Exercise Session — Computer Science — 10

Pointer-related Operators, References vs. Pointers, Iterators, `this->`, Dynamic Memory

Overview

Today's Plan

Follow-up

& vs *

References vs Pointers

`this->`

Dynamic Data Structures & Iterators

- `Our_list` Main Material
- `Our_list` Bonus Material

`n.ethz.ch/~iopopa`

Link to Webpage

Send an e-Mail

1. Follow-up

Follow-up from last session

- What happens in a class if you don’t explicitly define a default constructor?

Follow-up from last session

- What happens in a class if you don’t explicitly define a default constructor?
- The compiler provides a default constructor - **only if no constructors are explicitly declared in the class**

```
class MyClass {
public:
    int x;
    std::string str;
};

int main() {
    MyClass obj;  // Compiler-generated default constructor
    std::cout << obj.x << " " << obj.str; //Prints 0 and ""
}
```

Follow-up from last session

- What happens in a class if you don't explicitly define a default constructor?
- However, if any constructor is defined, the compiler does **not** generate the default constructor for you:

```cpp
class MyClass {
public:
    MyClass(int val) : x(val) {}  // Parameterized constructor
    int x;
};
int main() {
    MyClass obj; // Compiler Error!!
}
```

- If you still need a default constructor, you must explicitly declare it: `MyClass() = default;`

2. & vs *

The meanings of `&`

The symbol `&` has many meanings in C++ which can be very confusing
It has *3 different meanings* depending on its position in code:

The meanings of `&`

The symbol `&` has many meanings in C++ which can be very confusing
It has *3 different meanings* depending on its position in code:

The meaning of `&`

The meanings of `&`

The symbol `&` has many meanings in C++ which can be very confusing
It has *3 different meanings* depending on its position in code:

The meaning of `&`

1. as AND-operator

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

The meanings of `&`

The symbol `&` has many meanings in C++ which can be very confusing
It has *3 different meanings* depending on its position in code:

The meaning of `&`

1. as AND-operator
   ```
   bool z = x && y;
   ```
2. to declare a variable as an alias
   ```
   int& y = x;
   ```

The meanings of `&`

The symbol `&` has many meanings in C++ which can be very confusing
It has *3 different meanings* depending on its position in code:

The meaning of `&`

1. as AND-operator
   ```
   bool z = x && y;
   ```
2. to declare a variable as an alias
   ```
   int& y = x;
   ```
3. to get the address of a variable (address-operator)
   ```
   int *ptr_a = &a;
   ```

The meanings of $*$

Ditto with the symbol $\&$.

The meaning of $*$

The meanings of *

Ditto with the symbol `&`.

The meaning of *

1. as (arithmetic) multiplication-operator

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

The meanings of *

Ditto with the symbol `&`.

The meaning of *

1. as (arithmetic) multiplication-operator
   ```
   z = x * y;
   ```
2. to declare a pointer variable
   ```
   int* ptr_a = &a;
   ```

The meanings of *

Ditto with the symbol `&`.

The meaning of *

1. as (arithmetic) multiplication-operator
   ```
   z = x * y;
   ```
2. to declare a pointer variable
   ```
   int* ptr_a = &a;
   ```
3. to access a variable via its pointer (dereference-operator)
   ```
   int a = *ptr_a;
   ```

Questions?

3. References vs Pointers

Pointer Basics

Try program[1] tracing this in detail

```
int main() {

    int a = 5;
    int* x = &a;
    *x = 6;

    return 0;
}
```

[1]Full trace available here

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 program and write expected output, if the function is called

References

```cpp
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 program and write expected output, if the function is called

```
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 program and write expected output, if the function is called

Pointers

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


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

Trace program and write expected output, if the function is called

```
1 2 0x7ffe4d1fb904 0x7ffe4d1fb904
```

Pointers

```cpp
void pointers(){
    int  a = 1;
    int  b = 2;
    int* x = &a;
    int* y = x;


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

Trace program and write expected output, if the function is called

```
1 2 0x7ffe4d1fb904 0x7ffe4d1fb904
```

(The addresses could be different each time when called!)

Pointers und Adressen

```
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 program and write expected output, if the function is called

Pointers und Adressen

```cpp
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 program and write expected output, if the function is called

