

19. Dynamische Datenstrukturen II

Verkettete Listen, Vektoren als verkettete Listen

Anderes Speicherlayout: Verkettete Liste

- *Kein* zusammenhängender Speicherbereich und *kein* wahlfreier Zugriff
- Jedes Element zeigt auf seinen Nachfolger
- Einfügen und Löschen *beliebiger* Elemente ist einfach

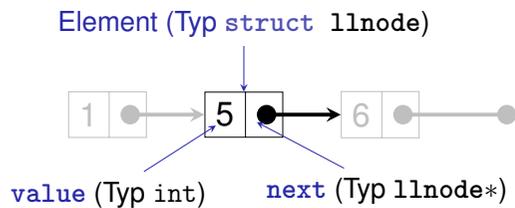


⇒ Unser Vektor kann als verkettete Liste realisiert werden

648

650

Verkettete Liste: Zoom

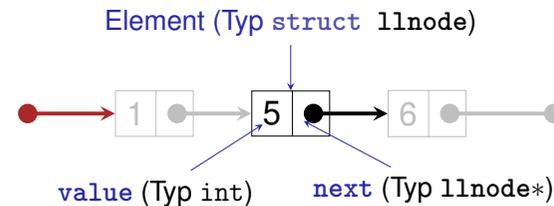


```
struct llnode {
    int value;
    llnode* next;

    llnode(int v, llnode* n): value(v), next(n) {} // Constructor
};
```

651

Vektor = Zeiger aufs erste Element



```
class llvec {
    llnode* head;
public:
    // Public interface identical to avec's
    llvec(unsigned int size);
    unsigned int size() const;
    ...
};
```

652

Funktion `llvec::print()`

```
struct llnode {  
    int value;  
    llnode* next;  
    ...  
};
```

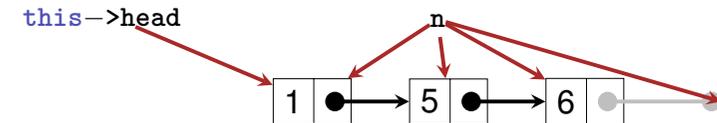
```
void llvec::print(std::ostream& sink) const {  
    for (llnode* n = this->head;   
         n != nullptr;   
         n = n->next) {  
        sink << n->value << ' ';  
    }  
}
```

← Zeiger auf erstes Element
← Abbruch falls Ende erreicht
← Zeiger elementweise voranschreiten
← Aktuelles Element ausgeben

653

Funktion `llvec::print()`

```
void llvec::print(std::ostream& sink) const {  
    for (llnode* n = this->head;  
         n != nullptr;  
         n = n->next)  
    {  
        sink << n->value << ' '; // 1 5 6  
    }  
}
```



654

Funktion `llvec::operator []`

Zugriff auf i -tes Element ähnlich implementiert wie `print()`:

```
int& llvec::operator [] (unsigned int i) {  
    llnode* n = this->head;  
  
    for (; 0 < i; --i)   
        n = n->next;  
  
    return n->value;  
}
```

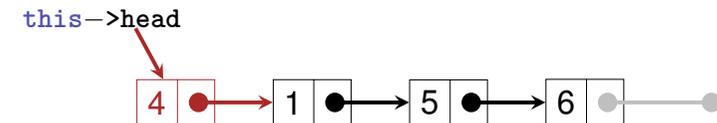
← Zeiger auf erstes Element
← Bis zum i -ten voranschreiten
← i -tes Element zurückgeben

655

Funktion `llvec::push_front()`

Vorteil `llvec`: Elemente am Anfang anfügen ist sehr einfach:

```
void llvec::push_front(int e) {  
    this->head =  
        new llnode{e, this->head};  
}
```



Achtung: Wäre der neue `llnode` nicht *dynamisch* alloziert, dann würde er am Ende von `push_front` sofort wieder gelöscht (= Speicher dealloziert)

