Programmentwicklung II

Bachelor of Science

Prof. Dr. Rethmann / Prof. Dr. Brandt

Fachbereich Elektrotechnik und Informatik
Hochschule Niederrhein

Sommersemester 2021

Inhalt

C++ als aktuelle Programmiersprache für Betriebssysteme, eingebettete Systeme, virtuelle Maschinen, Treiber und Signalprozessoren:

- gute Grundlage für C#, Java, PHP, Perl oder andere Sprachen
- C++11: Unterstützung von Nebenläufigkeit durch Threads, Erweiterung der STL um z.B. reguläre Ausdrücke, intelligente Zeiger (smart pointer), ungeordnete assoziative Container, eine Zufallszahlenbibliothek, numerische und mathematische Bibliotheken
- aktuell C++20: STL um Filesystem erweitert, STL stellt 69 Algorithmen in sequentieller, paralleler und vektorisierender Variante zu Verfügung

UML – Unified Modeling Language
graphische Darstellung der Systemkomponenten

Entwurfsmuster
irgendwer hat Ihr (Entwurfs-)Problem schon gelöst

Client-Server Architekturen
Verteilte Systeme implementieren mit Sockets und Remote Procedure Calls

Auszug aus *The C++ programming language* von Bjarne Stroustrup:

- You don't have to know every detail of C++ to write good programs.
- Focus on programming techniques, not on language features.
- Don't reinvent the wheel, use libraries.
- Don't believe in magic: understand what your libraries do, how they do it, and at what cost they do it.
- Zero-overhead principle: What you don't use, you don't pay for. What you do use, you couldn't hand code better.

Konfuzius:

> Wer fragt, ist ein Narr für eine Minute.
> Wer nicht fragt, ist ein Narr sein Leben lang.

Stellen Sie Fragen! Nur so können wir beurteilen, ob Sie etwas verstanden haben oder noch im Trüben fischen.

Der Lernerfolg wird am Ende durch eine Klausur geprüft:

- In der Klausur steht Ihnen kein Computer, keine Online-Hilfe, kein Debugger und kein Compiler zur Verfügung.
- Die Klausursituation ist daher grundlegend anders als die Situation zu Hause oder in der Übung und muss geübt werden.

Bereiten Sie sich auf die Klausur vor, indem Sie Programme zunächst auf einem Blatt Papier entwickeln.

- Gehen Sie die einzelnen Programmschritte durch und vollziehen Sie dabei nach, ob das Programm korrekt ist.
- Implementieren Sie dann das Programm genau so, wie es auf dem Papier steht und kompilieren Sie es.
- Syntaxfehler beim nächsten Programm möglichst vermeiden!
- Nach dem Beseitigen der Syntaxfehler: Programm testen.
- Logische Fehler beim nächsten Programm vermeiden!

Bei der strukturierten bzw. prozeduralen Programmierung werden

- Funktionen und Prozeduren dazu benutzt, Programme zu organisieren, z.B. `sqrt`, `sin`, `printf`, `toString`, ...
- Oft gebrauchte Funktionen, Prozeduren und Daten werden in Bibliotheken zur Verfügung gestellt: `stdio.h`, `stdlib.h`, `string.h`, `math.h`, `time.h`, ...

Funktionen reduzieren Copy-and-Paste von Programmteilen enorm. Anstelle von

```
if ((d1.jahr > d2.jahr)
        || (d1.jahr == d2.jahr && d1.monat > d2.monat)
        || (d1.jahr == d2.jahr && d1.monat == d2.monat &&
              d1.tag > d2.tag)) {
    ...
}
```

würden wir die Logik eines Datumvergleichs in einer Funktion bereitstellen

```
bool isGreater(date_t a, date_t b) {
    return (a.jahr > b.jahr)
           || (a.jahr == b.jahr && a.monat > b.monat)
           || (a.jahr == b.jahr && a.monat == b.monat &&
                 a.tag > b.tag);
}
```

und an den jeweiligen Programmstellen die Funktion aufrufen:

```
if (isGreater(d1, d2)) {
    ...
}
```

→ das Programm wird lesbar: literarisches Programmieren

Literarisches Programmieren bezeichnet das Schreiben von Computerprogrammen in einer Form, sodass sie vor allem für Menschen lesbar sind.

Dies steht im Gegensatz zur konventionellen Ansicht, dass Programme hauptsächlich effizient sein sollen und dann oft nur noch für den Computer lesbar sind.