```
5
5
0x7ffe4d1fb914
0x7ffe4d1fb914
```

(The addresses could be different each time when called!)

Questions?

4. `this->`

What the f*&k is `this->`?

The meaning of `this->`

`this->` has two parts

What the f*&k is `this->`?

The meaning of `this->`

`this->` has two parts

- `this`
    - is a pointer to the current object (class or struct `T`)

What the f∗&k is `this->`?

The meaning of `this->`

`this->` has two parts

- `this`
  - is a pointer to the current object (class or struct `T`)
  - so it is of type `T*`

What the f*&k is `this->`?

The meaning of `this->`

`this->` has two parts

- `this`
  - is a pointer to the current object (class or struct `T`)
  - so it is of type `T*`
- `->`
  - is a cool looking operator

What the f∗&k is `this->`?

The meaning of `this->`

`this->` has two parts

- `this`
  - is a pointer to the current object (class or struct `T`)
  - so it is of type `T*`
- `->`
  - is a cool looking operator
  - `this->member_element` is equivalent to `*(this).member_element`
  - the arrow operator dereferences a pointer to an object in order to access one of its members (functions or variables)

Example

What is `this` used here for?

```cpp
struct WeirdNumber {

    int number;

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

Example

What is `this` used here for?

```
struct WeirdNumber {

    int number;

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

To distinguish between the two `number` variables with the same name

Example

```
int main(){

    WeirdNumber a = {42}; // list initialization for structs
    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;
}
```

5. Dynamic Data Structures & Iterators

5. Dynamic Data Structures & Iterators

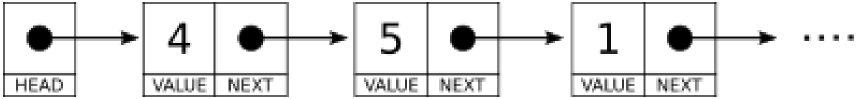
5.1. `Our_list` Main Material

`our_list`

We will implement (parts of) our own linked-list

our_list

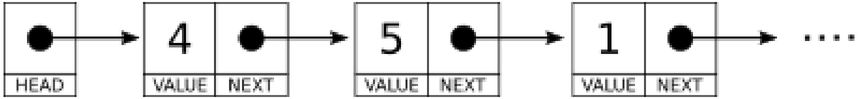
We will implement (parts of) our own linked-list



- A list is comprised of "blocks" of `lnodes` with one `lnode` always pointing to the next

`our_list`

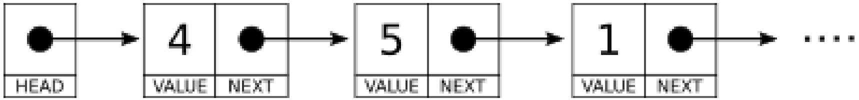
We will implement (parts of) our own linked-list



- A list is comprised of "blocks" of `lnodes` with one `lnode` always pointing to the next
- But what even is an `lnode`?

`our_list`

We will implement (parts of) our own linked-list



- A list is comprised of "blocks" of `lnodes` with one `lnode` always pointing to the next
- But what even is an `lnode`?
- Answer: A struct made up of an `int value` and an `lnode`-pointer

`our_list`

First task: Implement a constructor that initializes a new list with iterators

`our_list`

First task: Implement a constructor that initializes a new list with iterators

- We want to be able to write `our_list my_list(begin, end);`

`our_list`

First task: Implement a constructor that initializes a new list with iterators

- We want to be able to write `our_list my_list(begin, end);`
- Idea: Use the iterators to add new `lnodes` to the list

`our_list`

First task: Implement a constructor that initializes a new list with iterators

- We want to be able to write `our_list my_list(begin, end);`
- Idea: Use the iterators to add new `lnodes` to the list
- How can we access the different elements?

`our_list`

First task: Implement a constructor that initializes a new list with iterators

- We want to be able to write `our_list my_list(begin, end);`
- Idea: Use the iterators to add new `lnodes` to the list
- How can we access the different elements?
  - Access to Value of the `lnode` that the iterator is pointing to:

```
*it
```

`our_list`

First task: Implement a constructor that initializes a new list with iterators

- We want to be able to write `our_list my_list(begin, end);`
- Idea: Use the iterators to add new `lnodes` to the list
- How can we access the different elements?
  - Access to Value of the `lnode` that the iterator is pointing to:

    `*it`

  - Next `lnode` in line:

    `node->next`

`our_list`

First task: Implement a constructor that initializes a new list with iterators

- We want to write `our_list my_list(begin, end);`
- Idea: Use the iterators to add new `lnodes` to the list
- How can we access the different elements?
  - Access to Value of the `lnode` that the iterator is pointing to:

    ```
    *it
    ```
  - Next `lnode` in line:

    ```
    node->next
    ```
  - Create a pointer to a new `lnode`:

    ```
    new lnode{value, pointer}
    ```

  Remember: `new T` returns a `T*`

our_list: class our_list

```
class our_list {

        struct lnode {
            // ...
        };

        lnode* head;

    public:

        class const_iterator {
            // ...
      };