656

Funktion `llvec::llvec()`

Konstruktor kann mittels `push_front()` implementiert werden:

```
llvec::llvec(unsigned int size) {  
    this->head = nullptr; ← head zeigt zunächst ins Nichts  
  
    for (; 0 < size; --size) | ← size mal 0 vorne anfügen  
        this->push_front(0);  
}
```

Anwendungsbeispiel:

```
llvec v = llvec(3);  
std::cout << v; // 0 0 0
```

657

Funktion `llvec::push_back()`

Einfach, aber ineffizient: Verkettete Liste bis ans Ende traversieren und neues Element anhängen

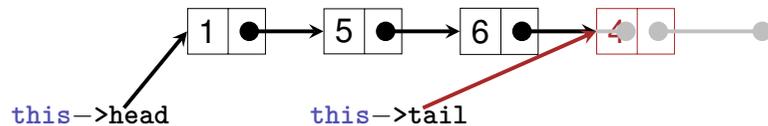
```
void llvec::push_back(int e) {  
    llnode* n = this->head; ← Beim ersten Element beginnen ...  
                                ... und bis zum letzten Element gehen  
  
    for (; n->next != nullptr; n = n->next); ←  
  
    n->next =  
        new llnode{e, nullptr}; ← Neues Element an bisher  
                                letztes anhängen  
}
```

658

Funktion `llvec::push_back()`

■ Effizienter, aber auch etwas komplexer:

- 1 Zweiter Zeiger, der auf das letzte Element zeigt: `this->tail`
- 2 Mittels dieses Zeigers kann direkt am Ende angehängt werden



■ **Aber:** Verschiedene Grenzfälle, z.B. Vektor noch leer, müssen beachtet werden

659

Funktion `llvec::size()`

Einfach, aber ineffizient: Grösse durch abzählen *berechnen*

```
unsigned int llvec::size() const {  
    unsigned int c = 0; ← Zähler initial 0  
  
    for (llnode* n = this->head; | ← Länge der Kette abzählen  
         n != nullptr; n = n->next)  
        ++c;  
  
    return c; ← Zähler zurückgeben  
}
```

660

Funktion `llvec::size()`

Effizienter, aber etwas komplexer: Grösse als Membervariable *nachhalten*

- 1 Membervariable `unsigned int` `count` zur Klasse `llvec` hinzufügen
- 2 `this->count` muss nun bei *jeder* Operation, die die Grösse des Vektors verändert (z.B. `push_front`), aktualisiert werden

20. Container, Iteratoren und Algorithmen

Container, Mengen, Iteratoren, const-Iteratoren, Algorithmen, Templates

Effizienz: Arrays vs. Verkettete Listen

- Speicher: Unser `avec` belegt ungefähr n ints (Vektorgrösse n), unser `llvec` ungefähr $3n$ ints (ein Zeiger belegt i.d.R. 8 Byte)
- Laufzeit (mit `avec = std::vector`, `llvec = std::list`):

```
prepending (insert at front) [100,000x]:
  ▶ avec: 675 ms
  ▶ llvec: 10 ms
appending (insert at back) [100,000x]:
  ▶ avec: 2 ms
  ▶ llvec: 9 ms
removing first [100,000x]:
  ▶ avec: 675 ms
  ▶ llvec: 4 ms
removing last [100,000x]:
  ▶ avec: 0 ms
  ▶ llvec: 4 ms

removing randomly [10,000x]:
  ▶ avec: 3 ms
  ▶ llvec: 113 ms
inserting randomly [10,000x]:
  ▶ avec: 16 ms
  ▶ llvec: 117 ms
fully iterate sequentially (5000 elements) [5,000x]:
  ▶ avec: 354 ms
  ▶ llvec: 525 ms
```

661

662

Vektoren sind Container

- Abstrakt gesehen ist ein Vektor
 - 1 Eine Ansammlung von Elementen
 - 2 Plus Operationen auf dieser Ansammlung
- In C++ heissen `vector<T>` und ähnliche „Ansammlungs“-Datenstrukturen *Container*
- In manchen Sprachen, z.B. Java, *Collections* genannt

