Eidgenössische Technische Hochschule Zürich

Datenstrukturen und Algorithmen

Summer Semester 2026

Lecture Notes

Contents

Datenstrukturen und Algorithmen

Fynn Krebser–fkrebser@student.ethz.ch

Sommersemester 2026

Lec 1

1 Effizientes Speichermanagement

Eine Grosse Stärke von C++ ist die Möglichkeit, Variablen mit Werten zu darzustellen. Zum Beispiel kann mit `std::vector<int> w=v` ein ganzer Vektor kopiert werden.

Dies bedeutet auch, dass standardmässig alle Variablen BY VALUE an Funktionen übergeben werden. Somit verändert sich ein Vektor nicht, wenn er in einer Funktion verwendet wird. Andererseits, muss der Speicher kopiert werden, was Zeit kostet.

Es können Referenztypen verwendet werden, um Argumente als Alias zu übergeben, so dass kein Speicher kopiert werden muss. Nachteil davon ist, dass so keine Werte übergeben werden können, welche keine Adresse im Speicher haben, da es auch keinen Sinn macht z.B. `swap(3,a+a)` aufzurufen.

Mögliche Nachteile einer pass by reference sind, dass wir keine R-Werte übergeben können und dass wir nicht mehr sicher sein können, dass die übergebenen Werte nicht verändert werden, da sie als Alias übergeben werden.

Eine Möglichkeit ist eine Const Referenz zu verwenden. Dies löst beide unsere Probleme.

1.1 Wertekategorien

Audrücke in C++ haben einen Typ, repräsentieren einen Wert und repräsentieren möglicherweise ein Objekt mit Adresse.

Wenn wir einen Zuweisung wie L=R ausführen möchten, müssen die Typen übereinstimmen aber auch die Werte Kategorien von L und R müssen stimmen.

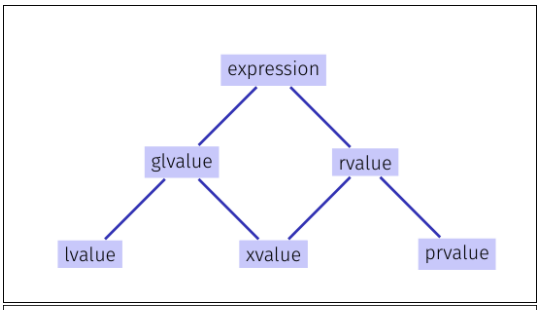
In Informatik haben wir Werte welche Links von einer Zuweisung stehen L-Werte genannt und Werte welche Rechts von einer Zuweisung stehen R-Werte genannt.

Somit kann ein L-Wert verwendet werden, wo ein R-Wert erwartet wird, aber nicht umgekehrt.

Heutzutage hat sich C++ weiterentwickelt und es gibt weitere Wertekategorien.

z.B. die generalized l value, die pure r value und die expired value.

Wenn wir v=w schreiben können, dann muss v weiter leben und somit kann v nicht verschoben werden. Wenn wir z.B. output(abs(v)) schreiben, dann könnte abs(v) verschoben werden, da es nicht weiter lebt. Eine prvalue sind Literale. Sie haben keinen Namen und können nicht verschoben werden, da sie keinen Speicher haben.



Eigenschaft		Hat Identität	Keine Identität
		glvalue	
Kann verschoben werden	rvalue	xvalue	prvalue
Kann nicht verschoben werden		lvalue	

Figure 1: Die Wahrheit

Table 1: Aufschlüsselung der Wertekategorien

Kartegorie	Beispiel
L-Wert	Variablen, Zeiger
xvalue	Resultat eines Funktionaufrufs
prvalue	Literale, Resultat von Operatoren

Definition 1.1: R-Wert-Referenz

Eine R-Wert-Referenz ist eine Referenz auf einen R-Wert. Sie wird mit `&&` deklariert.

