Übung zu
Algorithmen und Datenstrukturen (für ET/IT)

Sommersemester 2018

Mai Bui

Computer Aided Medical Procedures
Technische Universität München

Speicherorganisation

Normal gibt es einen gemeinsamen Speicher für Befehle und Daten (*Von-Neumann-Architektur*), der aber in Teilen organisiert ist:

- Im *Prozess-Adressraum* liegen die kompilierten Befehle.
- *Konstante Daten* werden separat hinterlegt.
- Für lokale Variablen und Steuerinformationen dient der *(Call) Stack* („Kellerprinzip")
- Weiterer Speicher steht auf dem *Heap* (Halde) zur freien Verfügung, der dort manuell verwaltet werden muss.

Stack

Jedes Programm erhält auf dem Stack einen Speicherbereich mit fixer Größe

- Wird verwendet für lokale Variablen und Funktionsparameter
- Neue lokale Variablen werden auf dem Stack abgelegt und bei Verlassen des Sichtbarkeitsbereichs wieder freigegeben
- Kein explizites Freigeben des Speichers nötig
- Effizientes Ablegen und Entfernen von Elementen

Heap

Nicht automatisch verwalteter Speicher, der über Zeiger addressiert wird

- Beliebige Größe
- Auf dem Heap liegende Objekte können global verfügbar sein
- Manuelle Speicherorganisation oder Garbage Collector notwendig

Grundsätzliche Organisation von Daten im Speicher

- Einzelne Werte und Objekte
- Felder (*Arrays*) von unmittelbar hintereinander liegenden Werten oder Objekten
  - Speicherposition der Nachbarn berechenbar
  - Beispiel: Bei einem `int`-Array liegt `array[i+5]` genau `5*sizeof(int)` Bytes hinter `array[i]`!
  - Die Größe ist fix und wird gewählt zur Laufzeit (Heap) oder zur Compile-Zeit (Call Stack).
- „Zufällig" verteilte Objekte, die auf ihre Nachbarn mittels Zeigern verweisen
  - Speicherposition der Nachbarn durch Verfolgen der Zeiger
  - Dynamische Größe und komplexere Topologie der Strukturen
  - Die Verwaltung ist aufwändiger!

Zeiger

- Verweise auf allozierten Speicher (also 32- oder 64-bit-Adressen) nennt man *Zeiger* (*Pointer*).

```
int* array = new int[n];

// Gebe die Adresse aus, sie verweist auf den Heap.
std::cout << array << std::endl;

// Jetzt greifen wir auf die erste Zelle zu.
// Sie liegt bei Adresse + 0 (also array + 0)
array[0] = 1;

// Die vierte liegt bei Adresse + 3 * Elementgroesse
// (also array + 3 * (4 Byte))
array[3] = 4;

delete[] array;
```

- Pointer sind lokale Variablen oder ebenfalls im Heap.
- Man kann natürlich auch die Adressen von lokalen Variablen und Funktionen herausbekommen; Arrays sind **immer** Zeiger.

Array

- Sozusagen Kette von n Variablen
- Indizierung mit $\mathbf{0} \leq \mathbf{i} < \mathbf{n}$