663

664

Container-Eigenschaften

- Jeder Container hat bestimmte *charakteristische Eigenschaften*
- Ein Array-basierter Vektor z.B. die folgenden:
 - Effizienter, index-basierter Zugriff (`v[i]`)
 - Effiziente Speichernutzung: Nur die Elemente selbst belegen Platz (plus Elementezähler)
 - Einfügen/Entfernen an beliebigem Index ist potenziell ineffizient
 - Suchen eines bestimmten Elements ist potenziell ineffizient
 - Kann Elemente mehrfach enthalten
 - Elemente sind in Einfügereihenfolge enthalten (geordnet aber unsortiert)

665

Container in C++

- Fast jede Anwendung erfordert die Verwaltung und Manipulation von beliebig vielen Datensätzen
- Aber mit unterschiedlichen Anforderungen (z.B. Elemente nur hinten anhängen, fast nie entfernen, oft suchen, ...)
- Deswegen enthält die Standardbibliothek von C++ diverse Container mit unterschiedlichen Eigenschaften, siehe <https://en.cppreference.com/w/cpp/container>
- Viele weitere sind über Bibliotheken Dritter verfügbar, z.B. https://www.boost.org/doc/libs/1_68_0/doc/html/container.html, <https://github.com/abseil/abseil-cpp>

666

Beispiel-Container: `std::unordered_set<T>`

- Eine *mathematische Menge* ist eine ungeordnete, duplikatfreie Zusammenfassung von Elementen:

```
{1, 2, 1} = {1, 2} = {2, 1}
```

- In C++: `std::unordered_set<T>`
- Eigenschaften:
 - Kann kein Element doppelt enthalten
 - Elemente haben keine bestimmte Reihenfolge
 - Kein indexbasierter Zugriff (`s[i]` nicht definiert)
 - Effiziente „Element enthalten?“-Prüfung
 - Effizientes Einfügen und Löschen von Elementen
- Randbemerkung: Implementiert als Hash-Tabelle

667

Anwendungsbeispiel `std::unordered_set<T>`

Problem:

- Gegeben eine Sequenz an Paaren (*Name, Prozente*) von Code-Expert-Submissions ...

```
// Input: file submissions.txt
Friedrich 90
Schwerhoff 10
Lehner 20
Schwerhoff 11
```

- ... bestimme die Abgebenden, die mindestens 50% erzielt haben

```
// Output
Friedrich
```

668

Anwendungsbeispiel `std::unordered_set<T>`

```
std::ifstream in("submissions.txt"); ← Öffne submissions.txt
std::unordered_set<std::string> names; ← Namen-Menge, initial leer

std::string name;
unsigned int score;

while (in >> name >> score) {
    if (50 <= score)
        names.insert(name);
}

std::cout << "Unique submitters: "
          << names << '\n';
```

Paar (Name, Punkte)

Nächstes Paar einlesen
Namen merken falls Punkte
ausreichen

Gemerkte Namen ausgeben

669

Beispiel-Container: `std::set<T>`

- Fast gleich wie `std::unordered_set<T>`, aber die Elemente sind *geordnet*
 $\{1, 2, 1\} = \{1, 2\} \neq \{2, 1\}$
- Elemente suchen, einfügen und löschen weiterhin effizient (besser als bei `std::vector<T>`), aber weniger effizient als bei `std::unordered_set<T>`
- Denn das Beibehalten der Ordnung zieht etwas Aufwand nach sich
- Randbemerkung: Implementiert als Rot-Schwarz-Baum

670

Anwendungsbeispiel `std::set<T>`

```
std::ifstream in("submissions.txt");
std::set<std::string> names; ← set statt unordered_set ...

std::string name;
unsigned int score;

while (in >> name >> score) {
    if (50 <= score)
        names.insert(name);
}

std::cout << "Unique submitters: "
          << names << '\n';
```