      // member functions
};
```

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: member functions

```
//                         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;
```

Exercise "`our_list::init`"

Exercise "`our_list::init`"

- Open "`our_list::init`" on **code** expert

Exercise "`our_list::init`"

- Open "`our_list::init`" on **code** expert
- Think about how you would approach the problem with pen and paper

Exercise "`our_list::init`"

- Open "`our_list::init`" on **code** expert
- Think about how you would approach the problem with pen and paper
- Implement a solution (optionally in groups)

Exercise "`our_list::init`" (Solution)

Exercise "our_list::init" (Solution)

```
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;
    }
}
```

Questions?

`our_list`

Second task: Implement a method of the class "`our_list`" that swaps a node with the next one

`our_list`

Second task: Implement a method of the class "`our_list`" that swaps a node with the next one

- You can use a similar approach to other swap functions (i.e. with a temporary variable `tmp`)

`our_list`

Second task: Implement a method of the class "`our_list`" that swaps a node with the next one

- You can use a similar approach to other swap functions (i.e. with a temporary variable `tmp`)
- However:
  - Use Pointers
  - What happens in the case of "0" (when the head pointer should be swapped)?
  - How can you avoid suddenly accessing memory that is not yours?

Exercise "`our_list::swap`"

Exercise "`our_list::swap`"

- Open "`our_list::swap`" on **code** expert

Exercise "`our_list::swap`"

- Open "`our_list::swap`" on **code** expert
- Think about how you would approach the problem with pen and paper

Exercise "`our_list::swap`"

- Open "`our_list::swap`" on **code** expert
- Think about how you would approach the problem with pen and paper
- Implement a solution (optionally in groups)

Exercise "`our_list::swap`" (Solution)

Exercise "our_list::swap" (Solution)

```
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 {/* ... */}
```

Exercise "our_list::swap" (Solution)

```
  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
```

Questions?

5. Dynamic Data Structures & Iterators

5.2. `Our_list` Bonus Material

Exercise "`our_list::extend`"

Exercise "`our_list::extend`"

- Open "`our_list::extend`" on **code** expert

Exercise "`our_list::extend`"

- Open "`our_list::extend`" on **code** expert
- Think about how you would approach the problem with pen and paper

Exercise "`our_list::extend`"

- Open "`our_list::extend`" on **code** expert
- Think about how you would approach the problem with pen and paper
- Implement a solution (optionally in groups)

Exercise "`our_list::extend`" (Solution)

Exercise "our_list::extend" (Solution)