Jon Bentley fragte in *Communications of the ACM*: „When was the last time you spent a pleasant evening in a comfortable chair, reading a good program?“

aus: `https://de.wikipedia.org/wiki/Literate_programming`

Stellen wir die Funktion dann noch in einer Bibliothek bereit, kann die Funktion sogar projektübergreifend verwendet werden.

In C++ können wir das Ganze durch geeignete Operatorüberladung noch lesbarer schreiben: `if (d1 > d2) ...`

Ziele der strukturierten Programmierung:

- Verständlicher und übersichtlicher Code.
- Wartbarer und erweiterbarer Code.
- Wiederverwendung von Algorithmen.
- Wiederverwenden von Code durch allgemeingültige Funktionen oder Makros anstelle von Copy-and-Paste.

Problem: Typisierung

```
int ival[30];
int icmp(const void *, const void *);
...
void qsort(ival, 30, sizeof(int), icmp);
...
int icmp(const void *a, const void *b) {
    int x = *(int *) a;
    int y = *(int *) b;
    return x - y;
}
```

Lösung in C: Zeiger auf `void`, Zeiger auf Funktionen bzw. Makros
besser in C++: Templates, Vererbung, Polymorphismus

Problem: globale Variablen oder lange Parameterlisten

- Programme sind einfacher zu verstehen, wenn sie aus kleinen, in sich geschlossenen, unabhängigen Teilen bestehen.
  - Globale Variablen führen zu voneinander abhängigen Funktionen. Das Ändern einer Funktion kann dazu führen, dass andere Funktionen nicht mehr korrekt funktionieren. Nach jeder Änderung muss man erneut das ganze Programm testen.
  - Übergeben wir alle benötigten Variablen als Parameter an die Funktionen, ergeben sich lange, unklare Parameterlisten.
- Keine Zugriffskontrolle: Bei den heutigen nebenläufigen Programmen ist es wichtig, den gleichzeitigen Zugriff mehrerer Threads auf gemeinsame Variablen zu synchronisieren.
- Namenskonflikte: In umfangreichen Programmen wird oft derselbe Variablenname zweimal verwendet.

Lösung in C: Module, incomplete data type
besser in C++: Klassen, private/protected, Namensräume

Problem: Lesbarkeit und Wartbarkeit

- Vergleichsoperatoren bei allgemeinen Datentypen
  Lösung in C: Funktionen
  besser in C++: Operatorüberladung
- Fehlerbehandlung
  Lösung in C: Fehlerflags als Rückgabewert einer Funktion, globale Fehlervariable `errno`, Signal-Handler
  besser in C++: Exceptions

Gehen wir die Probleme an! Lernen wir mit C++ eine tolle Programmiersprache kennen.

Liste

Oft müssen wir eine Liste von Elementen verwalten:

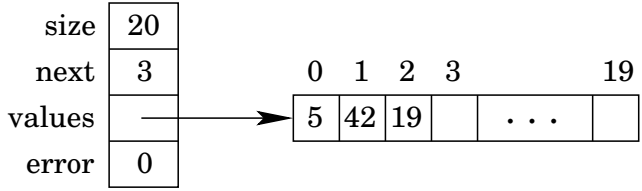
- Bücherliste in der Bibliothek
- Studentenliste im Prüfungsbüro
- Mitarbeiterliste in der Verwaltung
- KFZ-Liste im Straßenverkehrsamt
- ...

Die Anforderungen an solche Listen sind immer gleich:

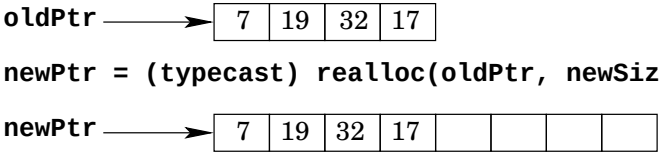
- hinzufügen von Werten
- löschen von Werten
- suchen (z.B. Halter des Fahrzeugs KR-AB 123)
- ausdrucken oder anzeigen der Liste
- ...