... und die Ausgabe erfolgt
alphabetisch sortiert

671

Container Ausgeben

- Bereits gesehen: `avec::print()` und `llvec::print()`
- Wie sieht's mit der Ausgabe von `set`, `unordered_set`, ... aus?
- Gemeinsamkeit: Über Container-Elemente iterieren und diese ausgeben

672

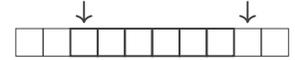
Ähnliche Funktionen

- Viele weitere nützliche Funktionen können mittels Container-Iteration implementiert werden:
- `contains(c, e)`: wahr gdw. Container `c` Element `e` enthält
- `min/max(c)`: Gibt das grösste/kleinste Element zurück
- `sort(c)`: Sortiert die Elemente von `c`
- `replace(c, e1, e2)`: Ersetzt alle `e1` in `c` mit `e2`
- `sample(c, n)`: Wählt zufällig `n` Elemente aus `c` aus
- ...

673

Zur Erinnerung: Iterieren mit Zeigern

- Iteration über ein *Array*:
 - Auf Startelement zeigen: `p = this->arr`
 - Auf aktuelles Element zugreifen: `*p`
 - Überprüfen, ob Ende erreicht: `p == p + size`
 - Zeiger vorrücken: `p = p + 1`
- Iteration über eine *verkettete Liste*:
 - Auf Startelement zeigen: `p = this->head`
 - Auf aktuelles Element zugreifen: `p->value`
 - Überprüfen, ob Ende erreicht: `p == nullptr`
 - Zeiger vorrücken: `p = p->next`



674

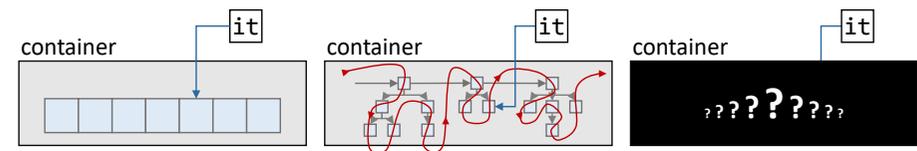
Iteratoren

- Iteration erfordert nur die vier eben gesehenen Operationen
- Aber deren Implementierung hängt vom Container ab
- \Rightarrow Jeder C++-Container implementiert seinen eigenen *Iterator*
- Geben ein Container `c`:
 - `it = c.begin()`: Iterator aufs erste Element
 - `it = c.end()`: Iterator *hinters* letzte Element
 - `*it`: Zugriff aufs aktuelle Element
 - `++it`: Iterator um ein Element verschieben
- Iteratoren sind quasi gepimpte Zeiger

675

Iteratoren

- Iteratoren ermöglichen Zugriff auf verschiedene Container auf *uniforme* Weise: `*it`, `++it`, etc.
- Nutzer bleiben unabhängig von der Container-Implementierung
- Iterator weiss, wie man die Elemente „seines“ Containers abläuft
- Nutzer müssen und sollen interne Details nicht kennen
- \Rightarrow Containerimplementierung kann geändert werden, ohne das Nutzer Code ändern müssen



676

Beispiel: Iteration über std::vector

```
std::vector<int> v = {1, 2, 3};

for (std::vector<int>::iterator it = v.begin();
     it != v.end(); ++it) {
    *it = -*it;
}

std::cout << v; // -1 -2 -3
```

it ist ein zu std::vector<int> passender Iterator

it zeigt initial aufs erste Element

Abbruch falls it Ende erreicht hat

it elementweise vorwärtsetzen

Aktuelles Element negieren (e → -e)

677

Beispiel: Iteration über std::vector

Zur Erinnerung: Type-Aliasse können genutzt werden um oft genutzte Typnamen abzukürzen

```
using ivit = std::vector<int>::iterator; // int-vector iterator

for (ivit it = v.begin();
     ...
```

678

Negieren als Funktion

```
void neg(std::vector<int>& v) {
    for (std::vector<int>::iterator it = v.begin();
         it != v.end(); ++it) {
        *it = -*it;
    }
}

// in main():
std::vector<int> v = {1, 2, 3};
neg(v); // v = {-1, -2, -3}
```