Wenn das Quellobjekt einer Zuweisung direkt nach der Zuweisung nicht weiter existiert, dann kann der Compiler den Move-Zuweisungsoperator anstelle des Zuweisungsoperators einsetzen. Damit wird eine potentiell teure Kopie vermieden.

Dabei wird der Move-Konstruktor oder der Move-Zuweisungsoperator aufgerufen, wenn das Quellobjekt ein R-Wert ist. Wenn das Quellobjekt ein L-Wert ist, wird der Kopierkonstruktor oder der Kopierzuweisungsoperator aufgerufen.

Definiert werden die Move-Konstruktor und der Move-Zuweisungsoperator wie folgt:

```cpp
class MyClass {
public:
    // Move-Konstruktor
    MyClass(MyClass&& other) {
        // Ressourcen von 'other' übernehmen
    }
    MyClass& operator=(MyClass&& other) {
        if (this != &other) {
            // Ressourcen von 'other' übernehmen
        }
        return *this;
    }
};
```

Ein Beispiel dafür wäre der Code

```
Vec operator + (const Vec& a, const Vec& b) {
  Vec tmp = a;
  // Add b to tmp
  return tmp;
}
int main(){
  Vec f;
  f = f + f + f + f;
}
```

Hierbei werden ganze 4 Kopien erstellt. Wenn wir den Move operator hinzufügen, werden nur noch 3 Kopien erstellt. Hierbei könnte man den Compiler unterstützen und den Vektor a by value übergeben.

```
Vec operator + (Vec a, const Vec& b) {
    // Add b to a
    return a;
}
```

Nun wird nur noch eine Kopie erstellt, bei der ersten Addition.

Move-Semantik kommt zum Einsatz, wenn ein x-wert (expired) zugewiesen wird. R-Wert-Rückgaben von Funktionen sind x-Werte.

2 C++ Advanced: Templates

Betrachte die folgende Funktion:

```
double sum(const std::vector<int>& v){
  double result = 0;
  for (auto x: v)
    result += x;
  return result;
}
```

Wir fragen uns nun, wie wir diese Funktion generischer machen können.

- Anderer Rückgabetyp
- Anderer Datentyp im Vektor
- Anderer Startwert
- Operation Anpassbar
- Datenstruktur anpassbar
- Eine Selektion der Daten

Für den letzten Punkt können wir ein Iterator-Paar übergeben.

Als laufendes Beispiel verwenden wir

```
class Pair{
  int left; int right;
public:
    Pair(int l, int r): left(l), right(r) {}

    int min() return left < right ? left : right;
};
```

Wir können nun vor der Definition der Klasse eine Template-Deklaration hinzufügen, um die Klasse generisch zu machen.

```
template<typename T>
class Pair{
    T left; T right;
public:
    Pair(T l, T r): left(l), right(r) {}
};
```

Dies nennt man Parametrischer Polymorphismus. Wenn die Klasse zu Ende ist hört dieses Typename automatisch auf. Es kann gelesen werden wie: Für alle Typen T.

Der Compiler generiert dann für jede Instanziierung der Klasse eine neue Klasse. Zum Beispiel wird die Klasse `Pair<int>` generiert, wenn wir `Pair<int> p(1,2)` schreiben.

Aktuell haben wir noch das Problem, dass wir nicht ein Pair von Pairs erstellen können. Der Compiler überprüft nur so wenig wie Nötig was potentiell zu Problemen führen kann.

Für Funktionen können wir ebenfalls Templates verwenden. Zum Beispiel könnte die Funktion

```
template<typename T>
T min(T l, T r){
    return l < r ? l : r;
```

```
}
```

Hierbei kann man auch die Funktion min<double> aufrufen, um die Funktion für double aufzurufen.

