

Übungsstunde — Informatik — 10

Adel Gavranović

Pointer-relevante Operatoren, Referenzen vs. Pointer, Iteratoren, `this->`, dynamischer Speicher

Übersicht

Follow-up

& vs *

Referenzen vs Pointer

this->

Dynamische Datenstrukturen & Iteratoren

Our_list Grundmaterial

Our_list Bonusmaterial



n.ethz.ch/~agavranovic

 Material

 Webpage

 Mail

1. Follow-up

Anzahl Argumente für überladene Operatoren

Anzahl Argumente für überladene Operatoren

Letzte Übungsstunde was ich überrascht, dass overloaded operators nur einen Input benötigen. Jetzt weiss ich wieso!

Anzahl Argumente für überladene Operatoren

Letzte Übungsstunde was ich überrascht, dass overloaded operators nur einen Input benötigen. Jetzt weiss ich wieso!

```
Tribool Tribool::operator&&(const Tribool& other) const {  
    Tribool result(std::min(value, other.value));  
    return result;  
}
```

A ~~aa~~ B

Anzahl Argumente für überladene Operatoren

Letzte Übungsstunde was ich überrascht, dass overloaded operators nur einen Input benötigen. Jetzt weiss ich wieso!

```
Tribool Tribool::operator&&(const Tribool& other) const {  
    Tribool result(std::min(value, other.value));  
    return result;  
}
```

Folgendes ist equivalent:

```
Tribool A("True"), B("Unknown");  
Tribool C = A && B;           // infix notation  
// Equivalent to  
Tribool D = A.operator&&(B);  // explicit member function notation
```

Non-member Funktionen brauchen noch immer beide Objekte als Input!

Ranged for-loops

Ranged for-loops

Letzte Übungsstunde habe ich einen Fehler beim Erklären der "Ranged for-loops" gemacht — Bitte entschuldigt die Verwirrung.

Ranged for-loops

Letzte Übungsstunde habe ich einen Fehler beim Erklären der "Ranged for-loops" gemacht — Bitte entschuldigt die Verwirrung.

```
std::vector<int> numbers = {1, 2, 3, 4, 5};

for (int i : numbers) {
    std::cout << (i += 1) << " ";
}

std::cout << "| ";

for (int i : numbers) {
    std::cout << i << " ";
}
```

Ranged for-loops

Letzte Übungsstunde habe ich einen Fehler beim Erklären der "Ranged for-loops" gemacht — Bitte entschuldigt die Verwirrung.

```
std::vector<int> numbers = {1, 2, 3, 4, 5};

for (int i : numbers) {
    std::cout << (i += 1) << " ";
}

std::cout << "| ";

for (int i : numbers) {
    std::cout << i << " ";
}
```

Was wird hier der Output sein?

Ranged for-loops

Letzte Übungsstunde habe ich einen Fehler beim Erklären der "Ranged for-loops" gemacht — Bitte entschuldigt die Verwirrung.

```
std::vector<int> numbers = {1, 2, 3, 4, 5};

for (int i : numbers) {
    std::cout << (i += 1) << " ";
}

std::cout << "| ";

for (int i : numbers) {
    std::cout << i << " ";
}
```

Was wird hier der Output sein? 2 3 4 5 6 | 1 2 3 4 5

Ranged for-loops mit Referenzen

Ranged for-loops mit Referenzen

Die Elemente im Container werden also "direkt" verfügbar gemacht, also keine Iteratoren nötig!

Ranged for-loops mit Referenzen

Die Elemente im Container werden also "direkt" verfügbar gemacht, also keine Iteratoren nötig!

```
std::vector<int> numbers = {1, 2, 3, 4, 5};

for (int& i : numbers) {                               // referenced (&) this time!
    std::cout << (i += 1) << " ";
}

std::cout << "| ";

for (int i : numbers) {
    std::cout << i << " ";
}
```

Ranged for-loops mit Referenzen

Die Elemente im Container werden also "direkt" verfügbar gemacht, also keine Iteratoren nötig!

```
std::vector<int> numbers = {1, 2, 3, 4, 5};

for (int& i : numbers) {                                // referenced (&) this time!
    std::cout << (i += 1) << " ";
}

std::cout << "| ";

for (int i : numbers) {
    std::cout << i << " ";
}
```

Was wird hier der Output sein?

Ranged for-loops mit Referenzen

Die Elemente im Container werden also "direkt" verfügbar gemacht, also keine Iteratoren nötig!

```
std::vector<int> numbers = {1, 2, 3, 4, 5};

for (int& i : numbers) {                               // referenced (&) this time!
    std::cout << (i += 1) << " ";
}

std::cout << "| ";

for (int i : numbers) {
    std::cout << i << " ";
}
```

Was wird hier der Output sein? 2 3 4 5 6 | 2 3 4 5 6

Fragen/Unklarheiten?

2. Feedback zu **code** expert

Allgemeines bezüglich **code expert**

Aufgabe "Recursive Function Analysis"

Aufgabe "Recursive Function Analysis"

- Achtet auf die Details!

Musterlösung für Teil 1

Musterlösung

```
bool f(const int n) {  
    if (n == 0) return false;  
    return !f(n - 1);  
}
```

Musterlösung für Teil 1

Musterlösung

```
bool f(const int n) {  
    if (n == 0) return false;  
    return !f(n - 1);  
}
```

i) |

```
// PRE:  n >= 0  
// POST: returns `false` if n is even  
//       returns `true`  if n is odd
```

Musterlösung für Teil 1

Musterlösung

```
bool f(const int n) {  
    if (n == 0) return false;  
    return !f(n - 1);  
}
```

i) |

```
// PRE:  n >= 0  
// POST: returns `false` if n is even  
//       returns `true`  if n is odd
```

ii) The function `f` immediately terminates for `n == 0`. With each recursive call `n` is decremented, i.e., `f` is called with parameter `<n`, eventually reaching `n == 0` (at which point it terminates — as mentioned above).

Musterlösung für Teil 1

Musterlösung

```
bool f(const int n) {  
    if (n == 0) return false;  
    return !f(n - 1);  
}
```

i) |

```
// PRE:  n >= 0  
// POST: returns `false` if n is even  
//       returns `true`  if n is odd
```

- ii) The function f immediately terminates for $n == 0$. With each recursive call n is decremented, i.e., f is called with parameter $<n$, eventually reaching $n == 0$ (at which point it terminates — as mentioned above).
- iii) $\text{Calls}_f(n) = n + 1$ (including first non-rec. call)

Musterlösung für Teil 2

Musterlösung

```
void g(const int n) {  
    if (n == 0) {  
        std::cout << "*"; return;  
    }  
    g(n - 1); g(n - 1);  
}
```

Musterlösung für Teil 2

Musterlösung

```
void g(const int n) {  
    if (n == 0) {  
        std::cout << "*"; return;  
    }  
    g(n - 1); g(n - 1);  
}
```

i) | `// PRE: n >= 0`
| `// POST: prints 2n stars to std::out`

Musterlösung für Teil 2

Musterlösung

```
void g(const int n) {  
    if (n == 0) {  
        std::cout << "*"; return;  
    }  
    g(n - 1); g(n - 1);  
}
```

i) | `// PRE: n >= 0`
| `// POST: prints 2n stars to std::out`

- ii) The function `g` immediately terminates for `n == 0`. With each recursive call `n` is decremented, i.e., `g` is called with parameter `< n`, eventually reaching `n == 0`. This is true for both recursive calls of `g`.

Musterlösung für Teil 2

Musterlösung

```
void g(const int n) {  
    if (n == 0) {  
        std::cout << "*"; return;  
    }  
    g(n - 1); g(n - 1);  
}
```

i) | `// PRE: n >= 0`
| `// POST: prints 2^n stars to std::out`

- ii) The function `g` immediately terminates for `n == 0`. With each recursive call `n` is decremented, i.e., `g` is called with parameter `< n`, eventually reaching `n == 0`. This is true for both recursive calls of `g`.
- iii) $\text{Calls}_g(n) = \sum_{i=0}^n 2^i = 2^{n+1} - 1$
(including first non-rec. call)

Fragen bezüglich **code expert** eurerseits?

3. Lernziele

Ziele

- Unterschiede zwischen Pointern und Referenzen verstehen
- Programme mit Pointern tracen und schreiben können
- Programme mit dynamischem Speicher schreiben können
- Simple Container implementieren können

4. Zusammenfassung

5. & VS *

Bedeutungen von &

Das Symbol & hat in C++ viele Bedeutungen. Das ist verwirrend. Es hat 3 *verschiedene Bedeutungen*, je nach Position im Code:

Bedeutungen von &

Das Symbol & hat in C++ viele Bedeutungen. Das ist verwirrend. Es hat 3 *verschiedene Bedeutungen*, je nach Position im Code:

Bedeutung von &

Bedeutungen von &

Das Symbol & hat in C++ viele Bedeutungen. Das ist verwirrend. Es hat 3 *verschiedene Bedeutungen*, je nach Position im Code:

Bedeutung von &

1. als AND-operator

```
bool z = x && y;
```

Bedeutungen von &

Das Symbol & hat in C++ viele Bedeutungen. Das ist verwirrend. Es hat 3 *verschiedene Bedeutungen*, je nach Position im Code:

Bedeutung von &

1. als AND-operator

```
bool z = x && y;
```

2. um eine Variable als Alias zu deklarieren

```
int& y = x;
```

Bedeutungen von &

Das Symbol & hat in C++ viele Bedeutungen. Das ist verwirrend. Es hat 3 *verschiedene Bedeutungen*, je nach Position im Code:

Bedeutung von &

1. als AND-operator

```
bool z = x && y;
```

2. um eine Variable als Alias zu deklarieren

```
int& y = x;
```

3. um die Adresse einer Variable zu erhalten (address-operator)

```
int *ptr_a = &a;
```

Bedeutungen von *

Dito mit dem Symbol *.

Bedeutung von *

Bedeutungen von *

Dito mit dem Symbol *.

Bedeutung von *

1. als (arithmetischer) Multiplikation-operator

$$z = x * y;$$

Bedeutungen von *

Dito mit dem Symbol *.

Bedeutung von *

1. als (arithmetischer) Multiplikation-operator

`z = x * y;`

2. um eine Pointer-Variable zu deklarieren

`int* ptr_a = &a;`



Bedeutungen von *

Dito mit dem Symbol *.

