(vector<double> &A, vector<double> &B, vector<double> &C,
int n) {
  //lda, ldb, ldc are leading dimensions of matrices A,B and C
  int lda = n;
  int ldb = n;
  int ldc = n;
  double s = 0.0;
  /*****************************************/
  // implement row major calculation here//
  /*****************************************/
}
```

Listing 11: Beispiel für Matrixmultiplikation in C++

3.2.5 OpenMP Beispiele

```
#include <iostream>
#include <omp.h>

int main(int, char*[]) {

    int num_procs = omp_get_num_procs();
    omp_set_num_threads(num_procs);

    int threads = 0;
    int id = 0;

    #pragma omp parallel private (id)
    {
     threads = omp_get_num_threads();
     id = omp_get_thread_num();
     std::cout << "hello world from thread: " << id << " out of ";
     std::cout << threads << "\n";
    }
    return 0;
}
```

Listing 12: OpenMP Hello World

```
#include <iostream>
#include <omp.h>

int main(int, char*[]) {

    int num_procs = omp_get_num_procs();
    omp_set_num_threads(num_procs);

    int arraysize=1000000000;
    int* myarray = new int[arraysize];

    int i;
    #pragma omp parallel for
    for (i=0;i<arraysize;i++) {
        myarray[i] = i;
    }

    return 0;
}
```

Listing 13: OpenMP Beispiel zur Parallelisierung einer for-Schleife

3.3 Testläufe

3.4 Sortierverfahren

Testdaten Sortieralgorithmen: 98 44 30 22 64 63 11 23 8 18

- Beispielausgabe Heap Sort:

```
Heap Sort:
Durchlauf 0: 98  44  30  22  64  63  11  23  8  18
percDown(63) Durchlauf 5: 98  44  30  22  64  63  11  23  8  18
percDown(64) Durchlauf 4: 98  44  30  22  64  63  11  23  8  18
percDown(22) Durchlauf 3: 98  44  30  23  64  63  11  22  8  18
percDown(30) Durchlauf 2: 98  44  63  23  64  30  11  22  8  18
percDown(44) Durchlauf 1: 98  64  63  23  44  30  11  22  8  18
percDown(98) Durchlauf 0: 98  64  63  23  44  30  11  22  8  18
Heap Sort:
Durchlauf 0: 98  64  63  23  44  30  11  22  8  18
percDown(18) Durchlauf 9: 64  44  63  23  18  30  11  22  8  98
percDown(8) Durchlauf 8: 63  44  30  23  18  8  11  22  64  98
percDown(22) Durchlauf 7: 44  23  30  22  18  8  11  63  64  98
percDown(11) Durchlauf 6: 30  23  11  22  18  8  44  63  64  98
percDown(8) Durchlauf 5: 23  22  11  8  18  30  44  63  64  98
percDown(18) Durchlauf 4: 22  18  11  8  23  30  44  63  64  98
percDown(8) Durchlauf 3: 18  8  11  22  23  30  44  63  64  98
percDown(11) Durchlauf 2: 11  8  18  22  23  30  44  63  64  98
percDown(8) Durchlauf 1: 8  11  18  22  23  30  44  63  64  98
8  11  18  22  23  30  44  63  64  98
Finished in    0.062 sec
```

- Beispielausgabe Mergesort

```
MergeSort(0,9)
MergeSort(0,4)
MergeSort(0,2)
MergeSort(0,1)
MergeSort(0,0)
MergeSort(1,1)
Merge(0,0,1): 44 98 30 22 64 63 11 23 8 18
MergeSort(2,2)
Merge(0,1,2): 30 44 98 22 64 63 11 23 8 18
MergeSort(3,4)
MergeSort(3,3)
MergeSort(4,4)
Merge(3,3,4): 30 44 98 22 64 63 11 23 8 18
```

```
Merge(0,2,4): 22 30 44 64 98 63 11 23 8 18
MergeSort(5,9)
MergeSort(5,7)
MergeSort(5,6)
MergeSort(5,5)
MergeSort(6,6)
Merge(5,5,6): 22 30 44 64 98 11 63 23 8 18
MergeSort(7,7)
Merge(5,6,7): 22 30 44 64 98 11 23 63 8 18
MergeSort(8,9)
MergeSort(8,8)
MergeSort(9,9)
Merge(8,8,9): 22 30 44 64 98 11 23 63 8 18
Merge(5,7,9): 22 30 44 64 98 8 11 18 23 63
Merge(0,4,9): 8 11 18 22 23 30 44 63 64 98
```

- Beispielausgabe Quicksort