Bei unserer Implementation müssen nun aber beide Typen identisch sein. Hierfür können wir die Funktion wie folgt anpassen:

```
template<typename T1, typename T2>
auto min(T1 l, T2 r){
    return l < r ? l : r;
}
```

Diese Technik können wir auf unserem Pair Beispiel verwenden, um die Funktion min zu implementieren.

```
template<typename T>
class Pair{
    T left; T right;
public:
    Pair(T l, T r): left(l), right(r) {}
    auto min() return min(left, right);
};
```

Lec 2

Wenn der Typ bei der Instanzierung nicht inferiert werden kann, muss er explizit angegeben werden.

Eine sehr interessante Möglichkeit ist, dass so auch Funktionen als Template-Parameter übergeben werden können. Zum Beispiel könnte eine Funktion wie folgt übergeben werden

```
template <typename Container, typename Function>
void apply(Container& c, Function f){
    for (auto& x: c) f(x);
}
```

Wenn wir nun eine Funktion wie square haben, welche eine Zahl quadriert, dann können wir diese Funktion an apply übergeben. Dabei müssen wir aber den Typ der Funktion angeben, da er nicht inferiert werden kann.

Zusammenfassend können wir unser Eingangsbeispiel wie folgt anpassen:

```
template<typename InputIt, class T, class BinaryOp>
T accumulate(InputIt first, InputIt last, T init,
↪   BinaryOp op){
    for (; first != last; ++first) {
        init = op(init, *first);
    }
    return init;
}
```

Im allgemeinen ist es möglich das sich eine Funktionalität verbessern lässt als Spezialfall einer anderen Funktionalität. Man spricht von Spezialisierung Zum Beispiel könnten bools effizienter gespeichert werden mit

```
template <>
class Pair<bool>{
    short both;
public:
    Pair(bool l, bool r): both((l?1:0) | (r?2:0)) {}
    bool min() return (both & 1) != 0;
};
```

Templates können auch mit Werten parametrisiert werden, wobei der Wert aber zur Kompilierzeit bekannt sein muss. Zum Beispiel unterstützt die Eigen-Bibliothek die Definition von Matrizen mit einer festen Grösse, welche als Template-Parameter übergeben wird.

Um spezifische Anforderungen an die Template-Parameter zu stellen, können wir sogenannte Concepts verwenden. Zum Beispiel könnten wir verlangen, dass der Typ T einen Operator < definiert hat, damit wir die Funktion min verwenden können.

3 Suchen

Das Grundproblem ist das wir eine Menge von Datensätzen haben und jeder Datensatz hat einen Schlüssel. Hierbei sind Schlüssel vergleichbar. Das Ziel ist es nun, einen Datensatz mit einem bestimmten Schlüssel zu finden.

In einem unsortierten Array mit n Elementen $A_1, \ldots, A_n$ mit Schlüssel b Dann können wir einfach alle Elemente durchgehen und vergleichen, ob der Schlüssel von A_i gleich b ist. Im schlimmsten Fall müssen wir alle n Elemente durchgehen, also ist die Laufzeit dieses Algorithmus $\Theta(n)$.

Bei gleichwahrscheinlicher Verteilung der Schlüssel ist der Erwartungswert der Anzahl der Vergleiche

$$\mathbb{E}(X) = \sum_{i=1}^{n} i \cdot \frac{1}{n} = \frac{n+1}{2}.$$

Nehmen wir nun an, dass die Elemente aufsteigend sortiert sind. Dann können wir das Array in der Mitte anschauen und wissen dann, ob wir in der linken oder rechten Hälfte weitersuchen müssen. Dies können wir dann rekursiv machen, bis wir das gesuchte Element gefunden haben oder bis die Suche fehlschlägt. Die Laufzeit dieses Algorithmus ist $\Theta(\log n)$.

Algorithm 3.1: Binary Search

Gegeben sei ein sortiertes Array A mit n Elementen und ein Schlüssel b. Wir wollen wissen, ob es ein Element in A gibt, dessen Schlüssel gleich b ist.

Der Algorithmus benötigt im schlechtesten Fall $\Theta(\log n)$ Elementarschritte.

