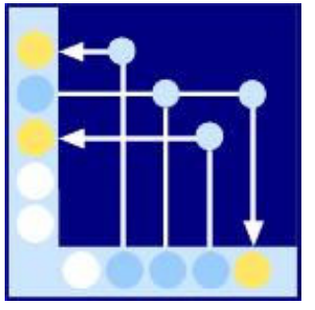
Hochschule Aalen
Fakultät Elektronik und Informatik
Studienbereich Informatik



Algorithmen und Datenstrukturen 2

Vorlesung im Wintersemester 2024/2025

Prof. Dr. habil. Christian Heinlein

christian.heinleins.net

1 Einleitung und Überblick

1.1 Vorlesungsüberblick

- Streuwerttabellen (hash tables)
  - Implementierung mit Verkettung (chaining)
  - Implementierung mit offener Adressierung (open addressing)
- Vorrangwarteschlangen (priority queues)
  - Implementierung mit binären Halden (binary heaps)
  - Implementierung mit Binomial-Halden (binomial heaps)
- Programmiertechniken
  - Nächstbest-Algorithmen (greedy algorithms)
  - Tabellengestützte Programmierung (dynamic programming)

- Elementare Graphalgorithmen
  - Breitensuche (breadth-first search)
  - Tiefensuche (depth-first search)
  - Starke Zusammenhangskomponenten (strongly connected components)
- Minimale Spannbäume (minimum spanning trees) und Gerüste
  - Algorithmus von Kruskal
  - Algorithmus von Prim
- Kürzeste Wege in Graphen (shortest path algorithms)
  - Algorithmus von Bellman und Ford
  - Algorithmus von Dijkstra
  - Algorithmus von Floyd und Warshall
- Anwendung mathematischer Methoden
  - Korrektheitsbeweise
  - Laufzeitabschätzungen

1.2 Organisatorisches

- Vorlesung
  - Folien sind online verfügbar
    - in der Regel kapitelweise, normalerweise vor Beginn des jeweiligen Kapitels
    - bei Bedarf nachträgliche Ergänzungen oder Korrekturen
  - Beweise werden zum Teil an der Tafel entwickelt
  - Programme werden zum Teil interaktiv am Rechner entwickelt
- „Papier und Bleistift“-Übungsaufgaben zu jedem Kapitel
- 3 größere Programmieraufgaben in C++ während des Semesters
  als Teil der Prüfungsleistung (Anteil 1/3)
- 90-minütige Klausur im Prüfungszeitraum (Anteil 2/3)
- Voraussetzungen
  - Keine formalen Voraussetzungen
  - Aber die Inhalte folgender Vorlesungen werden inhaltlich vorausgesetzt:
    - Mathematik-Vorlesungen des 1. und 2. Semesters
    - Programmier-Vorlesungen des 1. und 2. Semesters
    - Algorithmen und Datenstrukturen 1

1.3 Literaturhinweise

- T. H. Cormen, C. E. Leiserson, R. Rivest, C. Stein: *Algorithmen – Eine Einführung* (3. Auflage). R. Oldenbourg Verlag, München, 2010.

  Stellenweise werden Sachverhalte in dieser Vorlesung jedoch bewusst anders dargestellt als in diesem Standardwerk.

- R. Sedgewick, K. Wayne: *Algorithmen* (4., aktualisierte Auflage). Pearson, Hallbergmoos, 2010.

- U. Schöning: *Algorithmik*. Spektrum Akademischer Verlag, Heidelberg, 2001.

- D. E. Knuth: *The Art of Computer Programming* (Band 1 bis 3). Addison-Wesley, Reading, MA, 2001.

- Wikipedia, die freie Enzyklopädie: https://de.wikipedia.org
  Wikipedia, the free encyclopedia: https://en.wikipedia.org

  Wikipedia ist häufig eine nützliche Informationsquelle.
  Stellenweise sind die Artikel jedoch ungenau oder widersprüchlich, z. B. beim Thema Graphen.