Nachteil: Negiert immer den gesamten Vektor

679

Negieren als Funktion

Besser: Innerhalb eines bestimmten *Bereichs (Intervalls)* negieren

```
void neg(std::vector<int>::iterator begin;
         std::vector<int>::iterator end) {
    for (std::vector<int>::iterator it = begin;
         it != end; ++it) {
        *it = -*it;
    }
}
```

Elemente im Intervall [begin, end) negieren

680

Negieren als Funktion

Besser: Innerhalb eines bestimmten *Bereichs* (*Intervalls*) negieren

```
void neg(std::vector<int>::iterator start;
        std::vector<int>::iterator end);

// in main():
std::vector<int> v = {1, 2, 3};
neg(v.begin(), v.begin() + (v.size() / 2)); ← Erste Hälfte negieren
```

681

Algorithmen-Bibliothek in C++

- Die C++-Standardbibliothek enthält viele nützliche Algorithmen (Funktionen), die auf durch Iteratoren bestimmten Intervallen [*Anfang*, *Ende*) arbeiten
- Zum Beispiel `find`, `fill` und `sort`
- Siehe auch <https://en.cppreference.com/w/cpp/algorithm>

682

Ein Iterator für `llvec`

Wir brauchen:

1. Einen `llvec`-spezifischen Iterator mit mindestens folgender Funktionalität:
 - Zugriff aktuelles Element: `operator*`
 - Iterator vorwärtssetzen: `operator++`
 - Ende-Erreicht-Prüfung: `operator!=` (oder `operator==`)
2. Memberfunktionen `begin()` und `end()` für `llvec` um einen Iterator auf den Anfang bzw. hinter das Ende zu erhalten

683

Iterator `avec::iterator` (Schritt 1/2)

```
class llvec {
    ...
public:
    class iterator {
        ...
    };

    ...
}
```

- Der Iterator gehört zu unserem Vektor, daher ist `iterator` eine öffentliche *innere Klasse* von `llvec`
- Instanzen unseres Iterators sind vom Typ `llvec::iterator`

684

Iterator `llvec::iterator` (Schritt 1/2)

```
class iterator {
  llnode* node; ← Zeiger auf aktuelles Vektor-Element

public:
  iterator(llnode* n); ← Erzeuge Iterator auf bestimmtes Element
  iterator& operator++(); ← Iterator ein Element vorwärtssetzen
  int& operator*() const; ← Zugriff auf aktuelles Element
  bool operator!=(const iterator& other) const; ← Vergleich mit anderem Iterator
};
```

685

Iterator `llvec::iterator` (Schritt 1/2)

```
// Constructor
llvec::iterator::iterator(llnode* n): node(n) {} ← Iterator initial auf n zeigen lassen

// Pre-increment
llvec::iterator& llvec::iterator::operator++() {
  assert(this->node != nullptr);

  this->node = this->node->next; ← Iterator ein Element vorwärtssetzen

  return *this; ← Referenz auf verschobenen Iterator zurückgeben
}
```

686

Iterator `llvec::iterator` (Schritt 1/2)

```
// Element access
int& llvec::iterator::operator*() const {
  return this->node->value; ← Zugriff auf aktuelles Element
}

// Comparison
bool llvec::iterator::operator!=(const llvec::iterator& other)
  const {
  return this->node != other.node; ← this Iterator verschieden von other falls sie auf unterschiedliche Elemente zeigen
}
```

687

Ein Iterator für `llvec` (Wiederholung)

Wir brauchen:

- 1 Einen `llvec`-spezifischen Iterator mit mindestens folgender Funktionalität:
 - Zugriff aktuelles Element: `operator*`
 - Iterator vorwärtssetzen: `operator++`
 - Ende-Erreicht-Prüfung: `operator!=` (oder `operator==`)
- 2 Memberfunktionen `begin()` und `end()` für `llvec` um einen Iterator auf den Anfang bzw. hinter das Ende zu erhalten