Einige Details zu unserer Implementierung:

- Die Liste beruht auf einem Array.
  - Das Array wird bei Bedarf automatisch vergrößert und
  - wird automatisch verkleinert, wenn so viele Elemente aus der Liste entfernt wurden, dass das Array nur noch zu einem Viertel gefüllt ist.
- Damit der Code wiederverwendet werden kann,
  - wurden alle wichtigen Variablen in einer Struktur zusammengefasst
  - und alle Operationen sind als Funktionen ausgeführt.
- Struktur:



Wiederholen wir zunächst kurz, wie ein dynamisch angelegtes Array vergrößert werden kann:



Konnte der alte Speicherbereich nicht vergrößert werden, dann wird neuer Speicherbereich allokiert, die alten Werte in den neuen Speicher kopiert und der alte Speicherbereich frei gegeben. In diesem Fall ist `oldPtr` nicht mehr gültig.

Oft soll unter dem gleichen Namen wie zuvor das Array weiterhin benutzt werden, dann ist `newPtr = oldPtr`:

```
int *dynArr = (int *) calloc(4, sizeof(int));
    ...
dynArr = (int *) realloc(dynArr, sizeof(int) * 8);
```

Wichtig: Unterscheide Variablen und Strukturattribute!

```
typedef struct {
    int wert;       // Strukturattribut
    char *name;     // Strukturattribut
} foo_t;            // Name des Typs

int main(void) {
    char *h = "Hallo, Welt!";
    foo_t a;        // Variable vom Typ foo_t

    a.wert = 15;  // Strukturattribut der Variablen
    a.name = (char *) malloc(strlen(h) + 1);
    strcpy(a.name, h);
    ...
}
```

Über eine Struktur kann ein Array gebildet werden!

```
typedef struct {
    int wert;       // Strukturattribut
    char *name;     // Strukturattribut
} foo_t;            // Name des Typs

int main(void) {
    char *h = "Hallo, Welt!";
    foo_t a[5];     // Array: 5x Typ foo_t

    a[0].wert = 15;
    a[0].name = (char *) malloc(strlen(h) + 1);
    strcpy(a[0].name, h);
    ...
}
```

Ein Array kann Zeiger auf Strukturen enthalten!

```
typedef struct {
    int wert;      // Strukturattribut
    char *name;    // Strukturattribut
} foo_t;           // Name des Typs

int main(void) {
    char *h = "Hallo, Welt!";
    foo_t *a[5];  // Array: 5x Zeiger auf foo_t

    a[0] = (foo_t *) malloc(sizeof(foo_t));
    a[0]->wert = 15;
    a[0]->name = (char *) malloc(strlen(h) + 1);
    strcpy(a[0]->name, h);
    ...
}
```

Arrays von Strukturen können auch dynamisch angelegt werden!

```
typedef struct {
    int wert;       // Strukturattribut
    char *name;     // Strukturattribut
} foo_t;            // Name des Typs

int main(void) {
    char *h = "Hallo, Welt!";
    foo_t *a;       // dynamisches Array oder Zeiger auf 1x foo_t

    a = (foo_t *) calloc(5, sizeof(foo_t));

    a[0].wert = 15;
    a[0].name = (char *) malloc(strlen(h) + 1);
    strcpy(a[0].name, h);
    ...
}
```

Dynamische Arrays können Zeiger auf Strukturen enthalten!

```
typedef struct {
    int wert;       // Strukturattribut
    char *name;     // Strukturattribut
} foo_t;            // Name des Typs

int main(void) {
    char *h = "Hallo, Welt!";
    foo_t **a;      // dynamisches Array mit Zeiger auf foo_t

    a = (foo_t **) calloc(5, sizeof(foo_t *));

    a[0] = (foo_t *) malloc(sizeof(foo_t));
    a[0]->wert = 15;
    a[0]->name = (char *) malloc(strlen(h) + 1);
    strcpy(a[0]->name, h);
    ...
}
```

In C macht es einen Unterschied, ob wir bei der Deklaration einer Funktion die Parameterliste leer lassen oder als `void` definieren.

```c
#include <stdio.h>

int t() {        // leere Parameterliste
    return 42;
}

int s(void) {    // Parameterliste als void definiert
    return 43;
}

int main(void) {
    printf("%d\n", t(3));
    printf("%d\n", t(4, 6.5));
    printf("%d\n", s(3));
    return 0;
}
```

Kompilieren mittels `gcc -Wall -Wextra -pedantic void.c` liefert die folgende Ausgabe:

```
void.c: In function ‘main’:
void.c:14: error: too many arguments to function ‘s’
     printf("%d\n", s(3));
                    ^
void.c:7: note: declared here
 int s(void) {
```

Ersetzen wir den Aufruf der Funktion `s` durch einen Aufruf ohne Parameter, liefert der Compiler keine Fehlermeldung!