Bedeutung von *

1. als (arithmetischer) Multiplikation-operator

```
z = x * y;
```

2. um eine Pointer-Variable zu deklarieren

```
int* ptr_a = &a;
```

3. um auf eine Variable via ihrem Pointer zuzugreifen
(dereference-operator)

```
int a = *ptr_a;
```

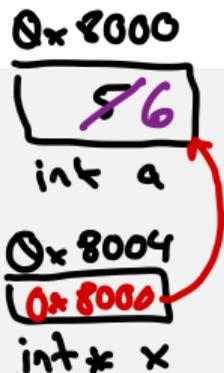
Fragen/Unklarheiten?

6. Referenzen vs Pointer

Pointer Basics

Versucht, folgendes Programm¹ detailliert zu tracen

```
int main() {  
  
    int a = 5;  
    int* x = &a;  
    (*x) = 6;  
    a  
    return 0;  
}
```



¹Vollständiger Trace [hier](#)

References

```
void references(){
    int a = 1;
    int b = 2;
    int& x = a;
    int& y = x;
    y = b;

    std::cout
    << a << " "
    << b << " "
    << x << " "
    << y << std::endl;
}
```

Trace das Programm und schreibe den erwarteten Output hin, wenn die Funktion aufgerufen wurde

References

```
void references(){
    int a = 1;
    int b = 2;
    int& x = a;
    int& y = x;
    y = b;

    std::cout
    << a << " "
    << b << " "
    << x << " "
    << y << std::endl;
}
```

Trace das Programm und schreibe den erwarteten Output hin, wenn die Funktion aufgerufen wurde

2 2 2 2

Pointers

```
void pointers(){  
    int a = 1;  
    int b = 2;  
    int* x = &a;  
    int* y = x;  
  
    std::cout  
    << a << " "  
    << b << " "  
    << x << " "  
    << y << std::endl;  
}
```

Trace das Programm und schreibe den erwarteten Output hin, wenn die Funktion aufgerufen wurde

Pointers

```
void pointers(){  
    int a = 1;  
    int b = 2;  
    int* x = &a;  
    int* y = x;  
  
    std::cout  
    << a << " "  
    << b << " "  
    << x << " "  
    << y << std::endl;  
}
```

Trace das Programm und schreibe den erwarteten Output hin, wenn die Funktion aufgerufen wurde

1 2 0x7ffe4d1fb904 0x7ffe4d1fb904

Pointers

```
void pointers(){  
    int a = 1;  
    int b = 2;  
    int* x = &a;  
    int* y = x;  
  
    std::cout  
    << a << " "  
    << b << " "  
    << x << " "  
    << y << std::endl;  
}
```

Trace das Programm und schreibe den erwarteten Output hin, wenn die Funktion aufgerufen wurde

1 2 0x7ffe4d1fb904 0x7ffe4d1fb904

(Die Adressen könnten bei jedem Aufruf anders sein!)

Pointers and Addresses

```
void ptrs_and_addresses(){  
    int a = 5;  
    int b = 7;  
  
    int* x = nullptr;  
    x = &a;  
  
    std::cout << a << "\n";  
    std::cout << *x << "\n";  
  
    std::cout << x << "\n";  
    std::cout << &a << "\n";  
}
```

Trace das Programm und schreibe den erwarteten Output hin, wenn die Funktion aufgerufen wurde

Pointers and Addresses

```
void ptrs_and_addresses(){
    int a = 5;
    int b = 7;

    int* x = nullptr;
    x = &a;

    std::cout << a << "\n";
    std::cout << *x << "\n";

    std::cout << x << "\n";
    std::cout << &a << "\n";
}
```

Trace das Programm und schreibe den erwarteten Output hin, wenn die Funktion aufgerufen wurde

5

5

0x7ffe4d1fb914

0x7ffe4d1fb914

(Die Adressen könnten bei jedem Aufruf anders sein!)

Fragen/Unklarheiten?

7. this->

Was zum f*&k ist this->?

Bedeutung von this->

this-> hat zwei Teile

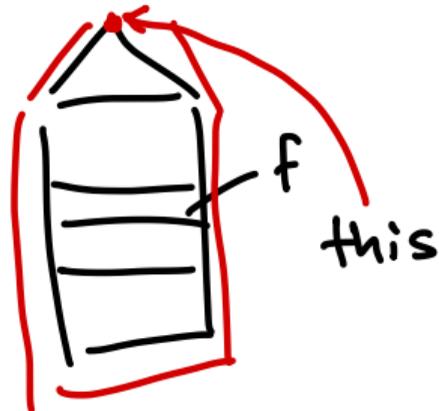
Was zum f*&k ist this->?

Bedeutung von this->

this-> hat zwei Teile

- this

- ist ein Pointer zum *aktuellen* Objekt (Class oder Struct T)



Was zum f*&k ist this->?

Bedeutung von this->

this-> hat zwei Teile

- **this**

- ist ein Pointer zum *aktuellen* Objekt (Class oder Struct T)
- also vom Typ T*

Was zum f*&k ist this->?

Bedeutung von this->

this-> hat zwei Teile

- **this**

- ist ein Pointer zum *aktuellen* Objekt (Class oder Struct T)
- also vom Typ T*

- **->**

- ist ein sehr cool aussehender Operator

Was zum f*&k ist this->?

Bedeutung von this->

this-> hat zwei Teile

- this

- ist ein Pointer zum *aktuellen* Objekt (Class oder Struct T)
- also vom Typ T*

- ->

- ist ein sehr cool aussehender Operator
- `this->member_element` ist äquivalent zu `*(this).member_element`
- Der Pfeil-Operator dereferenziert einen Pointer zu einem Objekt, um auf einen seiner Members zuzugreifen (Funktionen oder Variablen)

this → next → next ;
≙
** (* (*this).next).next*

Beispiel

Wofür wird `this` hier benutzt?

```
struct WeirdNumber {  
  
    int number;  
  
    void increment_by(int number){  
        (*this).number = (*this).number + number;  
        // or  
        // this->number = this->number + number;  
    }  
};
```

Beispiel

Wofür wird `this` hier benutzt?

```
struct WeirdNumber {  
  
    int number;  
  
    void increment_by(int number){  
        (*this).number = (*this).number + number;  
        // or  
        // this->number = this->number + number;  
    }  
};
```

Um die beiden gleichnamigen Variablen `number` zu unterscheiden

Beispiel

```
int main(){

    WeirdNumber a = {42};
    WeirdNumber b = {-17};

    a.increment_by(3);
    // 'this' in the call of the increment_by function
    // refers to the object a.
    b.increment_by(2);
    // 'this' in the call of the increment_by function
    // refers to the object b.

    std::cout << a.number << " " << b.number << std::endl;

    return 0;
}
```

8. Dynamische Datenstrukturen & Iteratoren

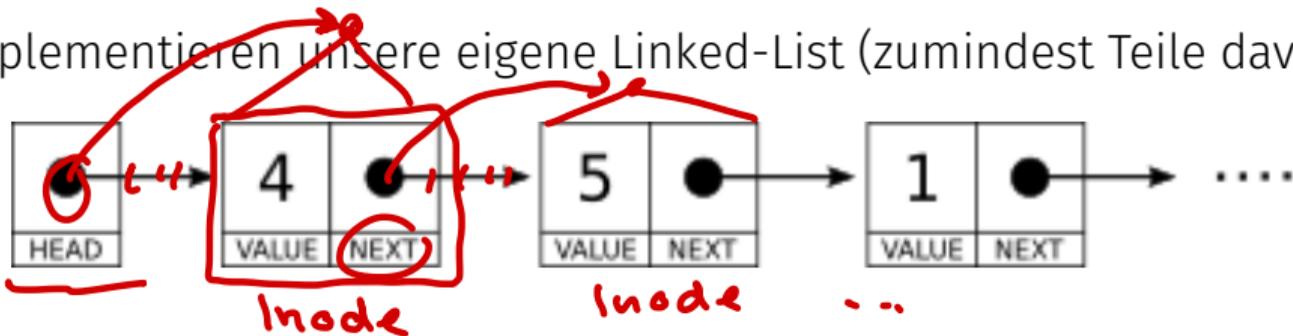
8. Dynamische Datenstrukturen & Iteratoren

8.1. `Our_list` Grundmaterial

Wir implementieren unsere eigene Linked-List (zumindest Teile davon)

our_list

Wir implementieren unsere eigene Linked-List (zumindest Teile davon)

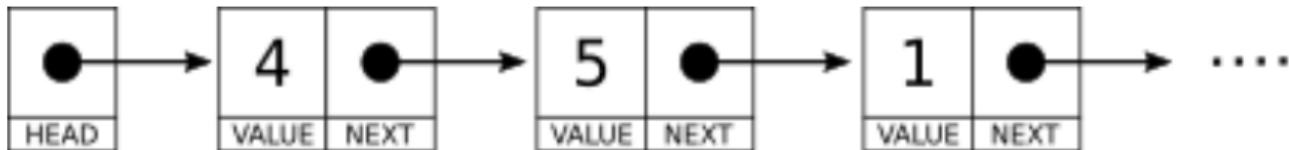


- Eine Liste besteht aus "Blöcken" von `lnodes`, wobei eine `lnode` immer auf die nächste zeigt



our_list

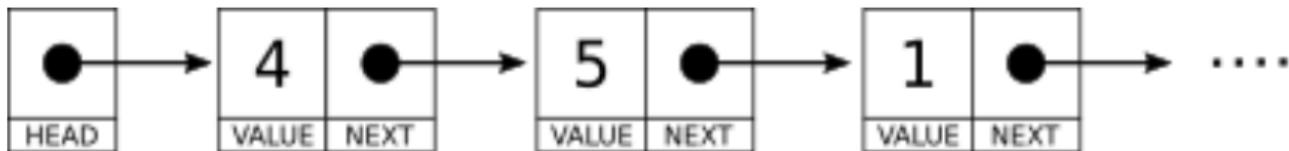
Wir implementieren unsere eigene Linked-List (zumindest Teile davon)



- Eine Liste besteht aus "Blöcken" von `lnodes`, wobei eine `lnode` immer auf die nächste zeigt
- Aber was ist überhaupt eine `lnode`?

our_list

Wir implementieren unsere eigene Linked-List (zumindest Teile davon)