```
QuickSort(a,0,9)
pivot = a[4] = 64
Result: 18 44 30 22 8 63 11 23 64 98
QuickSort(a,0,7)
pivot = a[3] = 22
Result: 18 11 8 22 30 63 44 23 64 98
QuickSort(a,0,2)
pivot = a[1] = 11
Result: 8 11 18 22 30 63 44 23 64 98
QuickSort(a,4,7)
pivot = a[5] = 63
Result: 8 11 18 22 30 23 44 63 64 98
QuickSort(a,4,6)
pivot = a[5] = 23
Result: 8 11 18 22 23 30 44 63 64 98
QuickSort(a,5,6)
pivot = a[5] = 30
Result: 8 11 18 22 23 30 44 63 64 98
QuickSort(a,8,9)
pivot = a[8] = 64
Result: 8 11 18 22 23 30 44 63 64 98
```

- Beispielausgabe Shellsort

```
Shell Sort (H = 2H+1)
8  11  18  22  23  30  44  63  64  98
Finished in    10 sec
```

3.5 Matrixmultiplikation

Testdaten Matrixmultiplikation:

$$\underbrace{\begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}}_{\text{C}} = \underbrace{\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}}_{\text{A}} \times \underbrace{\begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}}_{\text{B}}$$

- Beispielausgabe Matrixmultiplikation

```
spaltenweise
******A******
1
3
2
4
******B******
5
7
6
8
******C******
19
43
22
50

zeilenweise
******A******
1
2
3
4
******B******
5
6
7
8
******C******
19
22
43
50
```

3.6 Compiler-Einstellungen

3.6.1 Linux g++

Aktivierung von OpenMP unter Ubuntu durch Compilerflag **-fopenmp** und linken gegen OpenMP Library **-lgomp**. Beispielaufruf für g++:

```
g++ main.cpp MyAlgorithms.h MyAlrogithms.cpp −o praktikum3 −fopenmp −lgomp
```

3.6.2 Windows Visual Studio

Aktivierung von OpenMP unter Visual Studio:
https://msdn.microsoft.com/de-de/library/fw509c3b.aspx

```
öffnen Sie das Dialogfeld Eigenschaftenseiten des Projekts.
Erweitern Sie den Knoten Konfigurationseigenschaften.
Erweitern Sie den Knoten C/C++.
Wöhlen Sie die Eigenschaftenseite Sprache aus.
öndern Sie die Eigenschaft OpenMP−Unterstötzung.
```

3.6.3 Windows DevC++

Aktivierung von OpenMP unter DevC++:

```
Tools −> Compiler Options
Add the following commands when calling the compiler: −fopenmp
Add the following commands when calling the linker: −lgomp
```

3.6.4 Beispiele zum Plotten mit MATLAB / GNUPLOT

- MATLAB
  Erzeugen Sie im selben Ordner, indem sich Ihre Messungen befinden, eine M-Skript-Datei mit einem beliebigen Namen, z.B. *make_plots.m*:

```
clear ; clc ; close  all ;

fid =fopen('quicksort.txt');
data=textscan(fid,'%d %f');
fclose ( fid ) ;
x=data{1};
quicksort_y=data{2};

fid =fopen('mergesort.txt');
data=textscan(fid,'%d %f')
fclose ( fid ) ;
mergesort_y=data{2};

fid =fopen('heapsort.txt');
data=textscan(fid,'%d %f');
fclose ( fid ) ;
heapsort_y=data{2};

fid =fopen('shellsort.txt');
data=textscan(fid,'%d %f');
fclose ( fid ) ;
shellsort_y =data{2};



figure ;
title ('sorting algorithms');
xlabel('n [-]');
ylabel('t [s]');
hold on;
plot(x,quicksort_y);
plot(x,mergesort_y);
plot(x,heapsort_y);
plot(x, shellsort_y );
legend('quicksort','mergesort','heapsort','shellsort','Location','northwest');
hold  off ;
```

  Führen Sie das Skript anschließend aus:

```
>> make_plots
```

- GNUPLOT
  Erzeugen Sie im selben Ordner, indem sich Ihre Messungen befinden, eine Datei mit einem beliebigen Namen, z.B. *plots.gnu*:

```
reset
set autoscale x
set autoscale y
set xlabel "n [-]"
set ylabel "t [s]"
set key top left

plot \
"quicksort.txt" with linespoints title 'Quicksort',\
"mergesort.txt" with linespoints title 'Mergesort',\
"shellsort.txt" with linespoints title 'Shellsort',\
"heapsort.txt" with linespoints title 'Heapsort',\
```

  Starten Sie nun Gnuplot, wechseln Sie in das korrekte Verzeichnis, und fuehren Sie das Skript wie folgt aus:

```
$ cd 'pfad-zum-gnuplot-skript'
$ load "plots.gnu"
```

  Weiterführende Befehle zu GNUPLOT findet man z.B. hier:
  http://gnuplot.sourceforge.net/docs_4.0/gpcard.pdf

4 Praktikum 4 : Graphen

In dieser Aufgabe ist erneut Ihre Kompetenz als Systemarchitekt bei „Data Fuse“ gefragt. Die verschiedenen Standorte sollen mit neuen Routern verbunden werden und Sie sollen testen, ob die geplanten Router und deren Verbindungen ausreichen um alle Standorte zu verbinden. Ein weitere Test wird sein, dass die Entfernungen gemessen werden sollen.

In Abbildung 7 sehen Die ein Beispiel, wie ein solcher Graph aufgebaut sein könnte, die Knoten stellen die Router an den Standorten dar, die Kanten die Verbindungen mit einem Gewicht als fiktive Entfernung. Hier lässt sich schnell erkennen, dass alle Standorte verbunden sind, mit der minimalen Entfernung wird es auch hier auf den ersten Blick schon schwieriger.

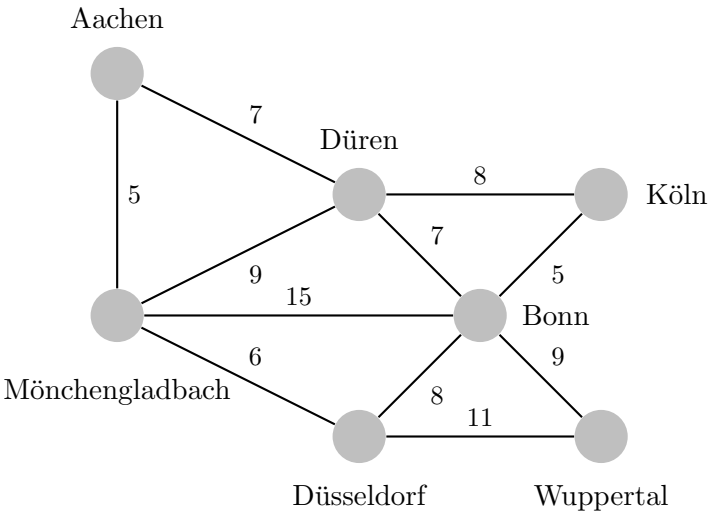


Abbildung 7: Beispiel Unternehmens Standorte mit Verbindungen

4.1 Aufgabenstellung

1. Beschäftigen Sie sich mit der Tiefen- und Breitensuche und seien Sie in der Lage den Unterschied zu erklären.

   Implementieren Sie anschließend beide Suchen. Nutzen Sie dazu für eine Suche die iterative Form und für die andere die rekursive Form. Seien Sie auch hier in der Lage die Unterschiede zu erklären. Sie können so überprüfen, ob alle Standorte erreicht werden können. Nutzen Sie dazu das „besucht“ flag der Knotenklasse.

   Als Beispielgraph dient die Datei „graph.txt“ mit dem Graph als Kantenliste. Die erste Zeile gibt die Knotenzahl m an, alle folgenden Zeilen die Kanten von $i \rightarrow j$, wobei die Knoten mit $0...(m-1)$ nummeriert sind. Der Graph ist ungerichtet.

2. Beschäftigen Sie sich mit der Thematik des minimalen Spannbaums. Kennen Sie die Bedeutung der Algorithmen nach Prim und Kruskal im Bezug auf den minimalen Spannbaum. Seien Sie in der Lage die beiden Algorithmen zu erklären und die Unterschieden zu verdeutlichen.

Implementieren Sie anschließend die Algorithmen nach Prim und Kruskal. Mit diesen können Sie einen Minimalen Spannbaum auf zwei verschiedene Weisen erstellen. Geben Sie die Gesamtkosten der beiden Algorithmen aus und vergleichen Sie diese.

Auch hierfür steht Ihnen wieder eine Datei „graph.txt“ zur Verfügung. Die erste Zeile enthält wieder die Knotenzahl, alle weiteren enthalten die Kanten mit Start und Endknoten sowie einem Gewicht. Beachten Sie folgende Notation: Startknoten → Endknoten → Gewicht.

Die Implementierung muss ebenfalls in eine Beispieldatei integriert werden. Sie müssen das Einlesen des Graphen und den Algorithmus übernehmen. Die Übrigen Funktionalitäten sind bereits gegeben.

4.2 Lösungshinweise

Für die Lösung steht Ihnen ein Framework zur Verfügung, was die Elemente der bisherigen Praktika nutzt. Dort sind die Stellen markiert, in denen Sie Ihre Algorithmen integrieren sollen. Es reicht, diese Lücken zu füllen, eine weitere Ergänzung ist nicht nötig.

Zudem folgen in den nächsten Abschnitten ein paar allgemeine Hinweise zu dem Algorithmen, die Ihnen die Programmierung erleichtern sollen. Beachten Sie, dass hier nur Hinweise gegeben werden. Gegebene Beispiele müssen erweitert oder überarbeitet werden und können nicht so übernommen werden.

4.2.1 Graph- und Knoten-Klasse

Um Ihnen die Arbeit zu erleichtern, wird hier ein Graph bereitgestellt. Er orientiert sich an den ersten Aufgaben und das meiste daraus sollte Ihnen bekannt sein. Ein Knoten besteht wie bei den Listen und Bäumen aus einem Wert als „Namen“ und bekommt nun noch zusätzlich die Attribute „besucht“ und „Distanz“. Wozu Sie diese verwenden können, wird in den nächsten Abschnitten erklärt.

Der Graph ist grundsätzlich aufgebaut wie ein Baum in Aufgabe 2. Zusätzlich werden aber mehr als 2 Nachbarn zugelassen und ein Knoten kann auch eine Verbindung wieder zum Elternknoten haben. Da es sich in diesen Aufgaben um ungerichtete Graphen handelt, wird jede Kante als Hin- und Rückkante angelegt. Jeder Knoten enthält daher eine Liste von Kanten. Eine Kante besteht dabei aus einem Zielknoten und für den zweiten Aufgabenteil zusätzlich noch aus einem Gewicht.

Ebenfalls wird ihnen in dieser Klasse bereits das verarbeiten der Datei zur Verfügung gestellt. Den Aufruf der Datei müssen Sie jedoch noch integrieren, da dies von Ihrer Plattform abhängt.

4.2.2 Datei einlesen

Zum einlesen einer „.txt“ Datei nutzen Sie <fstream>. Ein Beispiel zum einlesen sehen sie hier:

```
#include <iostream>
#include <fstream>                    // ofstream and ifstream
```

```
using namespace std;

int main(void)
{
  ifstream file ;                          // new reading stream
   file .open("Test.txt", ios_base::in);   // Open File "Test.txt" for reading

  if (! file )                             // File not open?
    cout << "Cannot open file\n" << endl;
  else {
    char c;
    while ( file .get(c)) {                // Reading character by character
      //do what ever is needed to build the graph
    }
  }

 return 0;
}
```

Um nun den Graphen ein zu lesen, schauen Sie sich die Trennzeichen zwischen den Zeichen einer Zeile und den verschiedenen Zeilen an. Anhand derer können Sie eine Zuordnung für ihren Graphen vornehmen. Zum Übertragen in den Graphen nutzen Sie die Graphenklasse. Sie finden die entsprechenden Stellen in den Beispieldateien zum ergänzen.

4.2.3 Tiefensuche

Die Tiefensuche wurde Ihnen in der Vorlesung vorgestellt. Sie sucht zunächst solange in die Tiefe, bis kein weiterer Knoten als Kind mehr folgt und geht dann eine Ebene höher. Zur Implementierung einer iterativen oder rekursiven Suche, ist in Algorithmus 1 und 2 der Pseudocode dargestellt. Beide Algorithmen sind hier so aufgebaut, dass sie prüfen ob ein Graph zusammenhängend ist. Nach einer abgeschlossenen Tiefensuche müssen alle Knoten besucht sein. Gibt es Knoten, die nicht durch die Tiefensuche erreicht werden können, so kann es keine Verbindung zum Startknoten geben.

Algorithm 1: Tiefensuche iterativ

input : Graph G, Startknoten s
output : Zusammenhang

```
Erzeuge einen Stack St
St ← s (Startknoten)
for (alle Knoten v) do
    v.visited ← false
while (St not empty) do
    n ← oberstes Element aus St
    n.visited ← true
    for (Alle Kinder k von n) do
        if (k not visited) then
            Qu ← k
for (alle Knoten k) do
    if (k not visited) then
        return false
return true
```

Algorithm 2: Tiefensuche rekursiv

input : Graph G, Startknoten s
output : Zusammenhang

```
Function dfs-start(x)
    for (alle Knoten v) do
        v.visited ← false
    dfs(s)
    for (alle Knoten v) do
        if (v.visited equals false) then
            return false
    return true
Function dfs(x)
    x.visited ← true
    for (Alle Kinder k von x) do
        if (k not visited) then
            dfs (k)
```

4.2.4 Breitensuche

Im Gegensatz zur Tiefensuche wird hier zunächst in der Breite gesucht. Das bedeutet, zu einem Knoten werden zunächst alle Kinder untersucht, bevor die Kindes-Kinder betrachtet werden. Die Programmierung ist iterativ sehr ähnlich zur Tiefensuche, statt einem Stack wird nun eine Queue verwendet.

Algorithm 3: Breitensuche iterativ

input : Graph G, Startknoten s

output : Zusammenhang

```
Erzeuge eine Queue Qu
Qu ← s (Startknoten)
for (alle Knoten v) do
    v.visited ← false
while (Qu not empty) do
    n ← oberstes Element aus St
    n.visited ← true
    for (Alle Kinder k von n) do
        if (k not visited) then
            Qu ← k
for (alle Knoten k) do
    if (k not visited) then
        return false
return true
```

Algorithm 4: Breiensuche rekursiv

input : Graph G, Startknoten s

output : Zusammenhang