Eine leere Parameterliste bei der Deklaration einer Funktion wird in C als beliebige Parameter vom beliebigen Typ interpretiert.

Liste: erster Versuch

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    int size, next, *values;
    char error;
} list_t;

list_t *create(void) {
    list_t *l;

    l = (list_t *) calloc(1, sizeof(list_t));
    l->size = 8;          // ^^^^^^^^^^^^^^ nicht vergessen
    l->next = 0;
    l->values = (int *) calloc(8, sizeof(int));
    l->error = 0;              // ^^^^^^^^^^^ nicht vergessen
    return l;
}
```

Liste: erster Versuch

```
char isFull(list_t *l) {
    return l->size == l->next;
}

void increase(list_t *l) {
    l->size *= 2;
    l->values = (int *) realloc(l->values,
            l->size * sizeof(int));
}                   // ^^^^^^^^^^^ nicht vergessen

void append(list_t *l, int val) {
    if (isFull(l))
        increase(l);

    l->values[l->next] = val;
    l->next += 1;
}
```

Liste: erster Versuch

```
int find(list_t *l, int val) {
    int pos;

    for (pos = 0; pos < l->next; pos++)
        if (l->values[pos] == val)
            return pos;
    return -1;
}

void decrease(list_t *l) {
    l->size /= 2;
    l->values = (int *) realloc(l->values,
            l->size * sizeof(int));
}                   // ^^^^^^^^^^^ nicht vergessen
```

Liste: erster Versuch

```
void erase(list_t *l, int val) {
    int pos = find(l, val);

    if (pos == -1)
        return;

    for (; pos < l->next -1; pos++)
        l->values[pos] = l->values[pos + 1];
    l->next -= 1;

    if (l->next < l->size / 4)
        decrease(l);
}
```

Liste: erster Versuch

```
int getValueAt(list_t *l, int pos) {
    if (pos < 0 || pos >= l->next) {
        l->error = 1;
        return -1;
    }
    return l->values[pos];
}

void destroy(list_t *l) {
    free(l->values);
    free(l);
}
```

Liste: erster Versuch

```
void toScreen(list_t *l) {
    for (int i = 0; i < l->next; i++) {
        if (i > 0)
            printf(", ");
        printf("%d", l->values[i]);
    }
    printf("\n");
}

int main(void) {
   int i;
   list_t *l;

   l = create();
   for (i = 1; i < 30; i++)
       append(l, i);
   toScreen(l);
```

Liste: erster Versuch

```
    for (i = 1; i < 30; i += 2)
        erase(l, i);
    toScreen(l);

    i = getValueAt(l, 20);
    if (l->error == 0)
        printf("value[%2d] = %2d\n", 20, i);
    else printf("20 out of range\n");

    destroy(l);
    return 0;
}
```

Frage: Was halten Sie von der Implementierung?

Aufteilen in Header- und Code-Datei

Programme würden zum Teil auch ohne Header-Dateien und ohne explizite Typumwandlungen funktionieren, wie an folgendem Beispiel ersichtlich ist:

```
void main(void) {
    int *p = malloc(sizeof(int));
    *p = 42;
    printf("Hallo, Welt!\nint: %d\n", *p);
    free(p);
}
```

Allerdings kann der Compiler dann nicht überprüfen, ob die Parameter und Rückgaben von Funktionen vom richtigen Typ sind. Der gcc liefert:

```
t.c: In function ‘main’:
t.c:2: warning: implicit declaration of function ‘malloc’
t.c:2: note: include ‘<stdlib.h>’ or provide declaration of ‘malloc’
t.c:4: warning: implicit declaration of function ‘printf’
t.c:4: note: include ‘<stdio.h>’ or provide declaration of ‘printf’
.....
```

Liste: zweiter Versuch

liste.h

```
typedef struct {
    int size, next, *values;
    char error;
} list_t;

list_t *create(void);
char isFull(list_t *l);
void increase(list_t *l);
void decrease(list_t *l);
void append(list_t *l, int val);
int find(list_t *l, int val);
int getValueAt(list_t *l, int pos);
void erase(list_t *l, int val);
void toScreen(list_t *l);
void destroy(list_t *l);
```

Liste: zweiter Versuch

liste.c