- Eine Liste besteht aus "Blöcken" von `lnodes`, wobei eine `lnode` immer auf die nächste zeigt
- Aber was ist überhaupt eine `lnode`?
- Antwort: ein Struct, das aus einem `int` `value` und einem `lnode` pointer besteht

our_list

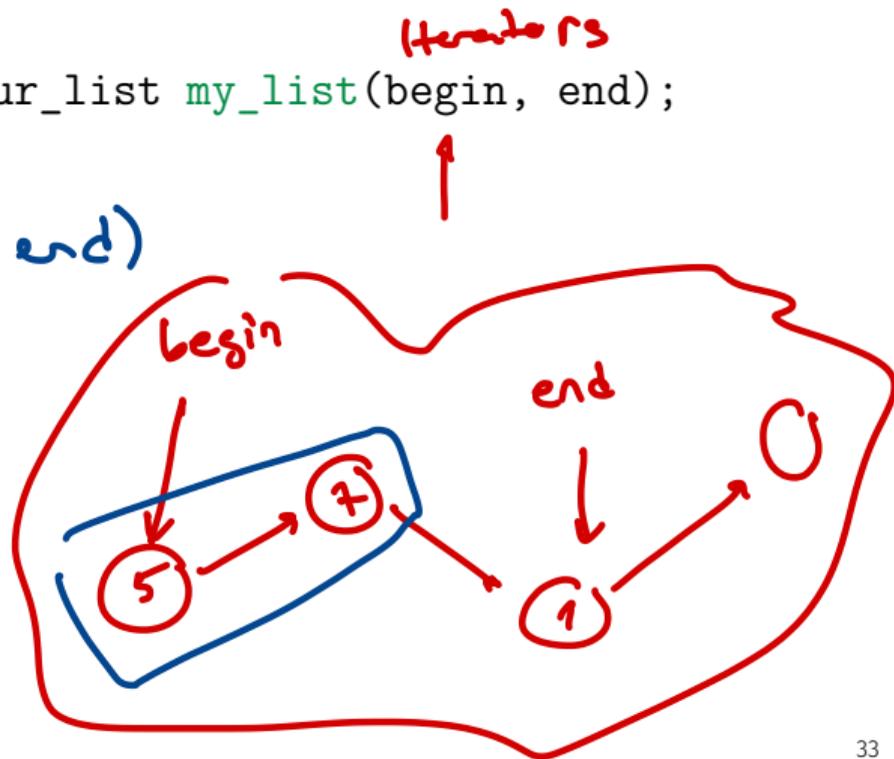
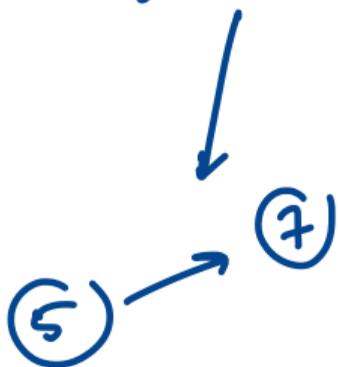
Erste Aufgabe: Implementiere einen Constructor, der eine neue Liste mit Iteratoren initialisiert

our_list

Erste Aufgabe: Implementiere einen Constructor, der eine neue Liste mit Iteratoren initialisiert

- Wir wollen schreiben können: `our_list my_list(begin, end);`

`our_list my_list(begin, end)`



Erste Aufgabe: Implementiere einen Constructor, der eine neue Liste mit Iteratoren initialisiert

- Wir wollen schreiben können: `our_list my_list(begin, end);`
- Idee: Benutze die Iteratoren, um neue `lnodes` in die Liste hinzuzufügen

Erste Aufgabe: Implementiere einen Constructor, der eine neue Liste mit Iteratoren initialisiert

- Wir wollen schreiben können: `our_list my_list(begin, end);`
- Idee: Benutze die Iteratoren, um neue `lnodes` in die Liste hinzuzufügen
- Wie können wir auf die verschiedenen Elemente zugreifen?

Erste Aufgabe: Implementiere einen Constructor, der eine neue Liste mit Iteratoren initialisiert

- Wir wollen schreiben können: `our_list my_list(begin, end);`
- Idee: Benutze die Iteratoren, um neue `lnodes` in die Liste hinzuzufügen
- Wie können wir auf die verschiedenen Elemente zugreifen?
 - Zugriff auf Wert der `lnode`, auf die der Iterator zeigt:

`*it`

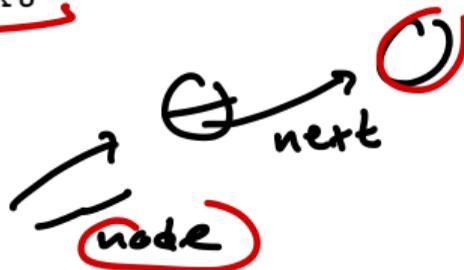
Erste Aufgabe: Implementiere einen Constructor, der eine neue Liste mit Iteratoren initialisiert

- Wir wollen schreiben können: `our_list my_list(begin, end);`
- Idee: Benutze die Iteratoren, um neue `lnodes` in die Liste hinzuzufügen
- Wie können wir auf die verschiedenen Elemente zugreifen?
 - Zugriff auf Wert der `lnode`, auf die der Iterator zeigt:

`*it`

- Nächste `lnode` in der Folge:

`node` → `next`



Erste Aufgabe: Implementiere einen Constructor, der eine neue Liste mit Iteratoren initialisiert

- Wir wollen schreiben können: `our_list my_list(begin, end);`
- Idee: Benutze die Iteratoren, um neue `lnodes` in die Liste hinzuzufügen
- Wie können wir auf die verschiedenen Elemente zugreifen?

- Zugriff auf Wert der `lnode`, auf die der Iterator zeigt:

`*it`

- Nächste `lnode` in der Folge:

`node->next`

- Pointer zu neuer `lnode` erstellen::

`new lnode{value, pointer}`

Denkt daran: `new T` gibt einen `T*` zurück

our_list: class our_list

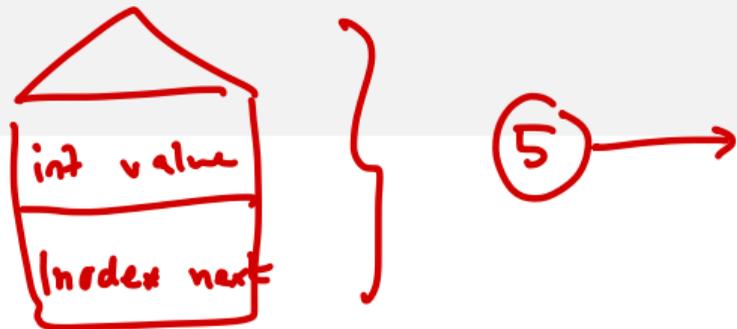
```
class our_list {  
    struct lnode {  
        // ...  
    };  
  
    lnode* head;  
  
public:  
  
    class const_iterator {  
        // ...  
    };  
  
    // member functions  
};
```

HEAD
lnode*

our_list: struct lnode

```
//                               in class our_list                               //
```

```
struct lnode {  
    int value;  
    lnode* next;  
};
```



our_list: const_iterator

```
//                               in class our_list                               //  
class const_iterator {  
    const lnode* node;  
public:  
    const_iterator(const lnode* const n);  
    // PRE: Iterator doesn't point to the element beyond the last one  
    // POST: Iterator points to the next element  
    const_iterator& operator++(); // Pre-increment  
    // POST: Return the reference to the number at which the  
    //       iterator is currently pointing  
    const int& operator*() const;  
    // True if iterators are pointing to different elements  
    bool operator!=(const const_iterator& other) const;  
    // True if iterators are pointing to the same element  
    bool operator==(const const_iterator& other) const;  
};
```

our_list: Memberfunktionen

```
//          in class our_list          //
```

```
our_list();
```

```
// PRE: begin and end are iterators pointing to the same vector  
//      and begin is before end  
// POST: Constructed our_list contains all elements between begin and end  
our_list(const_iterator begin, const_iterator end);
```

```
// POST: e is appended at the beginning of the vector  
void push_front(int e);
```

```
// POST: Returns an iterator that points to the first element  
const_iterator begin() const;
```

```
// POST: Returns an iterator that points after the last element  
const_iterator end() const;
```



L.push_front(5);



Aufgabe "our_list::init"

Aufgabe "our_list::init"

- Öffnet "our_list::init" auf **code expert**

Aufgabe "our_list::init"

- Öffnet "our_list::init" auf **code expert**
- Überlegt euch, wie ihr das Problem mit Stift und Papier angehen würdet

Aufgabe "our_list::init"

- Öffnet "our_list::init" auf **code expert**
- Überlegt euch, wie ihr das Problem mit Stift und Papier angehen würdet
- Programmiert eine Lösung (optional in Gruppen)

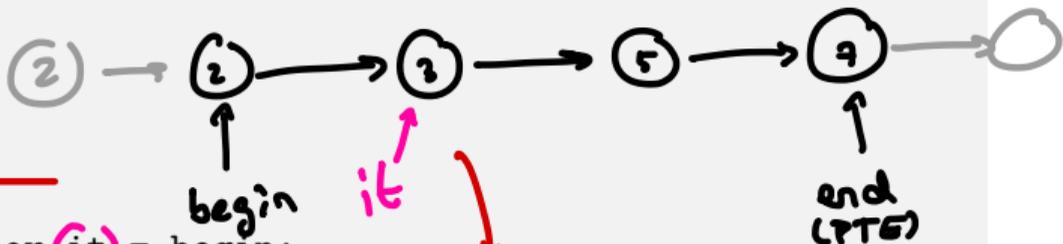
Aufgabe "our_list::init" (Lösung)

Aufgabe "our_list::init" (Lösung)

in ...cpp

```
xxx our_list::our_list(our_list::const_iterator begin, our_list::const_iterator end) {
```

```
    this->head = nullptr;
    if (begin == end) {
        return;
    }
```



```
    // add first element
```

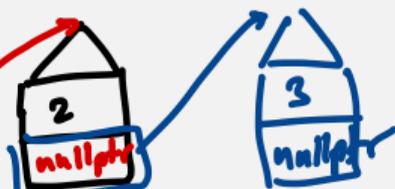
```
    our_list::const_iterator it = begin;
    this->head = new lnode { *it, nullptr };
    ++it;
```

```
    lnode *node = this->head;
```

```
    // add remaining elements
```

```
    for (; it != end; ++it) {
        node->next = new lnode { *it, nullptr };
        node = node->next;
    }
```

```
}
```



Fragen/Unklarheiten?