```
Function bfs-start(x)
    for (alle Knoten v) do
        v.visited ← false
    dfs(s)
    for (alle Knoten v) do
        if (v.visited equals false) then
            return false
    return true
Function bfs(x)
    x.visited ← true
    for (Alle Kinder k von x) do
        if (k not visited) then
            dfs (k)
```

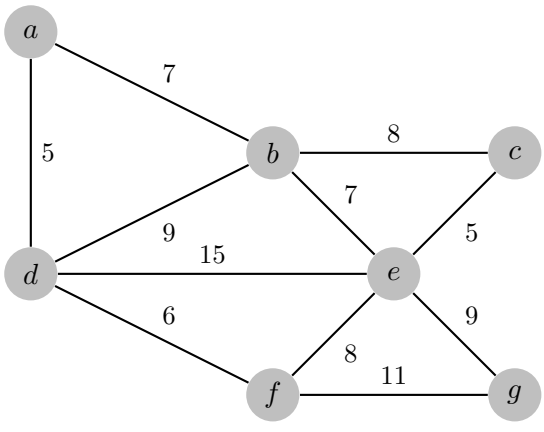
4.2.5 Prim

Der Prim Algorithmus dient der Erstellung eines Minimalen Spannbaum (Minimal-Spanning-Tree MST). Algorithmus 5 beschreibt ihn als Pseudocode.

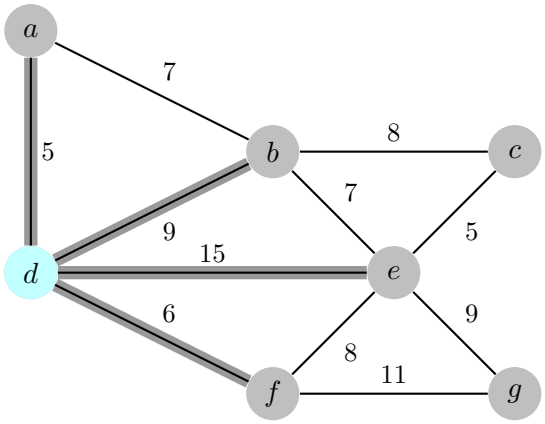
Algorithm 5: Prim Algorithmus

input : Graph G, Gewichtsfunktion w, Startknoten s
output : MST zu Graph G
parameter: $prev[u]$: Elternknoten von Knoten u im Spannbaum
$Adj[u]$: Adjazenzliste von u (alle Nachbarknoten)
$wert[u]$: Abstand von u zum entstehenden Spannbaum
$Q \leftarrow$ Prioritätswarteschlange
Schreibe alle Knoten V_G in Q
for *alle* $u \in Q$ **do**
    setze $wert[u] \leftarrow \infty$
    setze $prev[u] \leftarrow 0$
setze $wert[s] = 0$
while Q *nicht leer* **do**
    Wähle Knoten u aus Q mit $min(wert[u])$