1.4 Kurze Einführung in C++

1.4.1 Allgemeines

- C++ ist eine echte „Obermenge“ von C, d. h. alle aus C bekannten Sprachmittel können (im wesentlichen) unverändert verwendet werden.
- Darüber hinaus bietet C++ zahlreiche zusätzliche Sprachmittel, nicht nur für objektorientierte Programmierung, z. B. Schablonen (templates) und überladene Operatoren.

1.4.2 Standardbibliothek

- Die Definitionsdateien der C++-Standardbibliothek, wie z. B. `<iostream>` und `<string>`, haben keine Endung `.h` wie in C.
- Sie definieren Namen wie z. B. `std::cout` und `std::string` im Namensbereich `std`.
- `using namespace std` bewirkt, dass alle diese Namen auch ohne das Präfix `std::` verwendet werden können (vergleichbar mit `import std.*` in Java).
- Alternativ kann man Namen einzeln z. B. mit `using std::cout` importieren.

1.4.3 Hauptprogramm

- Das Hauptprogramm `main` ist wie in C eine globale Funktion mit Resultattyp `int` (nicht `void` wie in Java) und optionalen Parametern `int argc, char* argv []`, über die auf die Kommandozeilenargumente zugegriffen werden kann.

- Der Resultatwert von `main` wird an das Betriebssystem zurückgegeben. Resultatwert 0 zeigt üblicherweise an, dass das Programm erfolgreich ausgeführt wurde, während ein Wert ungleich 0 einen Fehler signalisiert.

- Wenn die Funktion `main` keine `return`-Anweisung ausführt, liefert sie automatisch den Wert 0 zurück. (Diese Sonderregel gilt aber nur für `main`.)

1.4.4 Zeichenketten

- Die Definitionsdatei `<string>` definiert einen Typ `string`, der ähnlich wie `String` in Java verwendet werden kann.

- Anders als in Java, können Objekte des Typs `string` aber problemlos mit dem normalen Gleichheitsoperator `==` verglichen werden.

1.4.5 Ein- und Ausgabe

- `cout << x` gibt den Wert `x` auf der Standardausgabe aus, sofern es eine passende Definition des Operators `<<` für den Typ von `x` gibt.

- Diese Ausgaben können verkettet werden, um verschiedene Dinge nacheinander auszugeben, z. B. `cout << "Der Wert von x ist " << x << endl`.

- `endl` bezeichnet dabei einen Zeilentrenner.

- Für eine Variable `x` irgendeines Typs `T` liest `cin >> x` einen Wert des Typs `T` von der Standardeingabe und speichert ihn in der Variablen `x`, sofern es eine passende Definition des Operators `>>` gibt.
  Führender Zwischenraum (Leerzeichen, Tabulatorzeichen und Zeilentrenner) wird dabei überlesen. (Deshalb kann man damit z. B. keine leere Zeichenkette einlesen.)

- Diese Eingaben können ebenfalls verkettet werden, um verschiedene Dinge nacheinander einzulesen, z. B. `cin >> x >> y`.

- Die Definitionsdatei `<iostream>` enthält entsprechende Definitionen der Operatoren `<<` und `>>` für alle elementaren Typen (ganze Zahlen, Gleitkommazahlen, Wahrheitswerte, Zeiger), `<string>` enthält entsprechende Definitionen für Zeichenketten.

- `getline(cin, s)` liest eine ganze Zeile von der Standardeingabe und speichert sie (ohne den Zeilentrenner am Ende) in der `string`-Variablen `s`.

- Sowohl `cin >> x` als auch `getline(cin, s)` liefern einen Wert zurück, der als Bedingung von `if` oder `while` verwendet werden kann, um zu überprüfen, ob etwas gelesen werden konnte oder ob man das Ende der Eingabe erreicht hat.

1.4.6 Referenzparameter

- Parameterwerte werden wie in C normalerweise „by value“ an Funktionen übergeben, d. h. es werden Kopien der Werte übergeben.