Zweite Aufgabe: Implementiere eine Funktion der Class "our_list", die eine Node mit der nächsten tauscht

Zweite Aufgabe: Implementiere eine Funktion der Class "our_list", die eine Node mit der nächsten tauscht

- Ihr könnt eine recht ähnliche Herangehensweise wie bei anderen Swap-Funktionen benutzen (also mit einer temporären Variable `tmp`)

Zweite Aufgabe: Implementiere eine Funktion der Class "our_list", die eine Node mit der nächsten tauscht

- Ihr könnt eine recht ähnliche Herangehensweise wie bei anderen Swap-Funktionen benutzen (also mit einer temporären Variable `tmp`)
- Aber:
 - Benutzt Pointer
 - Was passiert im Fall "0" (wenn der Head Pointer getauscht werden soll)?
 - Wie könnt ihr vermeiden, dass nicht plötzlich auf unerlaubten Speicher zugegriffen wird?

Aufgabe "our_list::swap"

Aufgabe "our_list::swap"

- Öffnet "our_list::swap" auf **code expert**

Aufgabe "our_list::swap"

- Öffnet "our_list::swap" auf **code expert**
- Überlegt euch, wie ihr das Problem mit Stift und Papier angehen würdet

Aufgabe "our_list::swap"

- Öffnet "our_list::swap" auf **code expert**
- Überlegt euch, wie ihr das Problem mit Stift und Papier angehen würdet
- Programmiert eine Lösung (optional in Gruppen)

Aufgabe "our_list::swap" (Lösung)

Aufgabe "our_list::swap" (Lösung)

```
void our_list::swap(int index) {  
  
    if (index == 0) {  
  
        assert(this->head != nullptr);  
        assert(this->head->next != nullptr);  
  
        lnode* tmp = this->head->next;  
        this->head->next = this->head->next->next;  
        tmp->next = this->head;  
        this->head = tmp;  
  
    } else { /* ... */ }
```

Aufgabe "our_list::swap" (Lösung)

```
else { lnode* prev = nullptr;
      lnode* curr = this->head;

      while (index > 0) { // Find the element
        prev = curr;
        curr = curr->next;
        --index;
      }

      assert(curr != nullptr);
      assert(curr->next != nullptr);

      lnode* tmp = curr->next; // Swap with the next one
      curr->next = curr->next->next;
      tmp->next = curr;
      prev->next = tmp;
    } // two '}' to close function
```

Fragen/Unklarheiten?

8. Dynamische Datenstrukturen & Iteratoren

8.2. Our_list Bonusmaterial

Aufgabe "our_list::extend"

Aufgabe "our_list::extend"

- Öffnet "our_list::extend" auf **code expert**

Aufgabe "our_list::extend"

- Öffnet "our_list::extend" auf **code expert**
- Überlegt euch, wie ihr das Problem mit Stift und Papier angehen würdet

Aufgabe "our_list::extend"

- Öffnet "our_list::extend" auf **code expert**
- Überlegt euch, wie ihr das Problem mit Stift und Papier angehen würdet
- Programmiert eine Lösung (optional in Gruppen)

Aufgabe "our_list::extend" (Lösung)

Aufgabe "our_list::extend" (Lösung)

```
void our_list::extend(our_list::const_iterator begin,
                    our_list::const_iterator end) {
    if (begin == end) { return; }
    our_list::const_iterator it = begin;
    if (this->head == nullptr) {
        this->head = new lnode { *it, nullptr };
        ++it;
    }
    lnode *n = this->head;
    while (n->next != nullptr) {
        n = n->next;
    }
    for (; it != end; ++it) {
        n->next = new lnode { *it, nullptr };
        n = n->next;
    }
}
```

Fragen/Unklarheiten?

Aufgabe "our_list::merge_sorted" (Schwierig)

Aufgabe "our_list::merge_sorted" (Schwierig)

Falls all diese Klassen, Pointers und dynamischen Datenzuweisungen nicht schon schwierig genug für euch waren, lasst uns auch noch Rekursion dazunehmen!

Merge-Sort

Merge-Sort

- Goal: Sort an **arbitrary** array as **quickly** as possible.

5	2	8	7	7	3	1
---	---	---	---	---	---	---



1	2	3	5	7	7	8
---	---	---	---	---	---	---

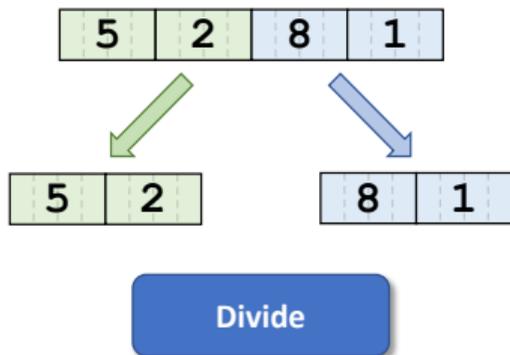
Merge-Sort

- Idea: **Divide and Conquer**

Merge-Sort

- Idea: **Divide and Conquer**

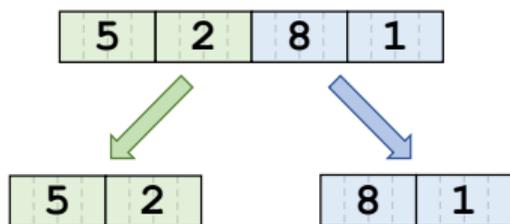
1. Split whole array into two parts. (**Divide**)



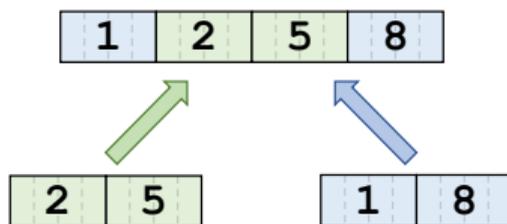
Merge-Sort

- Idea: **Divide and Conquer**

1. Split whole array into two parts. (**Divide**)
2. Then sort these and combine them. (**Conquer**)



Divide

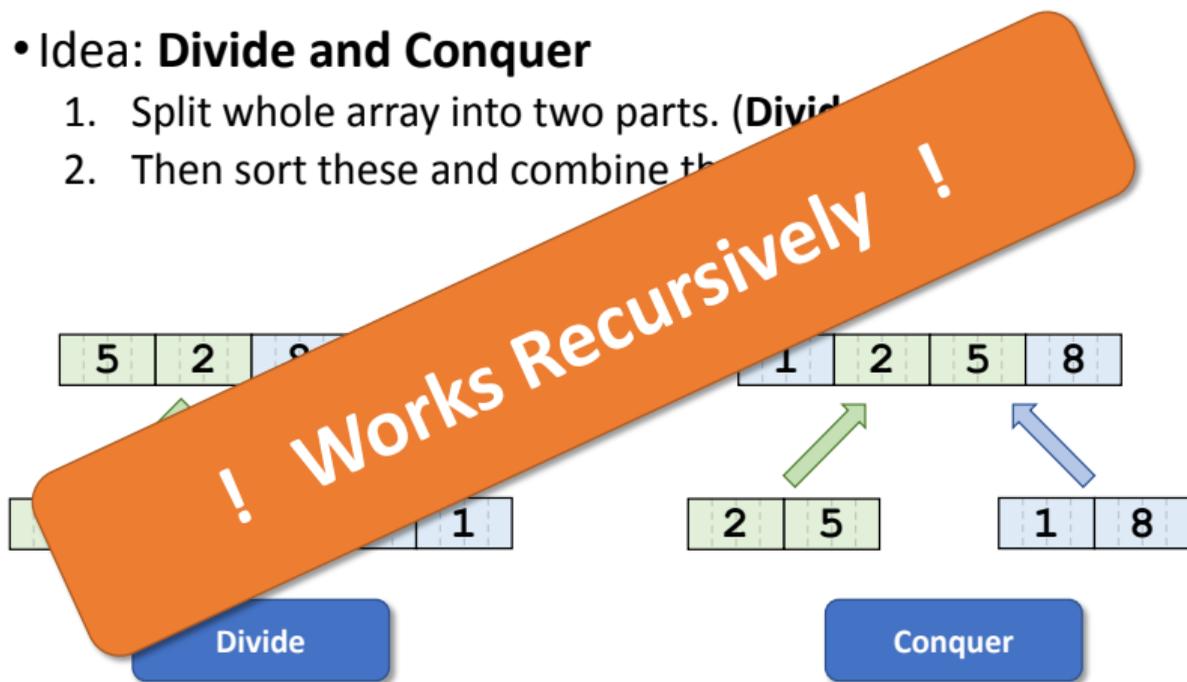


Conquer

Merge-Sort

- Idea: **Divide and Conquer**

1. Split whole array into two parts. (**Divide**)
2. Then sort these and combine them. (**Conquer**)



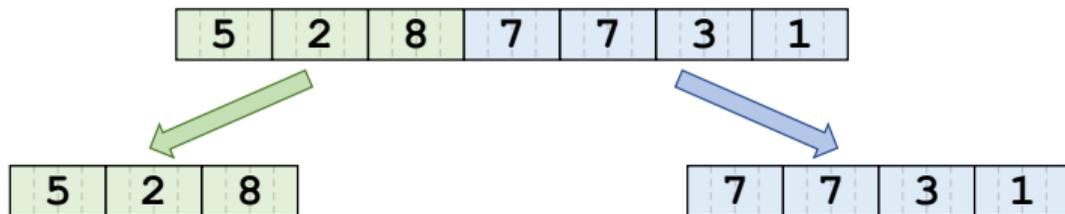
Merge-Sort

- Divide:

5	2	8	7	7	3	1
---	---	---	---	---	---	---

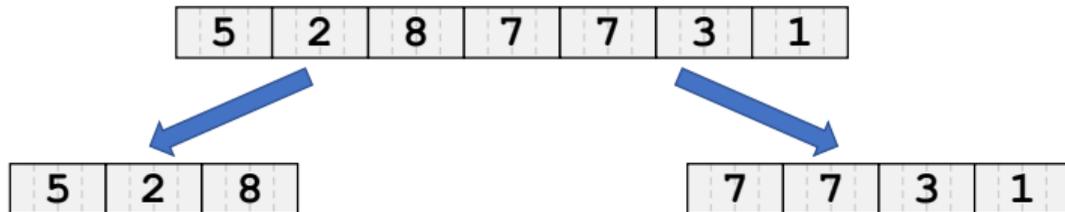
Merge-Sort

- Divide:



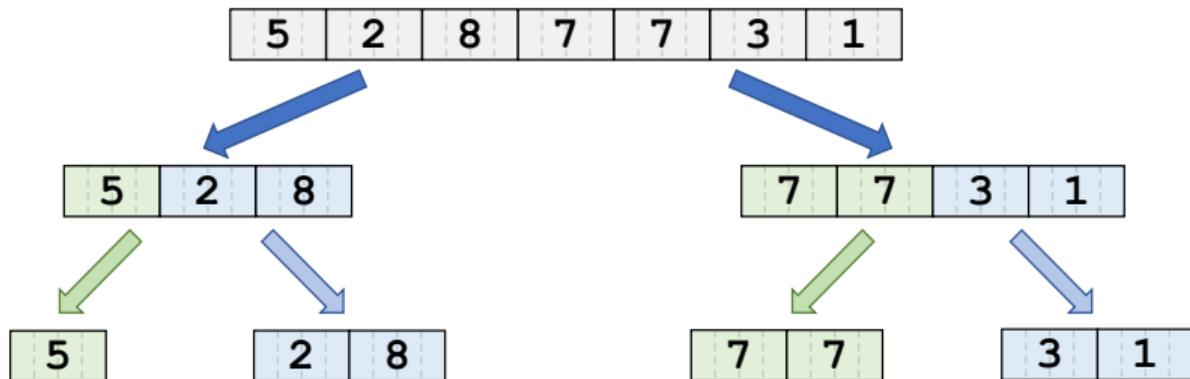
Merge-Sort

- Divide:



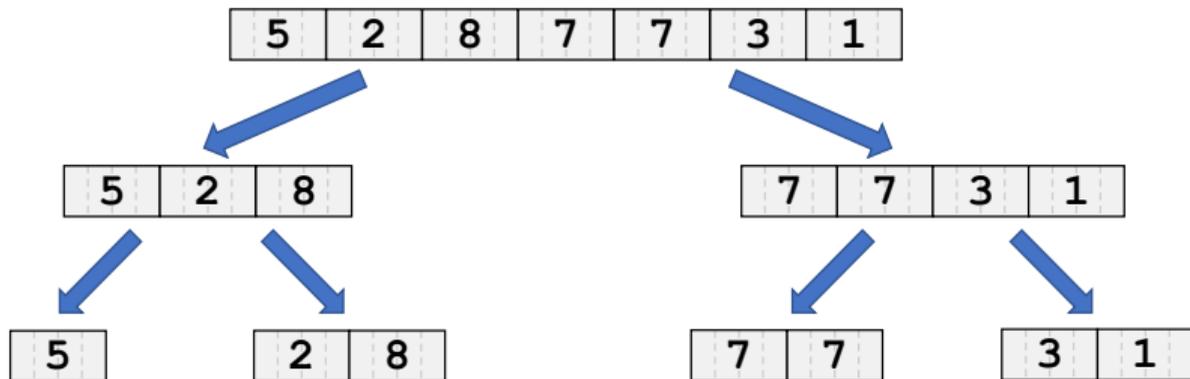
Merge-Sort

- Divide:



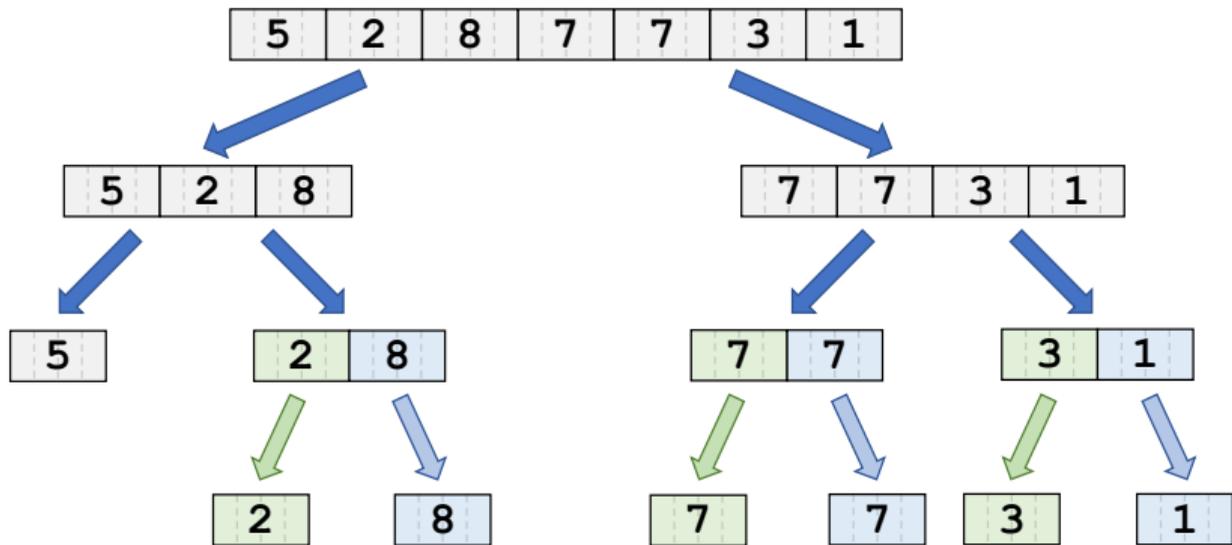
Merge-Sort

- Divide:



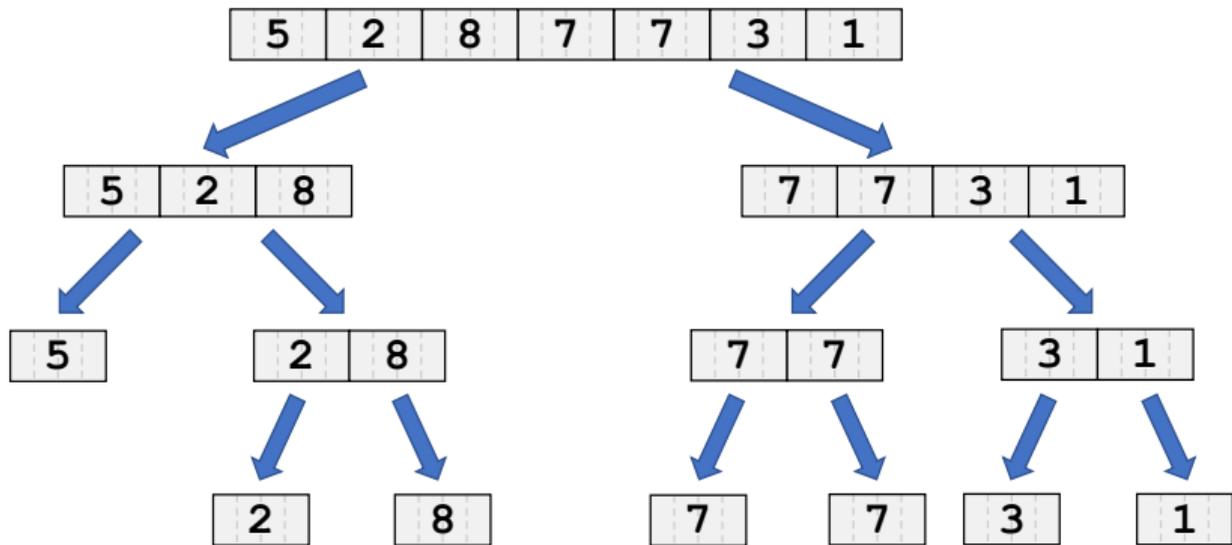
Merge-Sort

- Divide:



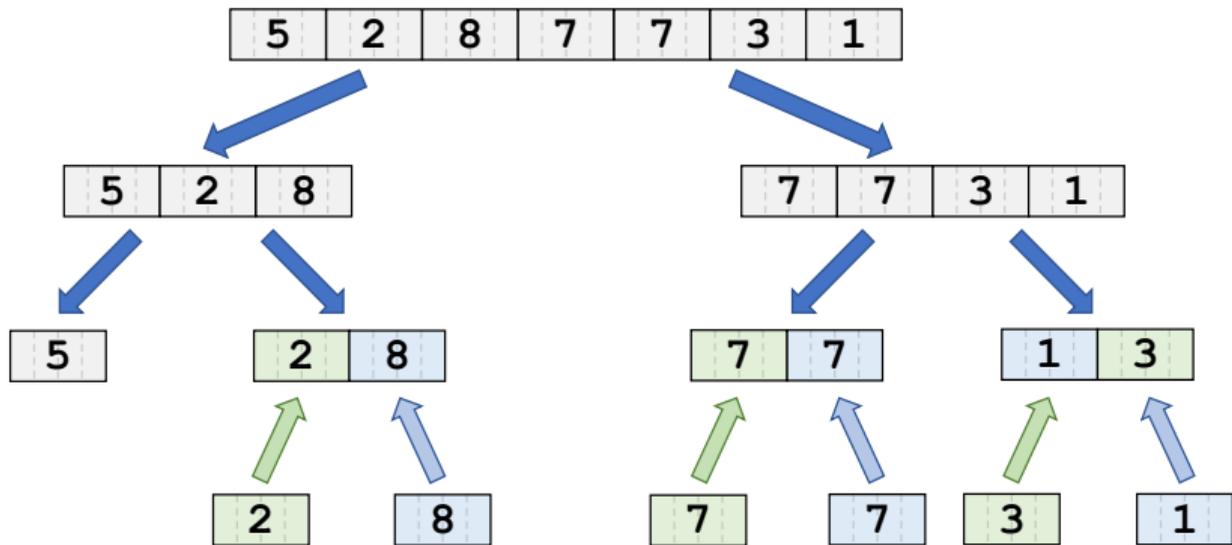
Merge-Sort

- Divide:



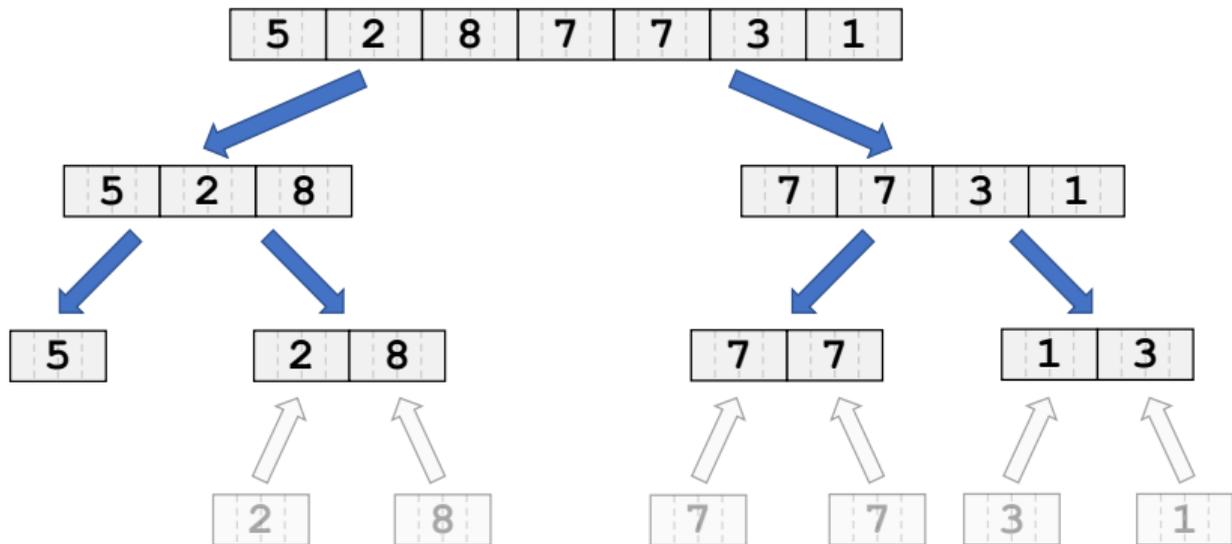
Merge-Sort

- Conquer:



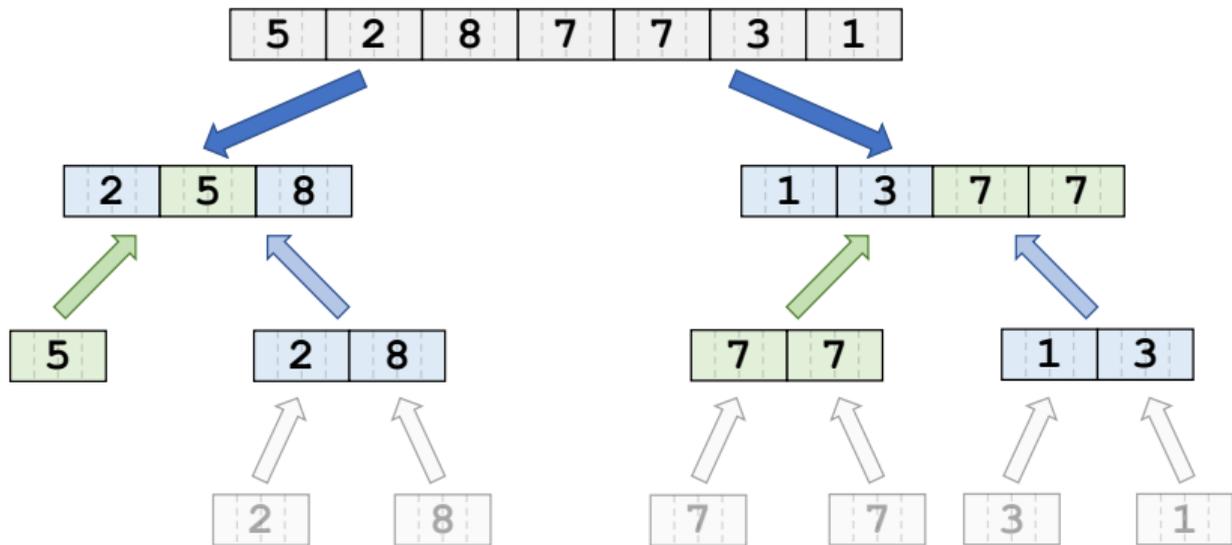
Merge-Sort

- Conquer:



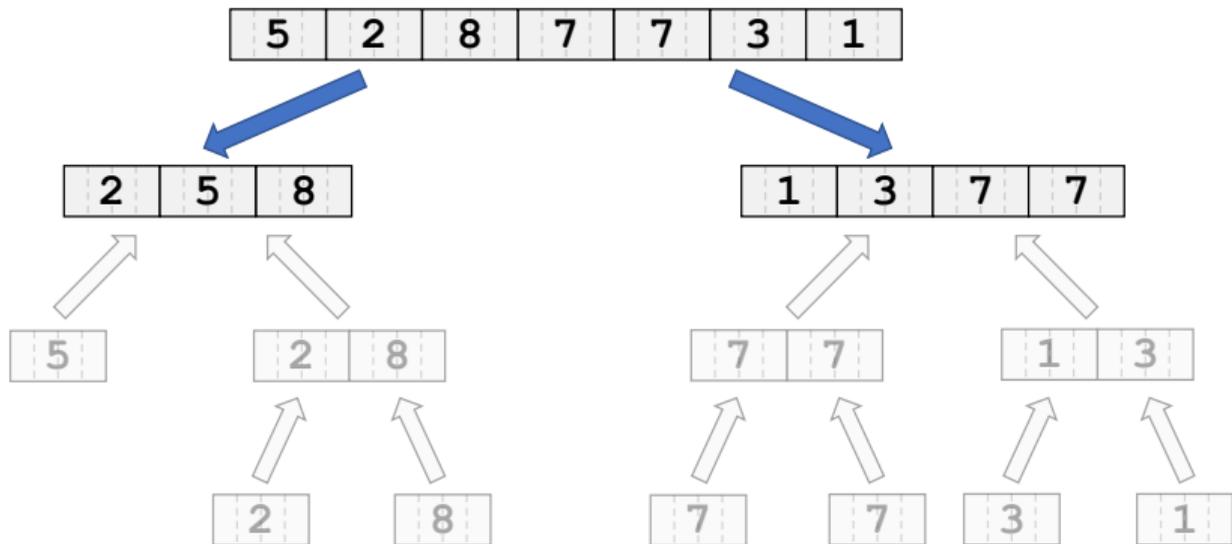
Merge-Sort

- Conquer:



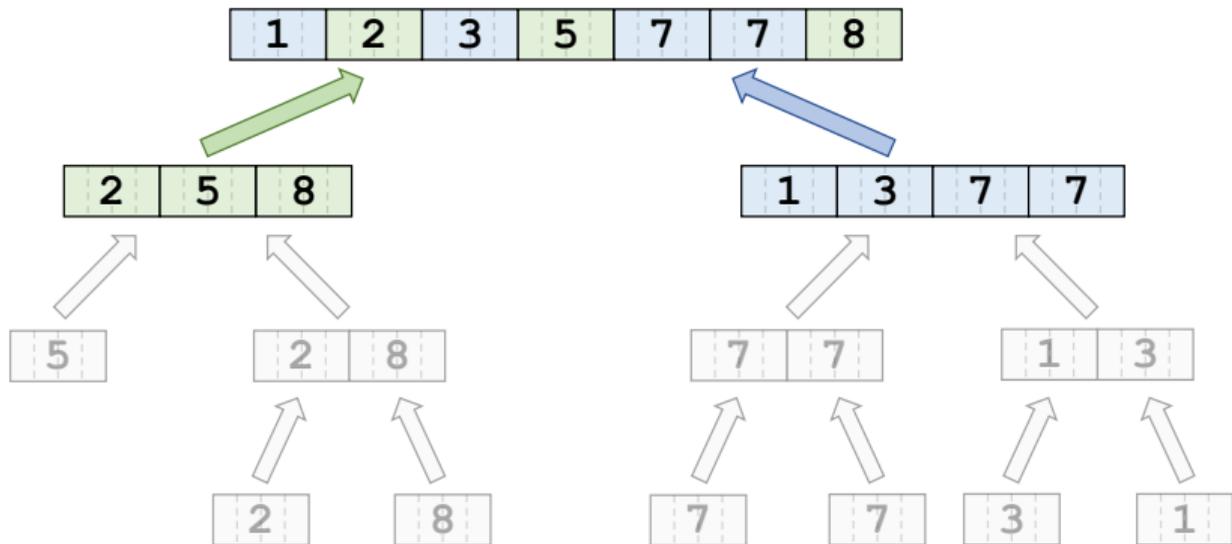
Merge-Sort

- Conquer:



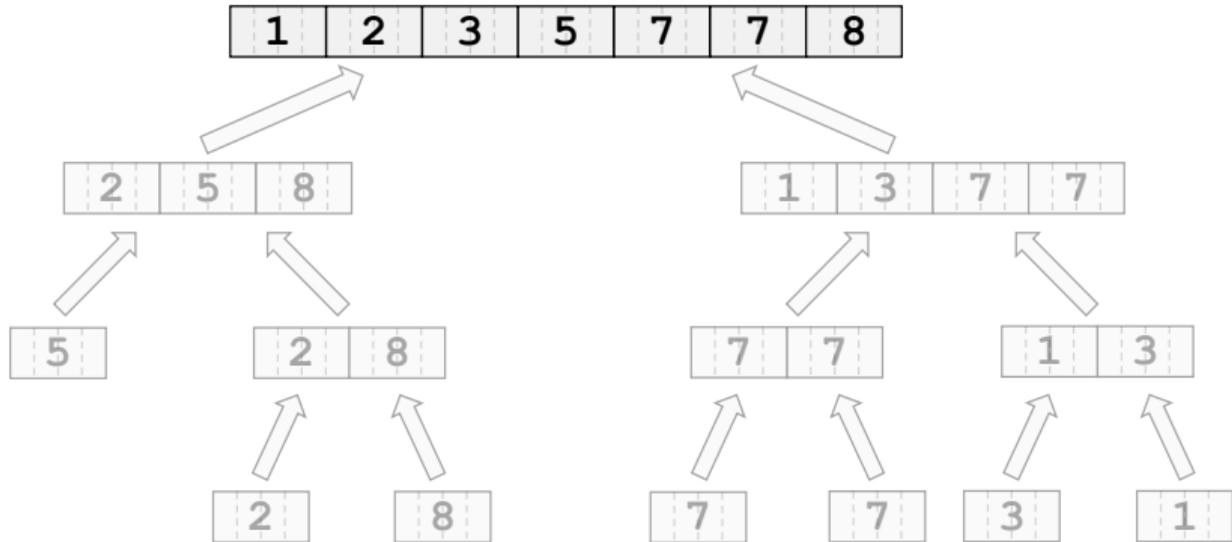
Merge-Sort

- Conquer:



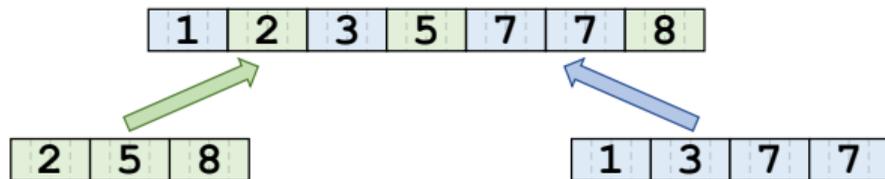
Merge-Sort

- Conquer:



Merge-Step

- How does

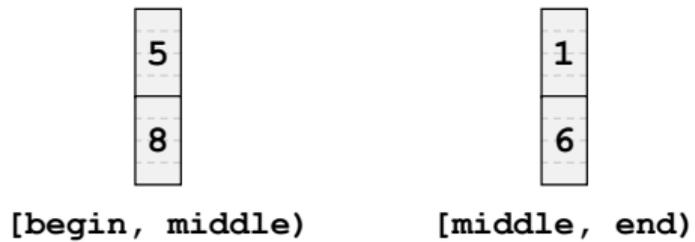


work?

- Card-player's trick:
Remove smaller «top card» (see next slides)

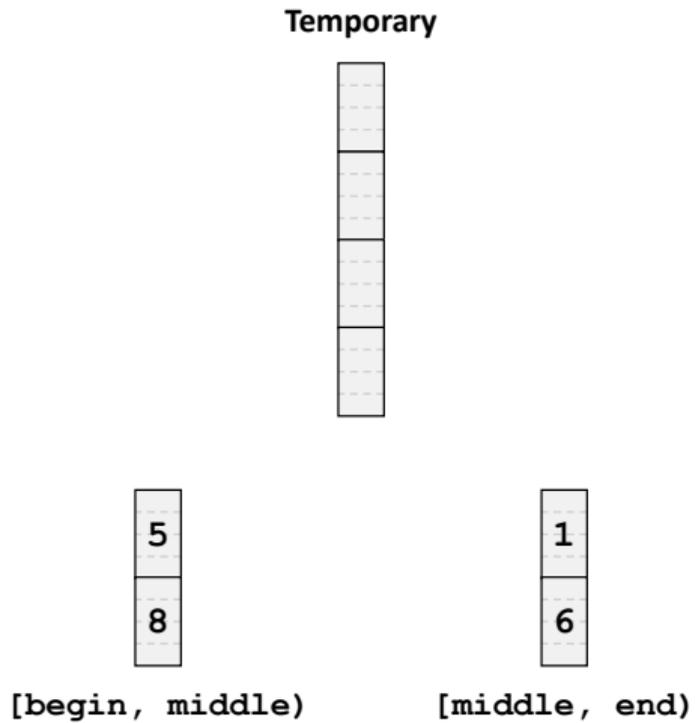
Merge-Sort – Exercise 1

- Idea:



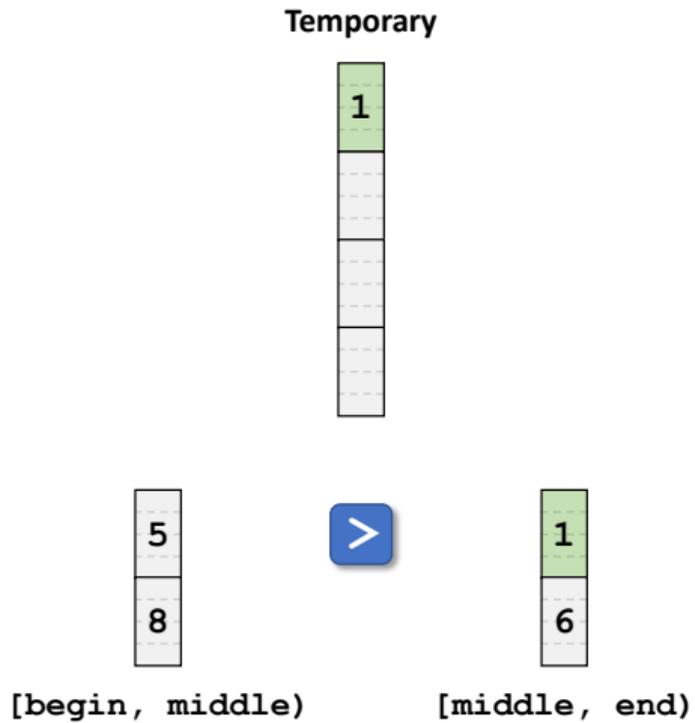
Merge-Sort – Exercise 1

- Idea:



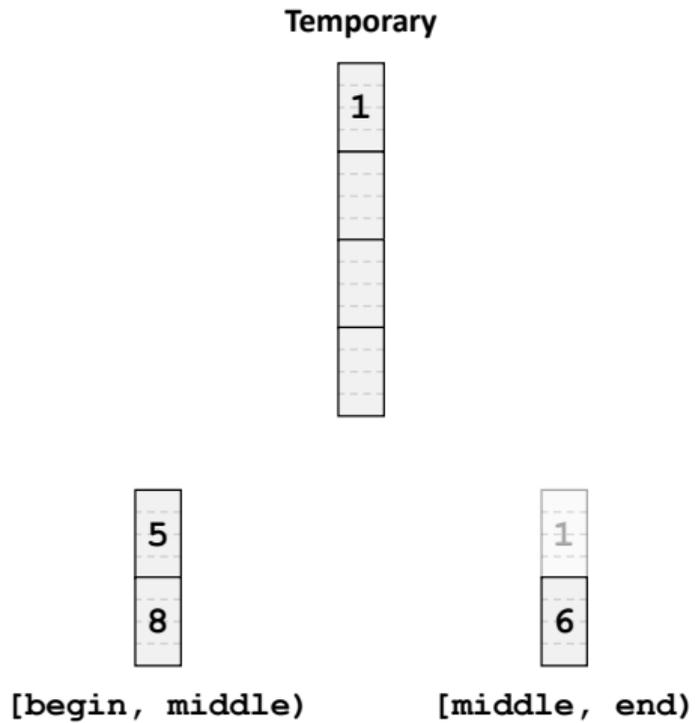
Merge-Sort – Exercise 1

- Idea:



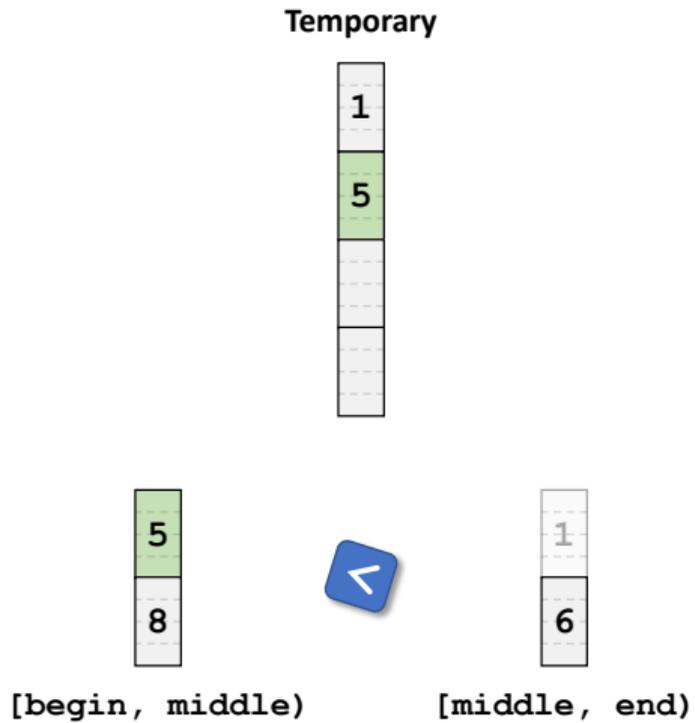
Merge-Sort – Exercise 1

- Idea:



Merge-Sort – Exercise 1

- Idea:



Merge-Sort – Exercise 1

- Idea:

Temporary



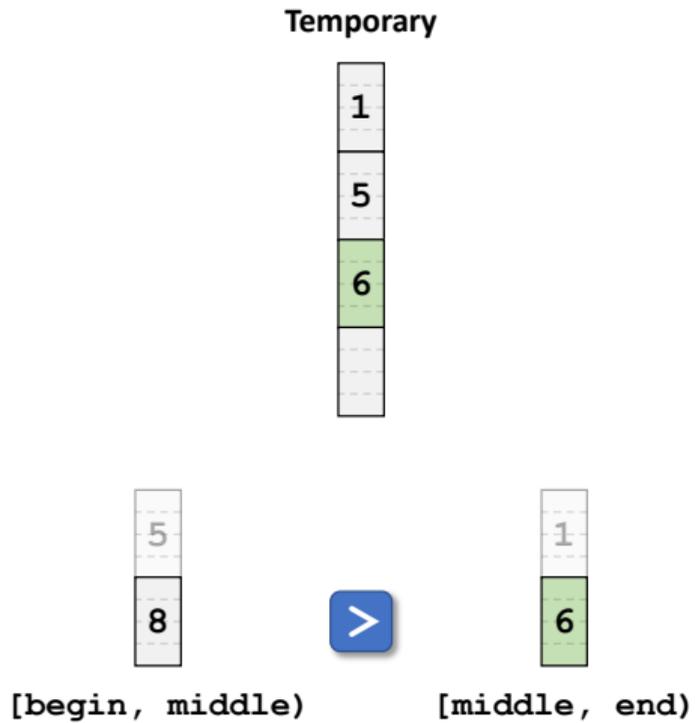
[begin, middle)