```
#include <stdio.h>
#include <stdlib.h>
#include "liste.h"

list_t *create(void) {
    list_t *l;

    l = (list_t *) calloc(1, sizeof(list_t));
    l->size = 8;
    l->next = 0;
    l->values = (int *) calloc(8, sizeof(int));
    l->error = 0;

    return l;
}

.....
```

Liste: zweiter Versuch

main.c

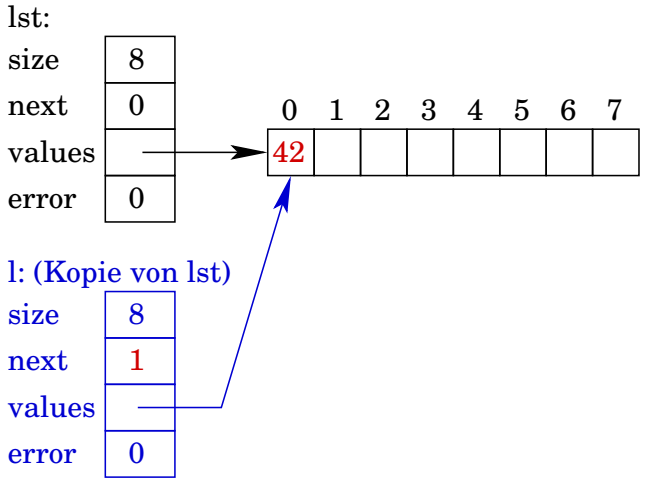
```c
#include <stdio.h>
#include "liste.h"

void main(void) {
    int i;
    list_t *l = create();

    for (i = 1; i < 30; i++)
        append(l, i);
    toScreen(l);

    i = getValueAt(l, 30);
    if (l->error != 0)
        printf("value[%2d] = %2d\n", 30, i);
    destroy(l);
}
```

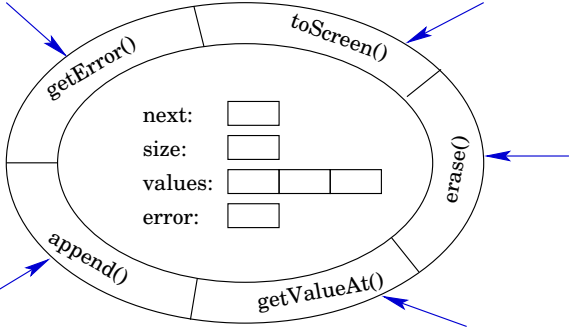
Warum wird die Liste als Zeiger übergeben? Weil sonst eine Kopie erzeugt würde, mit der die Funktionen wie `append(lst, 42)` dann arbeiten, und die Liste beim Aufrufer nicht geändert würde:



Liste: Verbesserungsmöglichkeiten

- Die Funktionen `increase` und `decrease` sind öffentlich bekannt, obwohl sie nur innerhalb der Liste verwendet werden und nicht von außen aufrufbar sein sollten.
- Der innere Aufbau der Liste ist durch das `typedef` in der Header-Datei nach außen hin bekannt.

⇒ Keine Datenkapselung! Im Hauptprogramm kann die Funktionalität der Liste durch einen schreibenden Zugriff wie `l->size = 0` zerstört werden.



Modulare Programmierung

Modulare Programmierung versucht der wachsenden Größe von Softwareprojekten Herr zu werden. Module können einzeln geplant, programmiert und getestet werden.

Universelle Module müssen nur einmal programmiert und können wiederverwendet werden. Je öfter ein Modul wiederverwendet wurde, desto sicherer kann man sein, dass es fehlerfrei ist.

Wenn alle Module erfolgreich getestet sind, können diese Einzelteile logisch miteinander verknüpft und zu einer größeren Anwendung zusammengesetzt werden.

Die modulare Programmierung erweitert den prozeduralen Ansatz, indem *Prozeduren zusammen mit Daten* in logischen Einheiten zusammengefasst werden.

aus: `https://de.wikipedia.org/wiki/Modulare_Programmierung`

In unserem Beispiel haben wir folgendes zu tun, um eine Liste als wiederverwendbares Modul zur Verfügung stellen zu können:

- In `liste.h` die Strukturvereinbarung entfernen und durch einen unvollständigen Typen (Vorwärtsdeklaration) ersetzen.
- In `liste.c` die aus `liste.h` entfernte Strukturvereinbarung aufnehmen.
- Wir müssen die Schnittstelle erweitern: Methoden zum Zugriff auf interne Variablen definieren, die im Hauptprogramm benötigt werden: `getError`
- Funktionen sind implizit immer als `extern` deklariert, ebenso wie globale Variablen, und sind daher von allen Programmstellen aus aufrufbar.
- Um Datenkapselung sicherzustellen, müssen wir mittels `static` die Funktionen „verstecken“, die nach außen nicht sichtbar sein sollen: `isFull`, `increase`, usw.

Liste: dritter Versuch

liste.h