- Verschiedene Schreibweisen:
  - ```
    // Fixe Anzahl von Elementen
    int array[5];
    ```
  - ```
    // Variable Anzahl, aber Speichermanagement
    int* array = new int[n]; /* ... */ delete[] array;
    ```
  - ```
    // Variable Anzahl, aber std::vector
    #include <vector>
    std::vector<int> array(n);
    ```
    ```

C++ Alternativen - Was ist der `std::vector`?

- Repräsentiert ein Array mit dynamischer Größe (Teil der Standardbibliothek)
- Höherer Speicherverbrauch im Vergleich zum Array
- Methoden:
  - `insert(position, value)` fügt ein Element vor der spezifizierten position ein
  - `empty()` prüft ob der Vektor leer ist
  - `begin()` gibt einen Zeiger auf den Beginn des Vektors zurück
  - ...
- Durchlaufen der des Vektors mit `std::iterator`

C++ Alternativen - Was ist die `std::list`?

- Implementierung als doppelt verkettete Liste
- Einfügen/Löschen in konstanter Laufzeit
- Methoden:
  - `insert(position, value)` fügt ein Element in die Liste ein
  - `front()` gibt eine Referenz auf das erste Element in der Liste zurück
  - `push_back(value)` fügt ein Element an das Ende der Liste an
  - `remove(value)` löscht Elemente mit einem bestimmten Wert
  - `empty()` überprüft ob die Liste leer ist
  - ...
- Durchlaufen der Liste mit `std::iterator`

C++ Alternativen - Was ist der `std::iterator`?

- Abstrakter Datentyp zum Durchlaufen von Datenstrukturen (Teil der Standardbibliothek)
- Interface von `std::iterator it` u.a.
  - Weiterschaltung über `it++`
  - Wert über `*it`
  - ...
- Verfügbar etwa in `std::vector` oder `std::list` via `begin()`, aber natürlich jeweils unterschiedlich implementiert!
- Bei einer verketteten Liste existiert etwa intern ein Zeiger `p` auf einen Container:
  - `it++` bewirkt `p = p->next;`
  - `*it` ergibt `p->data`
  - ...

Code

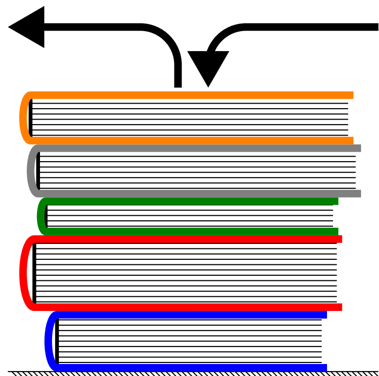
Abstrakte Datentypen

- Menge möglicher Operationen definiert das *Interface*
- Deren Auswirkungen sind wohldefiniert durch *Constraints*

- Die eigentliche Implementation ist *nicht* eindeutig definiert!

Stack

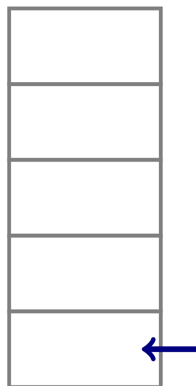
- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ...