[middle, end)

Merge-Sort – Exercise 1

- Idea:



Merge-Sort – Exercise 1

- Idea:

Temporary



[begin, middle)



[middle, end)

Merge-Sort – Exercise 1

- Idea:

Temporary



[begin, middle)



[middle, end)

Merge-Sort – Exercise 1

- Idea:

Temporary



[begin, middle)

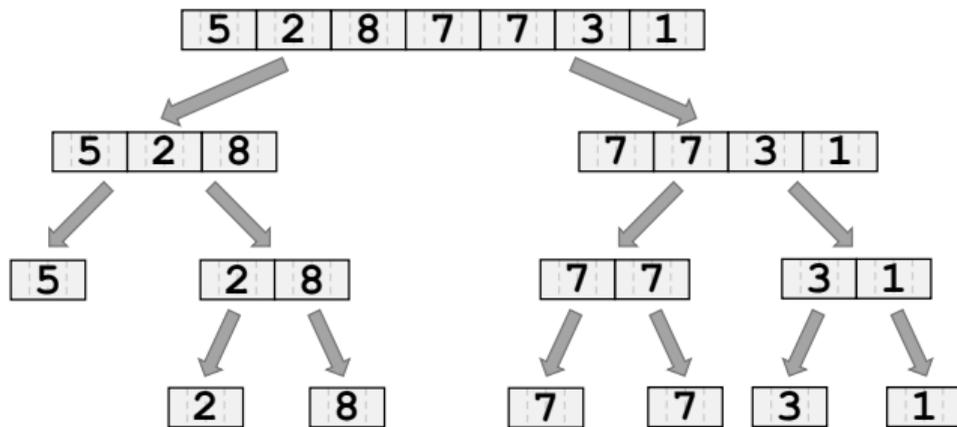


[middle, end)

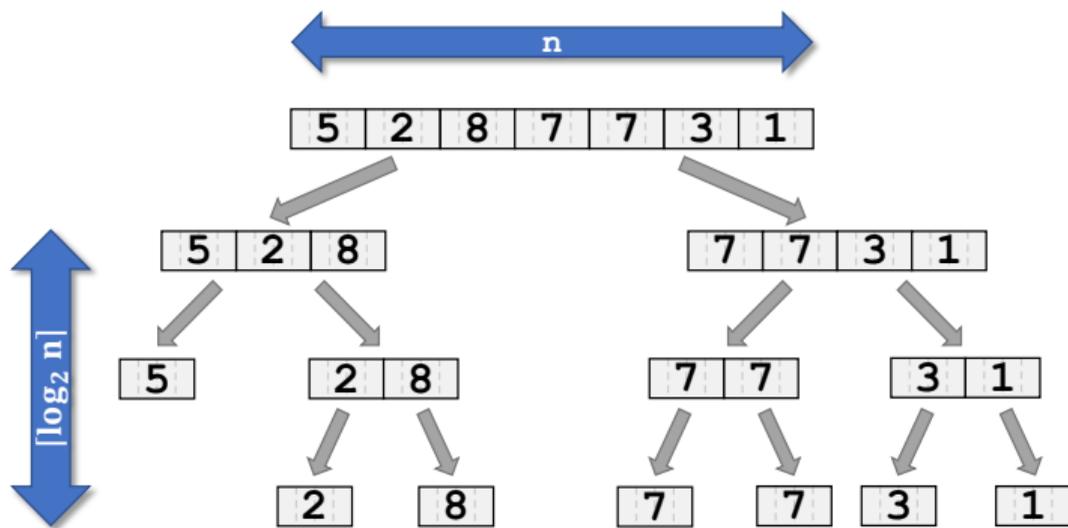
Runtime

(Intuition)

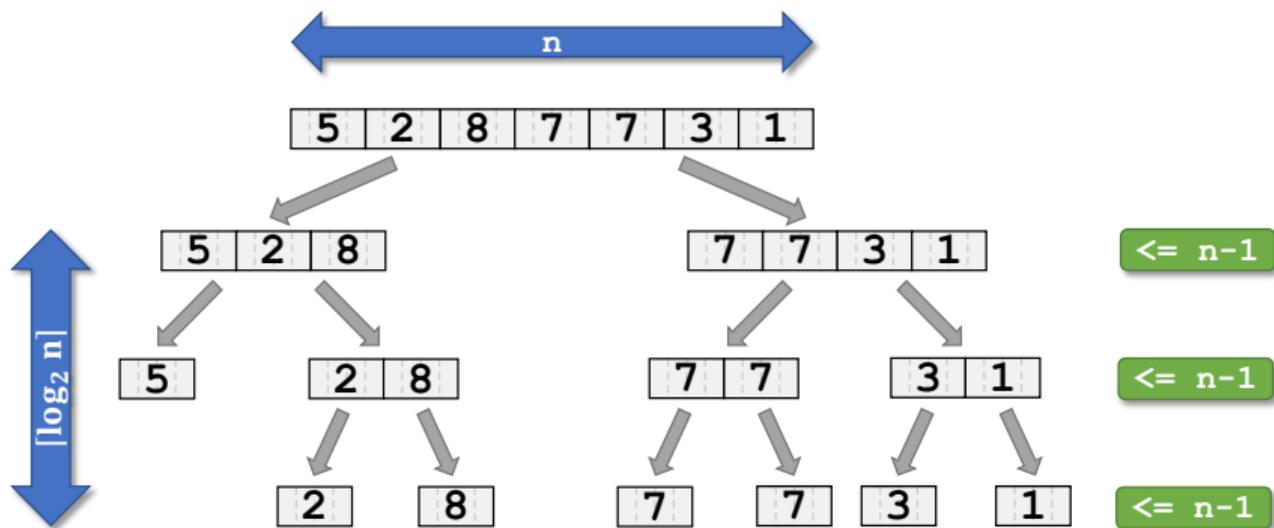
Alternative Proof



Alternative Proof

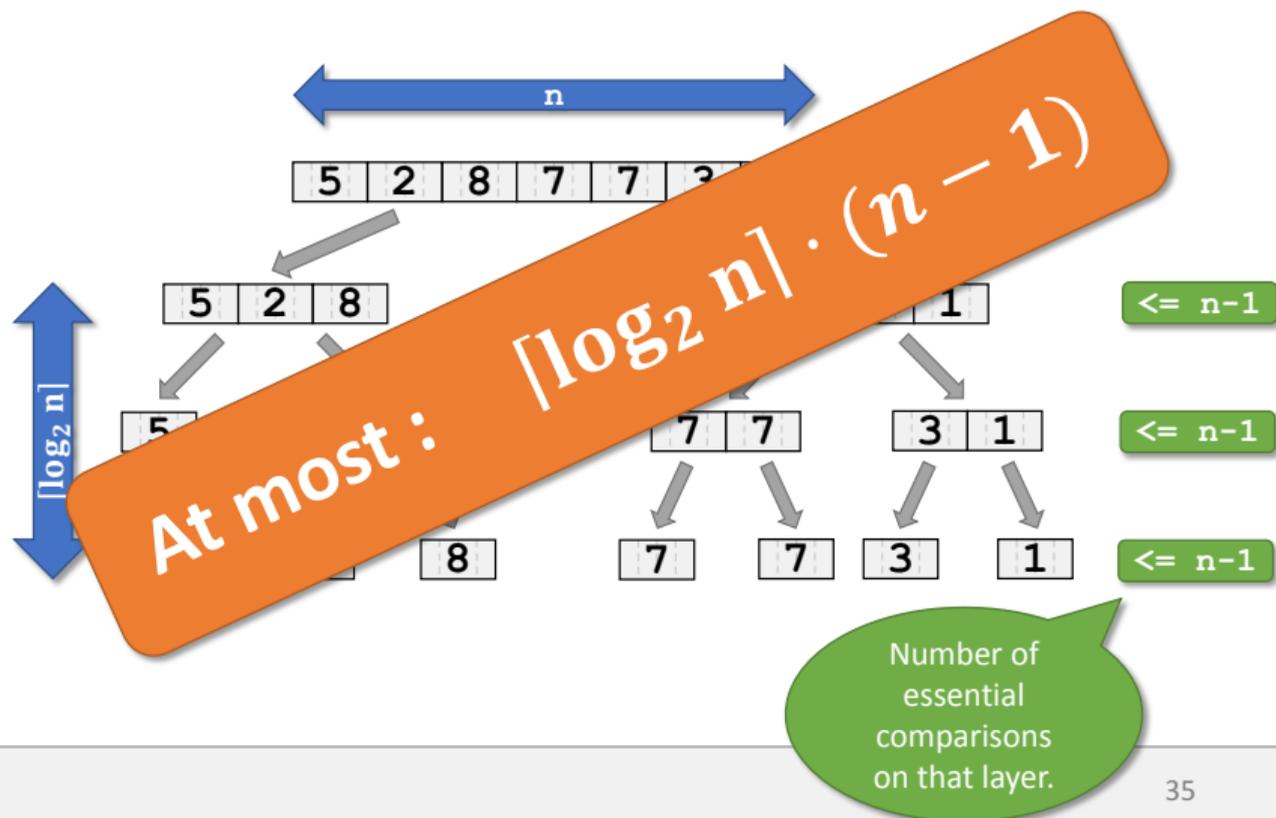


Alternative Proof



Number of essential comparisons on that layer.

Alternative Proof



Aufgabe "our_list::merge_sorted" (Schwierig)

Aufgabe "our_list::merge_sorted" (Schwierig)

- Öffnet "our_list::merge_sorted" auf **code expert**

Aufgabe "our_list::merge_sorted" (Schwierig)

- Öffnet "our_list::merge_sorted" auf **code expert**
- Überlegt euch, wie ihr das Problem mit Stift und Papier angehen würdet

Aufgabe "our_list::merge_sorted" (Schwierig)

- Öffnet "our_list::merge_sorted" auf **code expert**
- Überlegt euch, wie ihr das Problem mit Stift und Papier angehen würdet
- Programmiert eine Lösung (optional in Gruppen)

Aufgabe "our_list::merge_sorted" (Lösung)

Aufgabe "our_list::merge_sorted" (Lösung)

Siehe **code expert**

Fragen/Unklarheiten?

9. Outro

Allgemeine Fragen?

Bis zum nächsten Mal

Schöne Woche noch!