Algorithm 1: Binary Search Algorithmus

Input: Aufsteigend sortiertes Array A mit n Schlüsseln, Schlüssel b
Output: Index $i \in [1, n]$ mit $A[i] = b$, sonst 0
Funktion `BSearch`(A, n, b)**:**

```
if n = 0 then
    return 0 ;                    // erfolglose Suche
end
m ← ⌊(1 + n)/2⌋;
if b = A[m] then
    return m ;                    // gefunden
else
    if b < A[m] then
        return BSearch(A[1 ... m − 1], m − 1, b) ;
         // Element liegt links
    else
        k ← BSearch(A[m + 1 ... n], n − m, b);
         // Element liegt rechts
        if k = 0 then
            return 0
        end
        return m + k
    end
end
```

Es stellt sich die frage ob es einen vergleichsbasierten Suchalgorithmus gibt, welcher schneller als $\Theta(\log n)$ ist.

Wir können für einen Vergleichsbasierten Algorithmus einen Entscheidungsbaum betrachten, welcher die Vergleiche darstellt, welche der Algorithmus durchführt. Der Baum muss jedes Objekt in einem Blatt repräsentieren, folglich muss der Baum mindestens n Knoten haben.

Doch ein binärer Baum mit Höhe h hat höchstens 2^h Blätter, also muss $2^h \geq n$ gelten. Folglich muss $h \geq \log_2 n$ gelten, also ist die Laufzeit jedes vergleichsbasierten Suchalgorithmus mindestens $\Theta(\log n)$.

Theorem 3.2:

Jeder vergleichsbasierte Suchalgorithmus benötigt im schlechtesten Fall $\Omega(\log n)$ Vergleiche für *sortierte* Elemente.

In einem unsortierten Array mit n Elementen ist die Anzahl der Vergleiche im schlechtesten Fall $\Omega(n)$, da wir alle Elemente durchgehen müssen, um sicher zu sein, dass das gesuchte Element nicht vorhanden ist.

Theorem 3.3:

Jeder vergleichsbasierte Suchalgorithmus benötigt im schlechtesten Fall $\Omega(n)$ Vergleiche, wenn die Elemente *unsortiert* sind.

4 Auswählen

Gegeben sei ein unsortiertes Array mit paarweise verschiedenen Elementen. Wir wollen nun das k-t kleinste Element in diesem Array finden. Also das Element, sodass genau $k-1$ Elemente kleiner sind als dieses Element.

Ein erster naiver Algorithmus wäre wiederholt das Minimum zu entfernen. Dann hätten wir den Median in $\Theta(n^2)$ gefunden.

Einfacherweise können wir das Minimum und Maximum in $2n$ Vergleichen finden. Jedoch geht es auch in $\frac{3n}{2}$ Vergleichen, indem wir die Elemente paarweise vergleichen und das kleinere Element mit dem Minimum vergleichen und das grössere Element mit dem Maximum vergleichen.

Somit haben wir noch 3 Vergleiche für nur 2 Elemente, also $\frac{3n}{2}$ Vergleiche für n Elemente.

Dies weist darauf hin, dass wir das k-t kleinste Element effizienter finden können.

Die Idee ist zu PIVOTIEREN. Dazu wählen wir ein Element p aus dem Array aus und partitionieren das Array in drei Teile: die Elemente welche kleiner als p sind, die Elemente welche gleich p sind und die Elemente welche grösser als p sind. Je nachdem, wie viele Elemente kleiner als p sind, können wir entscheiden, ob wir das k-t kleinste Element in der linken Partition, der mittleren Partition oder der rechten Partition suchen müssen. Dann wenden wir Rekursion auf den entsprechenden Teil an.