```
// ====================================
// incomplete data type
// (forward declaration)
// ====================================
typedef struct list list_t;

// ====================================
// interface
// ====================================
list_t *create(void);
void append(list_t *l, int val);
int getValueAt(list_t *l, int pos);
void erase(list_t *l, int val);
void toScreen(list_t *l);
char getError(list_t *l);     // neu !!!
void destroy(list_t *l);
```

liste.c

```
#include ....

struct list {
    int size, next, *values;
    char error;
};

list_t *create() {
    list_t *l;

    l = (list_t *) calloc(1, sizeof(list_t));
    l->size = 8;
    l->next = 0;
    l->values = (int *) calloc(8, sizeof(int));
    l->error = 0;

    return l;
}
```

Liste: dritter Versuch

```
// private !
static void increase(list_t *l) {
    l->size *= 2;
    l->values = (int *) realloc(l->values,
            l->size * sizeof(int));
}

// private !
static void decrease(list_t *l) {
    l->size /= 2;
    l->values = (int *) realloc(l->values,
            l->size * sizeof(int));
}

// private !
static char isFull(list_t *l) {
    return l->size == l->next;
}
```

Liste: dritter Versuch

```
void append(list_t *l, int val) {
    if (isFull(l))
        increase(l);

    l->values[l->next] = val;
    l->next += 1;
}

// private !
static int find(list_t *l, int val) {
    int pos;

    for (pos = 0; pos < l->next; pos++)
        if (l->values[pos] == val)
            return pos;
    return -1;
}
```

Liste: dritter Versuch

```
int getValueAt(list_t *l, int pos) {
    if (pos < 0 || pos >= l->next) {
        l->error = 2;
        return -1;
    }
    return l->values[pos];
}

.....

char getError(list_t *l) {     // neu !
    return l->error;
}

void destroy(list_t *l) {
    free(l->values);
    free(l);
}
```

main.c

```c
#include <stdio.h>
#include "liste.h"

int main(void) {
    int i;
    list_t *l = create();

    for (i = 1; i < 30; i++)
        append(l, i);
    toScreen(l);

    i = getValueAt(l, 30);
    if (getError(l) == 0)
        printf("value[%2d] = %2d\n", 30, i);

    destroy(l);
    return 0;
}
```

Kommt Ihnen diese Art der Programmierung fremd vor?
Finden Sie diese Art der Programmierung seltsam oder zu kompliziert?

Wir kennen diese Art der Programmierung bereits aus C von den Dateioperationen. In der Header-Datei `stdio.h` ist definiert:

```
typedef  struct _IO_FILE  FILE;
```

In unseren Programmen konnten wir Funktionen darauf nutzen:

```
#include <stdio.h>
void main(void) {
    FILE *f;
    f = fopen("dat.txt", "rw"); // vgl. create()
    fprintf(f, ...);            // vgl. append()
    fscanf(f, ...);             // vgl. getValueAt()
    fgets(..., f);
    fclose(f);                  // vgl. destroy()
}
```

Warum verdoppeln wir die Größe des Arrays?

- Laufzeit von `append` im worst-case:
  - neuen Speicherbereich allokieren: $\mathcal{O}(1)$
  - verschiebe n Elemente in größeren Speicherbereich: $\mathcal{O}(n)$
  - kopiere neues Element in die Liste: $\mathcal{O}(1)$
- Laufzeit für N Ausführungen von `append`:

$$\sum_{i=1}^{N} 1 + \sum_{i=1}^{\log_2(N)} 2^i \;\leq\; N + 2^{\log_2(N)+1} \;=\; 3 \cdot N \;\in\; \mathcal{O}(N)$$

- durchschnittliche Laufzeit von `append`:

$$T(N) \;=\; \frac{3 \cdot N}{N} \;=\; \mathcal{O}(1)$$

- Würde nur um ein Element vergrößert, würden N Ausführungen dauern:

$$\sum_{i=1}^{N} i \;=\; N^2$$