- Dementsprechend sind Zuweisungen an Parameter nur innerhalb einer Funktion wirksam.

- Wenn Werte über Parameter zurückgegeben werden sollen, müssen diese als Referenzparameter gekennzeichnet werden, zum Beispiel:

```
void swap (int& x, int& y) {
  cout << "Am Anfang von swap: " << x << " " << y << endl;
  int z = x; x = y; y = z;
  cout << "Am Ende von swap: " << x << " " << y << endl;
}
```

```
int main () {
  int a = 1, b = 2;
  cout << "Vor dem Aufruf von swap: " << a << " " << b << endl;
  swap(a, b);
  cout << "Nach dem Aufruf von swap: " << a << " " << b << endl;
}
```

Nach dem Aufruf der Funktion `swap` sind die Werte der Variablen `a` und `b` wie gewünscht vertauscht.
Würde man die Referenzsymbole `&` bei den Parametern `x` und `y` weglassen, würden ihre Werte zwar vertauscht, die Werte von `a` und `b` blieben aber unverändert.

1.4.7 Strukturen und Klassen

- Zwischen Strukturen (`struct`) und Klassen (`class`) besteht kein wesentlicher Unterschied. Beide können neben Datenelementen auch Konstruktoren, Elementfunktionen (Methoden) und Destruktoren besitzen. Der einzige Unterschied ist, dass in einer Struktur alle Elemente öffentlich und in einer Klasse alle Elemente privat sind, sofern man nichts anderes vereinbart.

- Konstruktoren können i. w. wie in Java definiert werden.

- Eine Besonderheit sind jedoch Elementinitialisierer `data(expr)`, die zwischen Parameterliste und Rumpf eines Konstruktors stehen können und ausdrücken, dass

das Datenelement `data` mit dem Wert des Ausdrucks `expr` (meist einfach ein Parameter des Konstruktors) initialisiert wird.
Das ist vergleichbar mit einer Zuweisung `data = expr` im Rumpf des Konstruktors.
Ein Elementinitialisierer kann jedoch auch dann verwendet werden, wenn diese Zuweisung nicht möglich ist, z. B. weil das Datenelement `const` ist.

- Wenn Elementinitialisierer zur Initialisierung der Datenelemente verwendet werden, ist der Rumpf eines Konstruktors häufig leer.

- Wenn ein Typ keinen Konstruktor definiert, besitzt er automatisch einen leeren parameterlosen Konstruktor.
  Sobald einer oder mehrere Konstruktoren definiert werden, gibt es den parameterlosen Konstruktor aber nicht mehr automatisch; er muss dann ggf. explizit definiert werden.

- Der parameterlose Konstruktor wird an manchen Stellen „unsichtbar“ aufgerufen, z. B. wenn eine Variable des Typs nicht explizit initialisiert wird oder wenn eine Reihe (array) mit Objekten des Typs erzeugt wird.
  Wenn der parameterlose Konstruktor dann nicht existiert, erhält man beim Übersetzen einen Fehler.

1.4.8 Zeiger und dynamische Reihen

- `new` kann wie in Java zur Erzeugung dynamischer Objekte und Reihen verwendet werden und ist vergleichbar mit `malloc` in C.
  Das Pendant zu `free` heißt `delete` (für einzelne Objekte) bzw. `delete []` (für Reihen).
  Am Ende von `main` wird automatisch der gesamte Speicher des Programms freigegeben. Deshalb ist `delete` nur wichtig, um schon während der Ausführung eines Programms nicht mehr benötigten Speicher freizugeben.

- `new T [n]` erzeugt zur Laufzeit eine Reihe mit `n` Elementen des Typs `T` und liefert einen Zeiger des Typs `T*` darauf zurück. Wenn man diesen Zeiger an eine Variable `T* a` zuweist, kann `a` (aufgrund der aus C bekannten Äquivalenz von Zeigern und Reihen) anschließend wie eine Reihe verwendet werden; insbesondere bezeichnet `a[i]` für `i` von 0 bis `n-1` das `i`-te Element der Reihe.