- Constraint
  - LI-FO (Last In, First Out)
  - „Bücherstapel“
- Mögl. Implementation:
  - Array mit Schreibindex
  - Doppelt verkettete Liste
  - ...



Stack

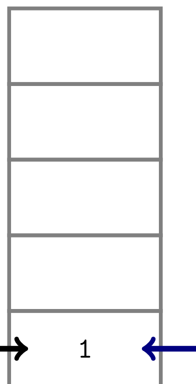
- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LI-FO (Last In, First Out)
  - Etwa über Schreibezeiger

`POP = ?`



Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LI-FO (Last In, First Out)
  - Etwa über Schreibezeiger

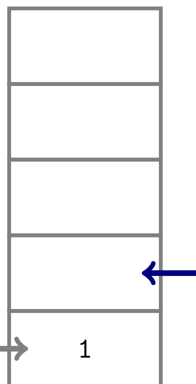


`PUSH(1)`

Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LI-FO (Last In, First Out)
  - Etwa über Schreibezeiger

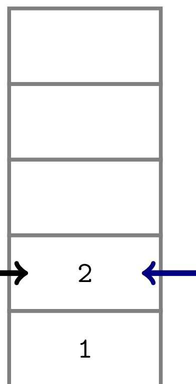
`PUSH(1)`



Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LI-FO (Last In, First Out)
  - Etwa über Schreibezeiger

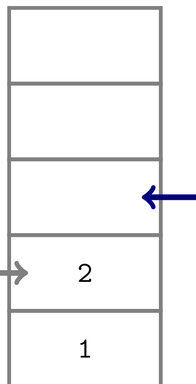
`PUSH(2)`



Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LI-FO (Last In, First Out)
  - Etwa über Schreibezeiger

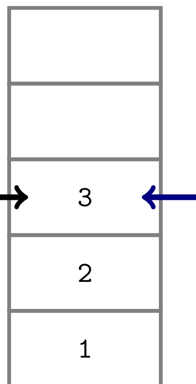
`PUSH(2)`



Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LI-FO (Last In, First Out)
  - Etwa über Schreibezeiger

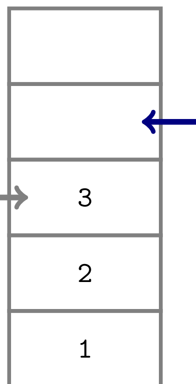
`PUSH(3)`



Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LI-FO (Last In, First Out)
  - Etwa über Schreibezeiger

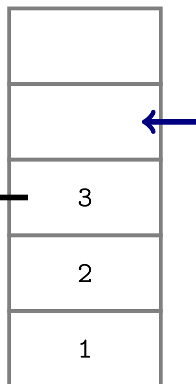
`PUSH(3)`



Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LI-FO (Last In, First Out)
  - Etwa über Schreibezeiger

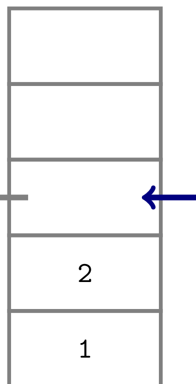
`POP = 3`



Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LI-FO (Last In, First Out)
  - Etwa über Schreibezeiger

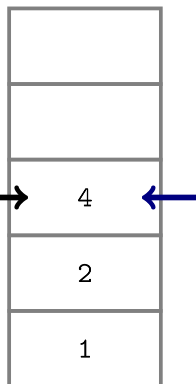
`POP = 3`



Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LI-FO (Last In, First Out)
  - Etwa über Schreibezeiger

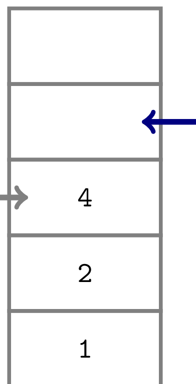
`PUSH(4)`



Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LI-FO (Last In, First Out)
  - Etwa über Schreibezeiger

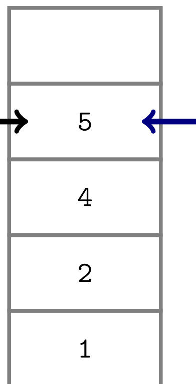
`PUSH(4)`



Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LI-FO (Last In, First Out)
  - Etwa über Schreibezeiger

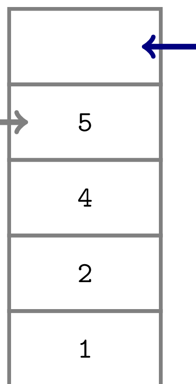
`PUSH(5)`



Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LI-FO (Last In, First Out)
  - Etwa über Schreibezeiger

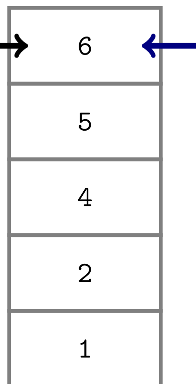
`PUSH(5)`



Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LI-FO (Last In, First Out)
  - Etwa über Schreibezeiger

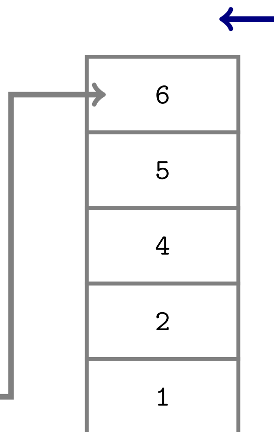
`PUSH(6)`



Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LI-FO (Last In, First Out)
  - Etwa über Schreibezeiger

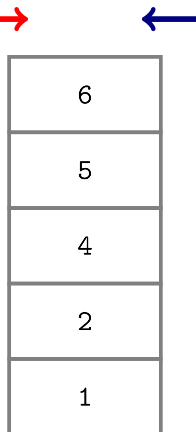
`PUSH(6)`



Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LI-FO (Last In, First Out)
  - Etwa über Schreibezeiger

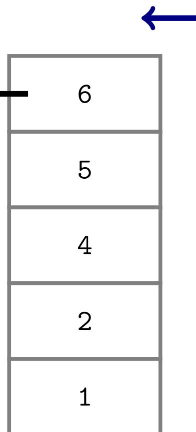
`PUSH(7)`



Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LI-FO (Last In, First Out)
  - Etwa über Schreibezeiger

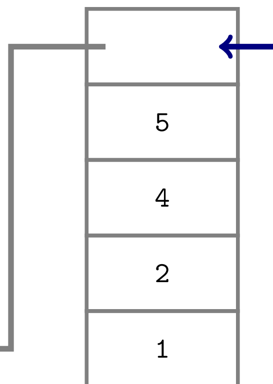
`POP = 6`



Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LI-FO (Last In, First Out)
  - Etwa über Schreibezeiger

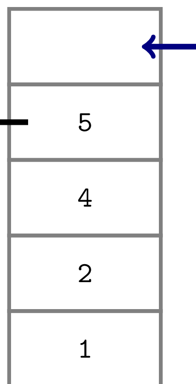
`POP = 6`



Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LI-FO (Last In, First Out)
  - Etwa über Schreibezeiger

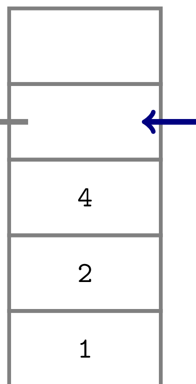
`POP = 5`



Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LI-FO (Last In, First Out)
  - Etwa über Schreibezeiger

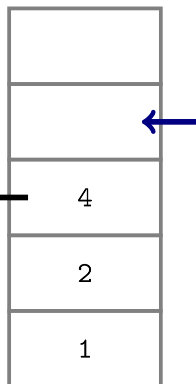
`POP = 5`



Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LI-FO (Last In, First Out)
  - Etwa über Schreibezeiger

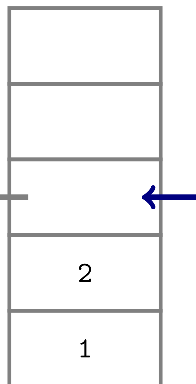
`POP = 4`



Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LI-FO (Last In, First Out)
  - Etwa über Schreibezeiger

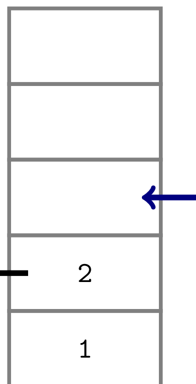
`POP = 4`



Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LI-FO (Last In, First Out)
  - Etwa über Schreibezeiger

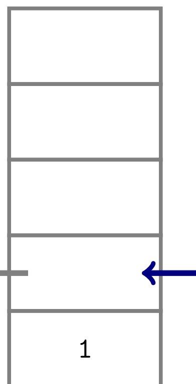
`POP = 2`



Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LI-FO (Last In, First Out)
  - Etwa über Schreibezeiger

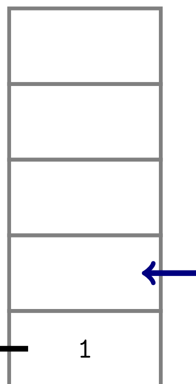
`POP = 2`



Stack

- Abstrakt definiert
- Interface-Funktionen
  - PUSH
  - POP
  - ggf. PEEK
- Constraint
  - LI-FO (Last In, First Out)
  - Etwa über Schreibezeiger

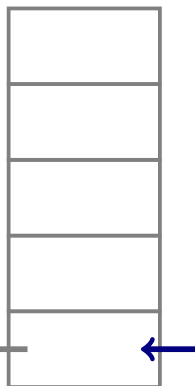