Lec 3

Algorithm 2: Partitionierung um Pivot p (Hoare-Schema)

Input: Array A, das den Pivot p mindestens einmal enthält
Output: Index $k \in [1,n]$, so dass $A_i \leq p$ für $i \in [1,k]$ und $A_i \geq p$ für $i \in (k,n]$; Rückgabe von k

```
l ← 1; r ← n;
while true do
    while A[l] < p do
        l ← l + 1;
    end
    while A[r] > p do
        r ← r − 1;
    end
    if l ≥ r then
        return r;
    end
    tausche A[l] und A[r];
    l ← l + 1;
    r ← r − 1;
end
```

Die Korrektheit des Algorithmus lässt sich durch 3 Invarianten zeigen:

1. Alle Elemente in $A[1 \dots l-1]$ sind kleiner gleich p.
2. Alle Elemente in $A[r+1 \dots n]$ sind grösser gleich p.
3. Die Elemente in $A[l \dots r]$ sind noch nicht untersucht.

Zuletzt terminiert der Algorithmus, da im letzten Schritt l und r um 1 verschoben werden, wodurch die Anzahl der Elemente in $A[l \dots r]$ um 2 reduziert wird.

Um den Pivot an die richtige Stelle zu bringen, können wir den Algorithmus erweitern, indem wir die Stelle des Pivots speichern und am Ende des Algorithmus den Pivot mit dem Element an der Stelle l oder r tauschen.

Der eigentliche Auswahlalgorithmus sieht nun wie folgt aus:

Algorithm 3: Quickselect zur Bestimmung des k-kleinsten Elements

Input: Array A der Länge n, Index $1 \leq k \leq n$
Output: Wert $p \in A$ mit $|\{i : A[i] < p\}| < k$ und $|\{i : A[i] \leq p\}| \geq k$

```
Funktion Select(A, n, k):
    if n = 1 then
        return A[1];
    end
    Wähle ein Pivot p aus A;
    m ← Partition(A, p);
    if k < m then
        return Select(A[1 ... m − 1], m − 1, k);
    else
        if k > m then
            return
             Select(A[m + 1 ... n], n − m, k − m);
        else
            return p;
        end
    end
```

Die Partition braucht hierbei $\Theta(n)$ Zeit, da jedes Element genau einmal untersucht wird. Wenn wir optimistisch sind, so ist

$$T(n) = T(n/2) + cn = cn + c \cdot \frac{n}{2} + c \cdot \frac{n}{4} + \dots \leq 2cn = \Theta(n).$$

Im schlechtesten Fall, wenn der Pivot immer das kleinste oder grösste Element ist, haben wir jedoch $\Theta(n^2)$, da wir dann immer nur ein Element aus dem Array entfernen. Wir wollen nun unser Pivot-Glück verbessern. Ein guter Pivot hat linear viele Elemente auf beiden Seiten. Wenn wir die Unterteilung mit einem Faktor q haben so ist

$$\begin{aligned} T(n) &\leq T(qn) + cn \leq T(q^2 n) + cn + cqn \\ &\leq T(q^k n) + cn \sum_{i=0}^{k-1} q^i \\ &\leq T(q^k n) + cn \cdot \frac{1}{1-q} \\ &\leq T(1) + cn \cdot \frac{1}{1-q} = \Theta(n). \end{aligned}$$

Der Zufall hilft uns hier. Im Erwartungswert erhalten wir einen guten Pivot nach 2 mal pivotieren.

Wenn wir nun immer in linearer Zeit einen Pivot finde möchten, können wir den Median der Mediane Algorithmus verwenden. Dabei teilen wir das Array in Gruppen von 5 Elementen auf, berechnen den Median jeder Gruppe und speichern diese Mediane in einem neuen Array. Dann berechnen wir erneut die Mediane dieses neuen Arrays, bis wir nur noch einen Median übrig haben. Dieser Median benutzen wir als Pivot für das Vorherige Array. Somit

finden wir einen neuen Median. Dieser neue Median ist ein guter Pivot, da er garantiert, dass mindestens 30% der Elemente kleiner und mindestens 30% der Elemente grösser sind als dieser Pivot. Dieser Pivot wird dann rekursiv berechnet, was zu einer Laufzeit von $\Theta(n)$ führt.