- Abhängig vom Typ `T`, werden die Elemente der Reihe dabei entweder automatisch mit dem parameterlosen Konstruktor von `T` initialisiert, oder sie bleiben uninitialisiert. Für elementare Typen sowie einfache Strukturtypen, die nur elementare Typen enthalten, findet z. B. keine Initialisierung statt. Für Typen, die einen oder mehrere Konstruktoren besitzen, wird der parameterlose Konstruktor aufgerufen, den es dann auch geben muss. Die genauen Regeln sind jedoch kompliziert.

- Durch Verwendung von `new T [n] ()` kann die Initialisierung der Elemente mit dem parameterlosen Konstruktor erzwungen werden, was insbesondere für elementare Typen, zu denen auch Zeigertypen zählen, sinnvoll ist; Elemente mit Zeigertypen werden dann z. B. automatisch mit `nullptr` initialisiert.

- Eine Reihe mit `n` Zeigern des Typs `T*`, die alle mit `nullptr` initialisiert sind, kann somit mittels `new T* [n] ()` erzeugt und an eine Variable `T** a` zugewiesen werden. Jedes Element `a[i]` hat dann Typ `T*`.

- In Java sind Variablen, deren Typ eine Klasse oder eine Reihe ist, automatisch immer sog. Referenzen und entsprechen deshalb Zeigern in C++, vgl. nachfolgende Tabelle.

C++	Java

`class Person {`
`  string name;`
`public:`
`  Person (string n)`
`  : name(n) {}`
`};`
	`class Person {`
	`  private String name;`
	`  public Person (String n) {`
	`  name = n;`
	`  }`
	`}`

`Person p("CH");` `cout << p.name << endl;`	Nicht möglich

`Person* p = new Person("CH");`
`cout << p->name << endl;`
`delete p;`
	`Person p = new Person("CH");`
	`System.out.println(p.name);`
	Nicht möglich, da nicht nötig

`int a [10];` `cout << a[i] << endl;`	Nicht möglich

`int* a;`
`a = new int [10];`
`cout << a[i] << endl;`
`delete [] a;`
	`int [] a;`
	`a = new int [10];`
	`System.out.println(a[i]);`
	Nicht möglich, da nicht nötig

1.4.9 Überladene Operatoren

- Die vordefinierten Operatoren für Arithmetik, Logik etc. können für eigene Typen neu definiert werden, zum Beispiel (hierfür muss `name` ein öffentliches Datenelement von `Person` sein):

```
// Sind p1 und p2 logisch gleich?
bool operator== (Person p1, Person p2) {
  // Verwendung des Gleichheitsoperators für Zeichenketten.
  return p1.name == p2.name;
}

// Sind p1 und p2 logisch verschieden?
bool operator!= (Person p1, Person p2) {
  // Verwendung des zuvor definierten Gleichheitsoperators
  // für Personen.
  return !(p1 == p2);
}

// Verwendung:
Person p1("CH");
Person p2("AH");
if (p1 == p2) cout << "gleich" << endl;
else cout << "nicht gleich" << endl;
```

1.4.10 Schablonen (templates)

- Eine Typschablone (class template) definiert eine Familie von Typen, die alle i. w. gleich aufgebaut sind, sich aber in der Belegung eines oder mehrerer Typparameter unterscheiden.

- Eine Funktionsschablone (function template) definiert entsprechend eine Familie von Funktionen.

1.4.11 Sonstiges

- `nullptr` bezeichnet einen Nullzeiger eines beliebigen Zeigertyps.

- Anders als in C, gibt es einen eigenen Typ `bool` mit Werten `true` und `false`.

- Trotzdem können auch Werte elementarer Typen wie in C als Bedingungen verwendet werden. Der Wert Null oder ein Nullzeiger wird als `false` interpretiert, alle anderen Werte als `true`.