POP = 1



Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LI-FO (Last In, First Out)
  - Etwa über Schreibezeiger

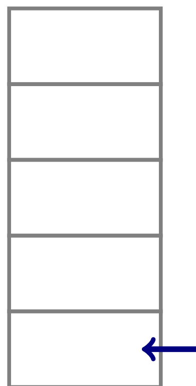
`POP = 1`



Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LI-FO (Last In, First Out)
  - Etwa über Schreibezeiger

`POP = ?`

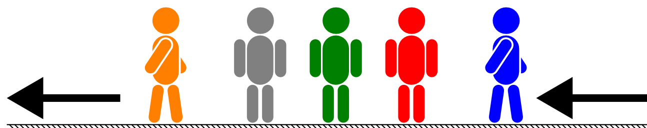


Code

Queue

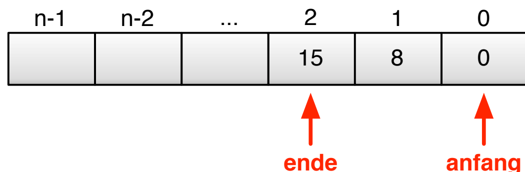
- Abstrakt definiert
- Interface-Funktionen
  - `ENQUEUE`
  - `DEQUEUE`
  - ...
- Constraint
  - FI-FO (First In, First Out)
  - „Warteschlange“

- Mögl. Implementation:
  - Verkettete Liste
  - Zwei Stacks
  - Ringpuffer
  - ...



Implementation Queue als sequentielle Liste

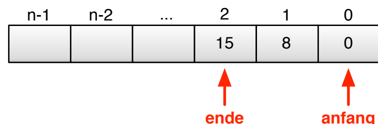
- Queue-Element speichern in sequentieller Liste L (Länge n)
- Anfang der Queue wird durch Index anfang markiert
- Ende der Queue wird durch Index ende markiert



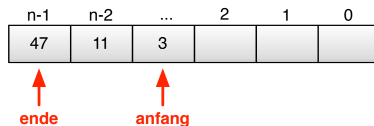
- enqueue(x) fügt Element bei Index ende+1 ein
- dequeue liefert Element bei Index anfang zurück und entfernt es durch Inkrement von anfang

Implementation Queue als sequentielle Liste 2

Problem:



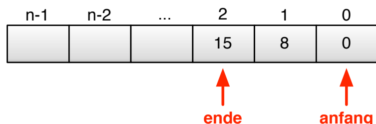
wird nach ein paar Operationen zu



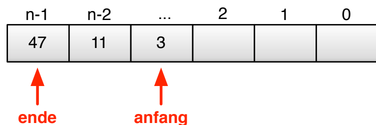
Linksdrift!

Implementation Queue als sequentielle Liste 2

Problem:



wird nach ein paar Operationen zu



Linksdrift!

Lösungsansatz: zirkuläre sequentielle Liste.

Code

Unit Tests

- Verifikation oft nicht praktikabel oder fehlerhaft
- Bau eines *separaten* Testprogramms mit systematischen Tests
  - Bekannte Eingabewerte, erwartete Ausgabe?
  - Zufällige Eingabewerte, erfüllt Ausgabe Nachbedingung?
- Design der Testfälle ist schwierig, keine (Sonder-)Fälle vergessen!
- Wenn die Vorbedingungen nicht erfüllt sind, ist das Verhalten eigentlich undefiniert. Trotzdem die Eingabewerte prüfen!
  - Auch zur Laufzeit!
  - Sicherheit!
  - Aussagekräftige Fehlermeldungen!

Prüfung mittels `assert`-Statement

- C/C++ stellt Statement zur Verfügung:

  `assert(`⟨Bedingung⟩`);`

- Bedingung geprüft *zur Laufzeit* des Programms
  - Keine Aktion falls die Bedingung *wahr* ist.