    for $v \in Adj[u]$ **do**
        if $v \in Q$ *und* $w(u, v) < wert[v]$ **then**
            setze $prev[v] \leftarrow u$
            und $wert[v] \leftarrow w(u, v)$

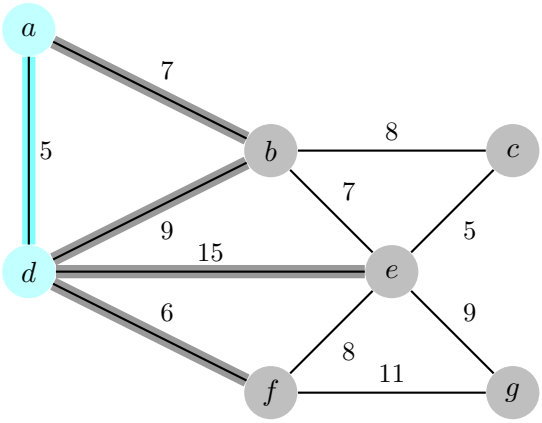
Ein Beispiel mit dem folgenden Graph beschreibt den Algorithmus wobei als Startknoten Knoten d verwendet wurde. Die grau eingefärbten Kanten sind die zur Auswahl stehenden und die mint gefärbten Kanten die dem MST hinzugefügten Kanten.



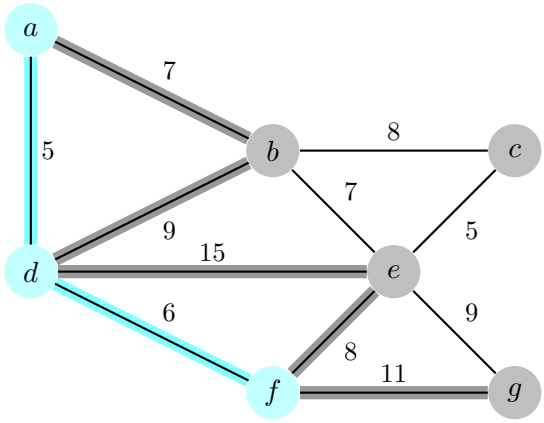
Originalgraph



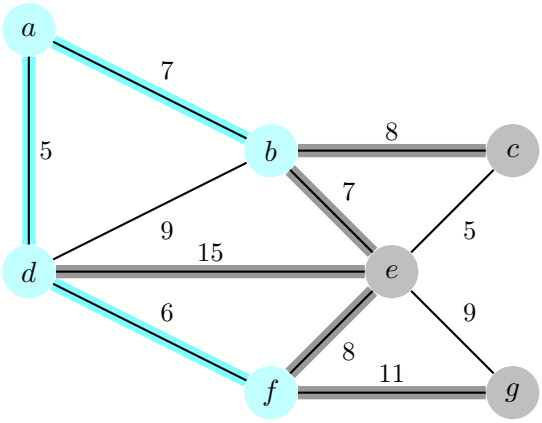
Startknoten d, erste Kantenauswahl



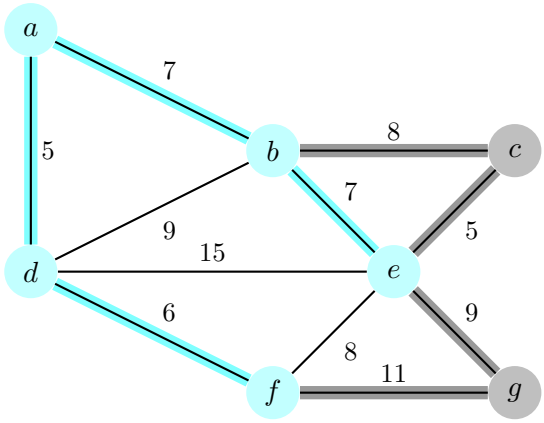
a hinzugefügt, neue Kantenauswahl



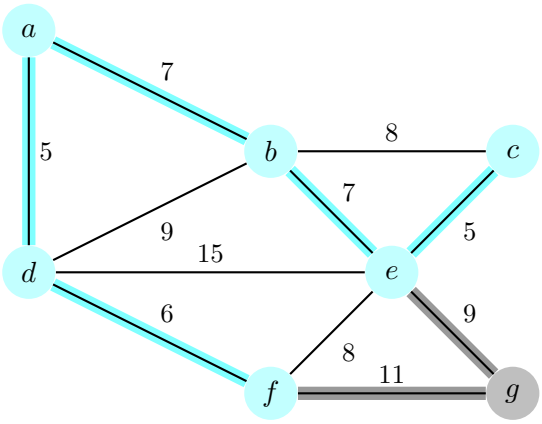
f hinzugefügt, neue Kantenauswahl



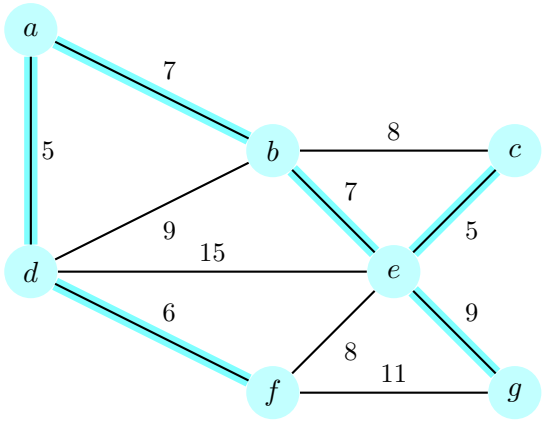
b hinzugefügt, neue Kantenauswahl



e hinzugefügt, neue Kantenauswahl



c hinzugefügt, neue Kantenauswahl



g hinzugefügt, MST vollständig

Der vollständige MST verbindet nun alle Knoten kostenminimal von Knoten d ausgehend. Die Kosten dieses MST betragen 39.

4.2.6 Kruskal

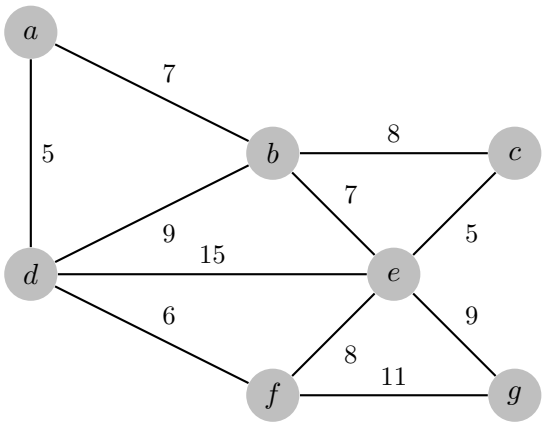