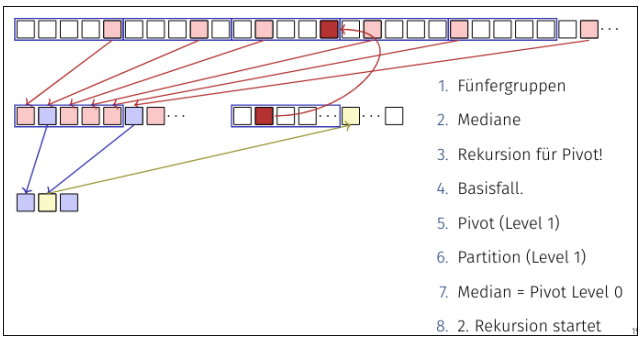


Figure 2: Median der Mediane Algorithmus

Implementiert wird der Median der Mediane Algorithmus wie folgt:

Algorithm 4: Median-of-Medians

Input: Array A der Länge n
Output: Median der Mediane
Funktion `MMSelect`(A, n)**:**
  if $n \leq 5$ **then**
    return `MedianOfFive`(A);
  end
  $A' \leftarrow$ `MedianOfFiveArray`(A);
  return `MMSelect`$(A', \left\lfloor \frac{|A'|+1}{2} \right\rfloor)$;

Hierbei ist `MMSelect` die Quickselect Funktion, welche als Pivot den Median der Mediane verwendet.

Theorem 4.1:

Das k-t kleinste Element in einem Array der Länge n kann in $\Theta(n)$ Zeit gefunden werden.

5 Sortieren

Gegeben ist ein Array der Länge n. Wir wollen eine Permutation A' dieses Arrays finden, so dass A' aufsteigend sortiert ist.

Ein erster naiver Algorithmus wäre sogenanntes Selection Sort. Dabei gehen wir durch das Array und suchen das Minimum, tauschen es mit dem ersten Element und entfernen es dann aus dem Array. Dann suchen wir das Minimum des verbleibenden Arrays, tauschen es mit dem zweiten Element und so weiter.

Die Invariante dieses Algorithmus ist, dass die ersten i Elemente des Arrays die i kleinsten Elemente in aufsteigender Reihenfolge enthalten.

Dieser Algorithmus ist in $\Theta(n^2)$, da wir n mal das Minimum suchen müssen, was jeweils $\Theta(n)$ Zeit benötigt. Es finden jedoch nur $n-1$ Vertauschungen statt, was $\Theta(n)$ ist.

Das Problem an diesem Algorithmus ist, dass wir alle verbleibenden Elemente durchgehen müssen, um das Minimum zu finden.

Ein anderer Algorithmus ist Insertion Sort. Dieser erinnert eher an das Sortieren von Jasskarten. Dabei gehen wir von links nach rechts durch das Array und nehmen das aktuelle Element und fügen es an der richtigen Stelle in die bereits sortierte Liste der vorherigen Elemente ein.

Die Invariante dieses Algorithmus ist, dass die ersten i Elemente des Arrays sortiert sind. Dieser Algorithmus macht im schlechtesten Fall $\Theta(n^2)$ Vertauschungen/Verschiebungen, da jedes Element im schlimmsten Fall an den Anfang des Arrays verschoben werden muss. Die Anzahl Vergleiche ist ebenfalls $\Theta(n^2)$, da jedes Element mit allen vorherigen Elementen verglichen werden muss. Wir können diesen Algorithmus jedoch verbessern, indem wir die richtige Stelle für das aktuelle Element mit einer binären Suche finden. Dadurch reduzieren wir die Anzahl der Vergleiche auf $\Theta(n \log n)$, während die Anzahl der Vertauschungen/Verschiebungen weiterhin $\Theta(n^2)$ bleibt.

Index