688

Iterator `avec::iterator` (Schritt 2/2)

```
class l1vec {
    ...
public:
    class iterator {...};

    iterator begin();
    iterator end();

    ...
}
```

`l1vec` braucht Memberfunktionen um Iteratoren *auf den Anfang* bzw. *hinter das Ende* des Vektors herausgeben zu können

689

Iterator `l1vec::iterator` (Schritt 2/2)

```
l1vec::iterator l1vec::begin() {
    return l1vec::iterator(this->head);
}
// Iterator auf erstes Vektorelement

l1vec::iterator l1vec::end() {
    return l1vec::iterator(nullptr);
}
// Iterator hinter letztes Vektorelement
```

690

Const-Iteratoren

- Neben `iterator` sollte jeder Container auch einen *Const-Iterator* `const_iterator` bereitstellen
- Const-Iteratoren gestatten nur Lesezugriff auf den darunterliegenden Container
- Zum Beispiel für `l1vec`:

```
l1vec::const_iterator l1vec::cbegin() const;
l1vec::const_iterator l1vec::cend() const;

const int& l1vec::const_iterator::operator*() const;
...
```

- Daher nicht möglich (Compilerfehler): `*(v.cbegin()) = 0`

691

Const-Iteratoren

Const-Iterator *kann* verwendet werden um nur Lesen zu erlauben:

```
l1vec v = ...;
for (l1vec::const_iterator it = v.cbegin(); ...)
    std::cout << *it;
```

Hier könnte auch der nicht-const `iterator` verwendet werden

692

Const-Iteratoren

Const-Iterator *muss* verwendet werden falls Vektor selbst const ist:

```
const llvec v = ...;
for (llvec::const_iterator it = v.cbegin(); ...)
    std::cout << *it;
```

Hier kann nicht der `iterator` verwendet werden (Compilerfehler)

693

Exkurs: Templates

- **Ziel:** Ein *generischer* Ausgabe-Operator `<<` für *iterierbare Container*: `llvec`, `avec`, `std::vector`, `std::set`, ...
- D.h. `std::cout << c << '\n'` soll für jeden solchen Container `c` funktionieren

694

Exkurs: Templates

Templates ermöglichen *Typ-generische* Funktionen und Klassen:

- Templates ermöglichen die Nutzung von *Typen als Argumenten*

```
template <typename S, typename C>
S& operator<<(S& sink, const C& container);
```

Die spitzen Klammern kennen wir schon von `std::vector<int>`. Vektoren sind auch als Templates realisiert.

Intuition: Operator funktioniert für jeden Ausgabestrom `sink` vom Typ `S` und jeden Container `container` vom Typ `C`

695

Exkurs: Templates

Templates ermöglichen *Typ-generische* Funktionen und Klassen:

- Templates ermöglichen die Nutzung von *Typen als Argumenten*

```
template <typename S, typename C>
S& operator<<(S& sink, const C& container);
```

- Der Compiler *inferiert* passende Typen aus den Aufrufargumenten

```
std::set<int> s = ...;
std::cout << s << '\n';
```

← `S = std::ostream, C = std::set<int>`

696

Exkurs: Templates

Implementierung von `<<` *schränkt S und C ein* (Compilerfehler falls nicht erfüllt):

```
template <typename S, typename C>
S& operator<<(S& sink, const C& container) {
    for (typename C::const_iterator it = container.begin();
         it != container.end();
         ++it) {
        sink << *it << ' ';
    }

    return sink;
}
```

C muss Iteratoren bereitstellen – mit passenden Funktionen

697

Exkurs: Templates

Implementierung von `<<` *schränkt S und C ein* (Compilerfehler falls nicht erfüllt):

```
template <typename S, typename C>
S& operator<<(S& sink, const C& container) {
    for (typename C::const_iterator it = container.begin();
         it != container.end();
         ++it) {
        sink << *it << ' ';
    }

    return sink;
}
```

S muss Ausgabe von Elementen (*it) und Zeichen (' ') unterstützen

698