Dieser Algorithmus ist ebenfalls zum finden eines MST, arbeitet aber leicht anders als Prim. In diesem Fall wird kein Startknoten gewählt, sondern alle Kanten zu Beginn der Größe nach sortiert und immer die (dem Kantengewicht nach) kleinste Kante als nächste gewählt, sofern mindestens einer der Knoten noch nicht mit dem MST verbunden wurde. Auch hierzu sehen Sie in Algorithmus 6 den Pseudocode, den Sie zur Implementierung verwenden können.

Algorithm 6: Kruskalt Algorithmus

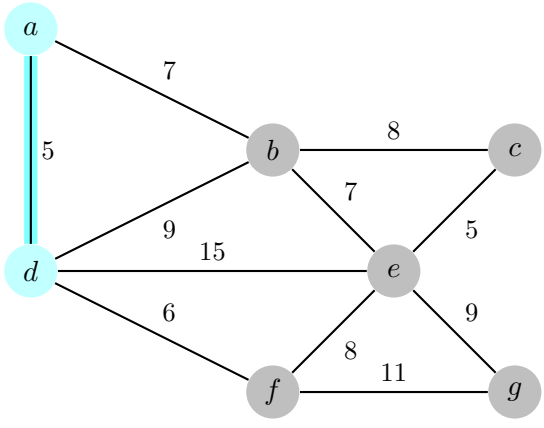
input : Graph G, Gewichtsfunktion w
output : MST zu Graph G
parameter: $E(1)$ Kantenmenge der in MST enthaltenen Kanten
$E(2)$ Kantenmenge der noch zu bearbeitenden Kanten
$E_1 \leftarrow 0$
$L \leftarrow E$
Sortiere die Kanten in L aufsteigend nach ihrem Kantengewicht.
while $L \neq 0$ **do**
  Wähle eine Kante $e \in L$ mit kleinstem Kantengewicht
  Entferne die Kante e aus L
  if *Graph* $(V, E_1 \cup \{e\})$ *enthält keinen Kreis* **then**
    $E_1 \leftarrow E_1 \cup \{e\}$

Die folgenden Abbildungen zeigen am gleichen Beispiel wie bei Prim den Ablauf des Algorithmus. Die Kanten wurden initial wie folgt sortiert:

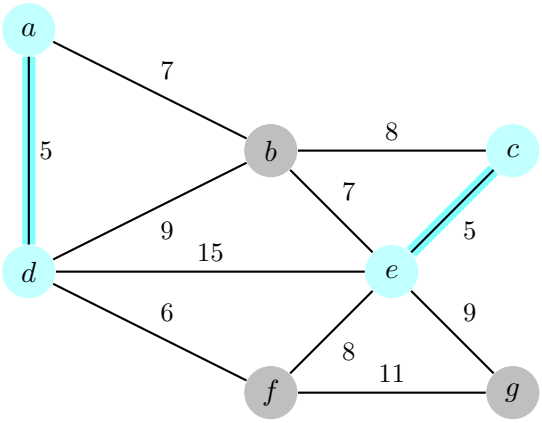
$(a, d, 5)(c, e, 5)(d, f, 6)(a, b, 7)(b, e, 7)(b, c, 8)(e, f, 8)(b, d, 9)(e, g, 9)(f, g, 11)(d, e, 15)$



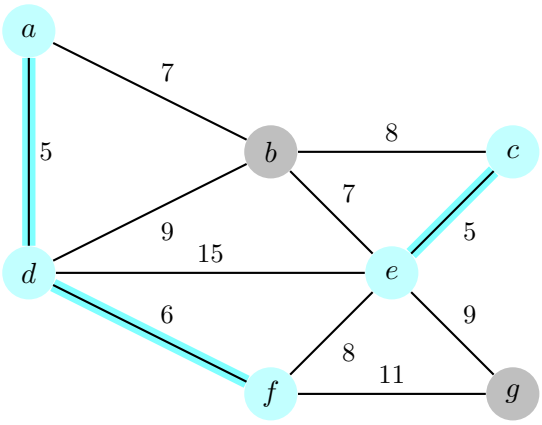
Originalgraph



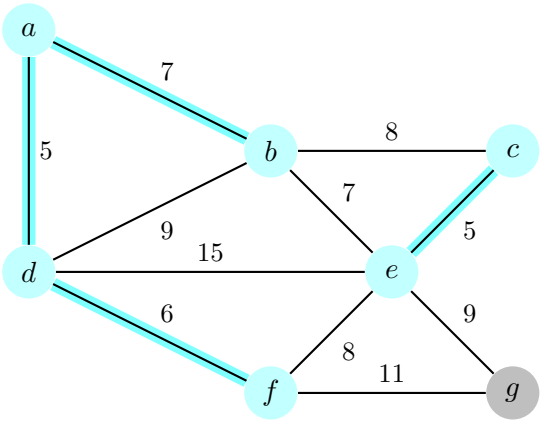
Kante $(a, d, 5)$ hinzugefügt



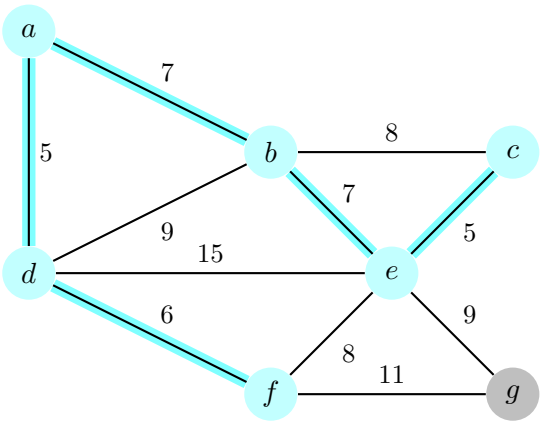
Kante $(c, e, 5)$ hinzugefügt



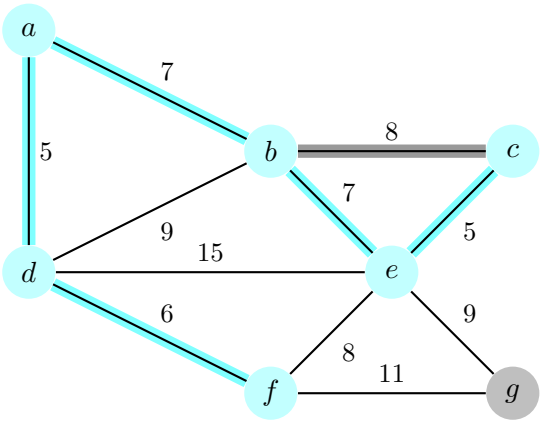
Kante $(d, f, 6)$ hinzugefügt



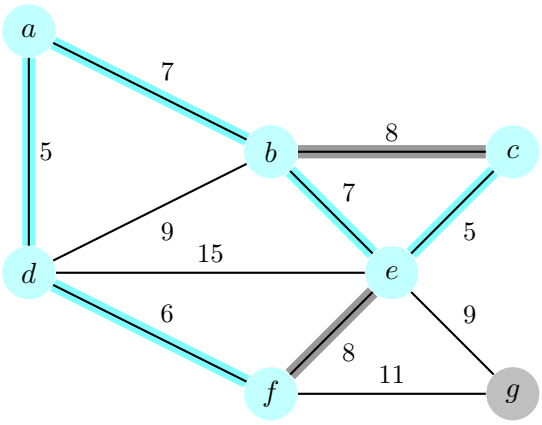
Kante $(a, b, 7)$ hinzugefügt



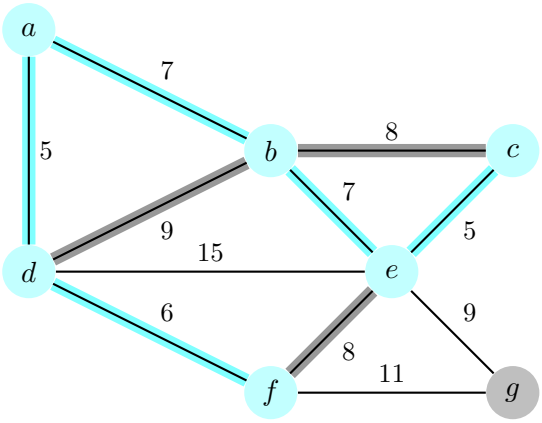
Kante $(b, e, 7)$ hinzugefügt



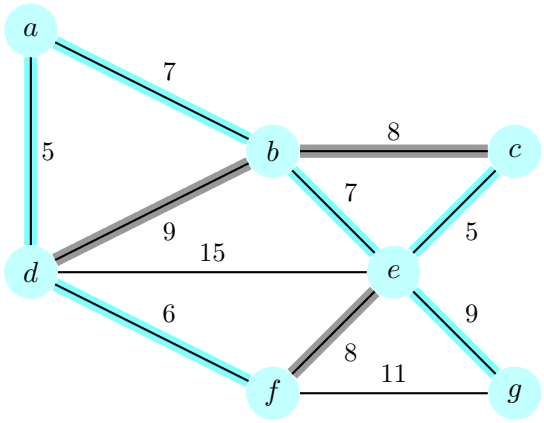
Kante $(b, c, 8)$ verwerfen



Kante $(e, f, 8)$ verwerfen



Kante $(b, d, 9)$
verwerfen



Kante $(e, g, 9)$ hinzufügen

Nachdem Knoten g hinzugefügt wurde ist der MST vollständig und es müssen keine weiteren Kanten mehr betrachtet werden. Die Gesamtkosten berechnen sich durch die Summe aller verwendeten Kante. In diesem Fall wurden exakt die gleichen Kanten wie beim Prim Algorithmus verwendet und somit beträgt die Summe des MST auch hier 39.