- `using X = Y` ist gleichbedeutend mit `typedef Y X` in C, d. h. es definiert den Namen `X` als Synonym bzw. Alias für den Typ `Y`.

- Eine `for`-Schleife

```
for (init; cond; next) { body }
```

ist (in C, C++ und Java) fast gleichbedeutend mit einer `while`-Schleife

```
init; while (cond) { body; next; }
```

Der einzige Unterschied ist, dass `continue` in einer `while`-Schleife mit der Auswertung von `cond` fortfährt, in einer `for`-Schleife jedoch mit der Auswertung von `next`.

`init` und `next` können aus mehreren Ausdrücken bestehen, die durch Kommas getrennt sind.

Wenn die „Laufvariable“ nach der Schleife nicht mehr benötigt wird, kann sie „nebenbei“ im `init`-Teil definiert werden.

- Eine `for`-Schleife

```
for (T x : a) { body }
```

führt den Schleifenrumpf `body` für jedes Element `x` von `a` aus. `a` muss entweder eine Reihe mit bekannter Größe (also kein Zeiger) oder ein Container der Standardbibliothek (`vector`, `list` etc.) oder etwas Vergleichbares mit Elementen des Typs `T` sein.

1.4.12 Beispiel

```
// Vorzeichenlose ganze Zahl.
using uint = unsigned int;

// Verkettete Liste mit Elementen des Typs T.
template <typename T>
struct List {
  // Knoten einer solchen Liste.
  struct Node {
    T elem;     // Element.
    Node* next; // Zeiger auf den nächsten Knoten oder Nullzeiger.

    // Initialisierung mit Element e und Verkettungszeiger n.
    Node (T e, Node* n) : elem(e), next(n) {}
  };

  // Zeiger auf den ersten Knoten oder Nullzeiger.
  Node* head;

  // Initialisierung als leere Liste.
  List () : head(nullptr) {}
```

```
// Element x zur Liste hinzufügen (und zwar am Anfang).
void add (T x) {
  head = new Node(x, head);
}

// i-tes Element der Liste (gezählt ab 0) über den Referenz-
// parameter x zurückliefern, falls vorhanden.
// Resultatwert true genau dann, wenn es ein i-tes Element gibt.
bool get (uint i, T& x) {
  for (Node* p = head; p; p = p->next, i--) {
    if (i == 0) {
      x = p->elem;
      return true;
    }
  }
  return false;
}
```

```
  // Das erste Element x aus der Liste entfernen, falls vorhanden.
  // Resultatwert true genau dann, wenn Element x vorhanden war.
  // Bei Verwendung dieser Funktion muss es einen Operator == zum
  // Vergleich zweier Objekte des Typs T geben. Bei Bedarf kann so
  // ein Operator passend definiert werden, vgl. § 1.4.9.
  bool remove (T x) {
    Node* q = nullptr;
    for (Node* p = head; p; q = p, p = p->next) {
      if (p->elem == x) {
        if (q) q->next = p->next;
        else head = p->next;
        delete p;
        return true;
      }
    }
    return false;
  }
};
```

```
#include <iostream>      // cin, cout, endl
#include <string>        // string
using namespace std;

// Interaktives Testprogramm.
int main () {
  // Liste von Zeichenketten.
  List<string> ls;

  // Hilfsvariablen.
  string cmd, elem;

  // Befehle von der Standardeingabe lesen und verarbeiten:
  // + elem -> elem zur Liste hinzufügen
  // ? i    -> i-tes Element der Liste ausgeben
  // - elem -> elem aus der Liste entfernen
  // q      -> Programm beenden
```

```
  while (cout << "cmd: ", cin >> cmd) {
    if (cmd == "+") {
      cin >> elem;
      ls.add(elem);
    }
    else if (cmd == "?") {
      uint i;
      cin >> i;
      if (ls.get(i, elem)) cout << elem;
      cout << endl;
    }
    else if (cmd == "-") {
      cin >> elem;
      ls.remove(elem);
    }
    else if (cmd == "q") {
      break;
    }
  }
}
```