  - Programmende mit Fehler falls die Bedingung *falsch* ist.
- Etwa für Invarianten oder andere Zwischenbedingungen...
- Import via `#include <assert.h>`
- Sinnvoll eigentlich nur im *Debug*-Modus

Code

Wiederholung: Divide and Conquer - MergeSort

Sei $A = \{a_1, \ldots, a_n\}$ Feld mit n natürlichen Zahlen $a_i \in \mathbb{N}$.

Aufgabe: sortiere A in aufsteigender Reihenfolge.

- Lösung mit Divide and Conquer-Muster: Merge Sort
- Idee:
  - Divide: teile A auf in zwei gleich große Teilfelder
  - Rekursion: rufe Merge Sort rekursiv für die zwei Teilfelder auf
  - Conquer: setze die Teilfelder zusammen (merge bzw. mischen)
- Wann ist Teilfolge "klein", d.h. wann löst man explizit?
  $\rightarrow$ Teilfolge mit nur einem Element $\rightarrow$ sortiert!

Wiederholung: MergeSort - Algorithmus

Input: zu sortierendes Feld A
Output: sortiertes Feld

MergeSort(A):
  if (A ein-elementig) { // Teilfeld ist klein
    return A;
  }
  else {
    halbiere A in A_1 und A_2; // Divide
    $A_1 =$ **MergeSort**(A_1); // Rekursion
    $A_2 =$ **MergeSort**(A_2); // Rekursion
    return Merge(A_1, A_2); // Conquer
  }

$\rightarrow$ "komplizierter" Teil ist in **Merge**!

Wiederholung: MergeSort - Merge Funktion

Input: zu sortierende Felder A_1, A_2
Output: sortiertes Feld B

Merge(A_1, A_2):
  B = leeres Feld;
  while (A_1 und A_2 nicht leer) {
    entferne das kleinere der Anfangselemente von A_1 bzw. A_2;
    hänge dieses Element an B an;
  }
  hänge das verbliebene, nicht-leere Feld A_1 oder A_2 an B an;
  return B;

→ tatsächliche Implementation muß Entfernen/Anhängen effizient lösen!

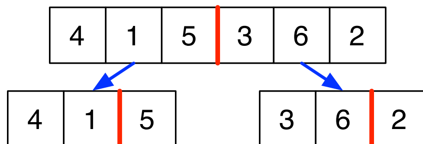
Wiederholung: MergeSort - Beispiel

Divide

4	1	5	3	6	2

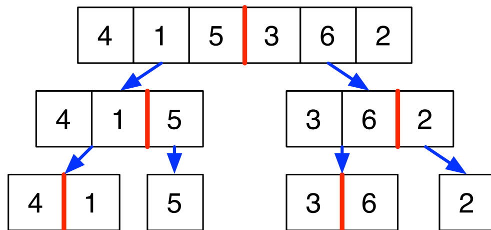
Wiederholung: MergeSort - Beispiel

Divide



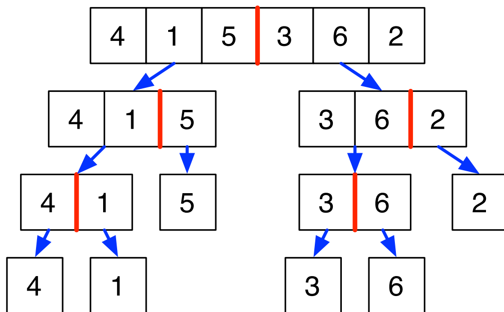
Wiederholung: MergeSort - Beispiel

Divide

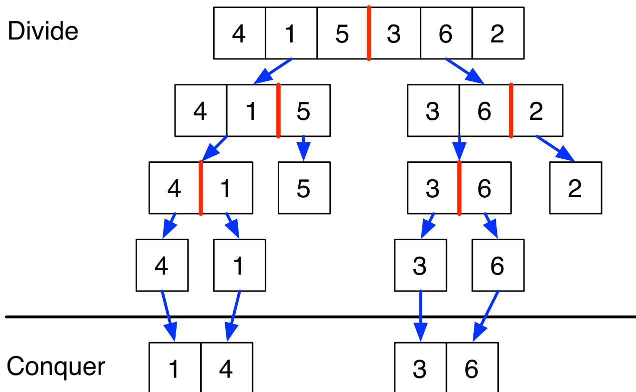


Wiederholung: MergeSort - Beispiel

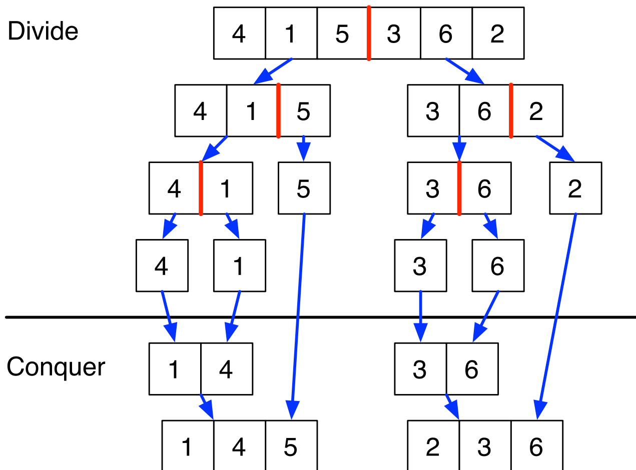
Divide



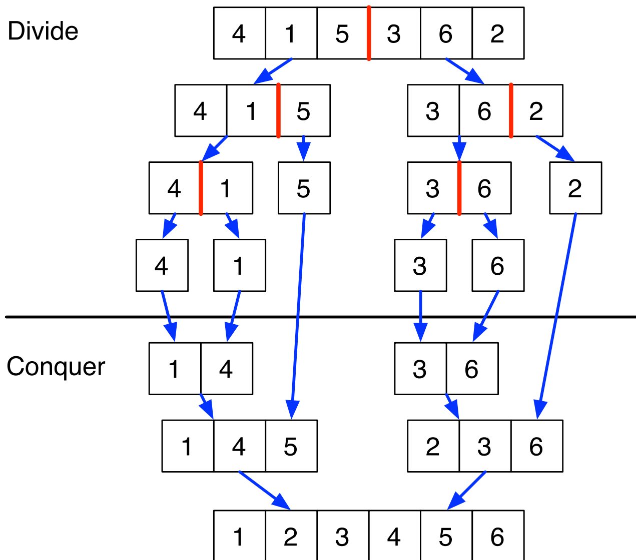
Wiederholung: MergeSort - Beispiel



Wiederholung: MergeSort - Beispiel



Wiederholung: MergeSort - Beispiel



Code