```
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;
  }
}
```

Questions?

Exercise "`our_list::merge_sorted`" (Difficult)

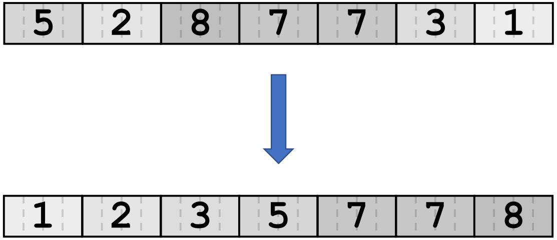
Exercise "`our_list::merge_sorted`" (Difficult)

In case all these classes, pointers, dynamic data allocation wasn’t hard enough for you, let’s throw recursion to the mix too!

Merge-Sort

Merge-Sort

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

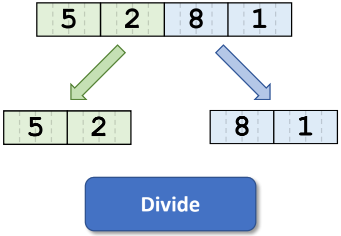


Merge-Sort

- Idea: **Divide and Conquer**

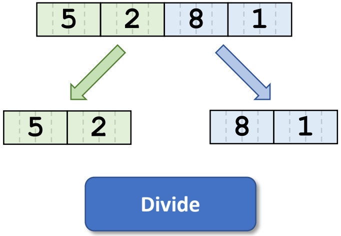
Merge-Sort

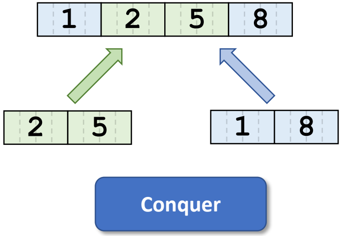
- Idea: **Divide and Conquer**
    1. Split whole array into two parts. (**Divide**)



Merge-Sort

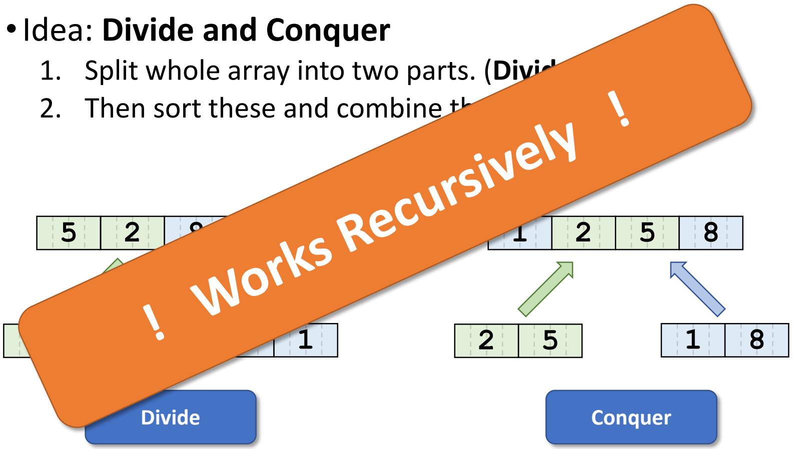
- Idea: **Divide and Conquer**
  1. Split whole array into two parts. (**Divide**)
  2. Then sort these and combine them. (**Conquer**)





Merge-Sort

- Idea: **Divide and Conquer**
  1. Split whole array into two parts. (**Divid**
  2. Then sort these and combine th



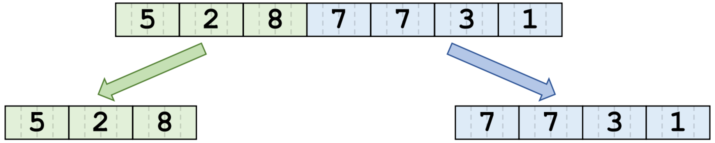
Merge-Sort

- Divide:

5	2	8	7	7	3	1

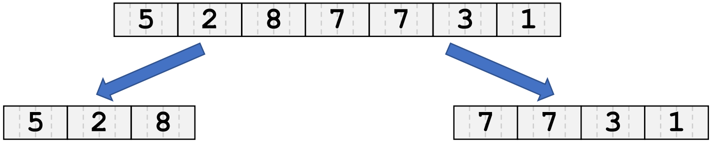
Merge-Sort

- Divide:



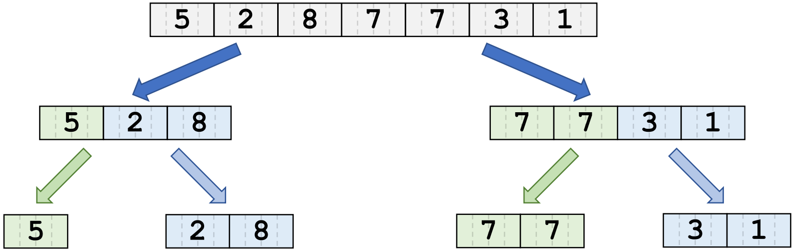
Merge-Sort

- Divide:



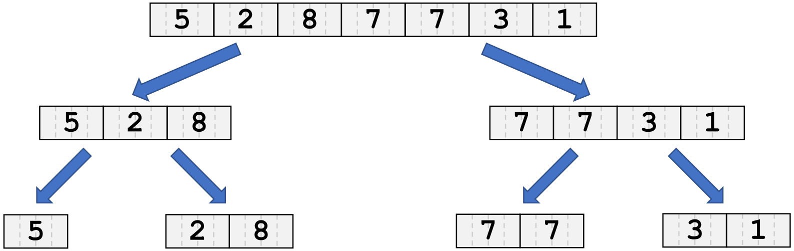
Merge-Sort

- Divide:



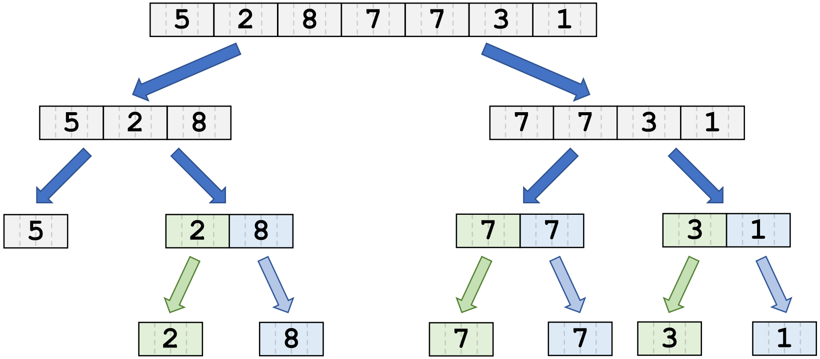
Merge-Sort

- Divide:



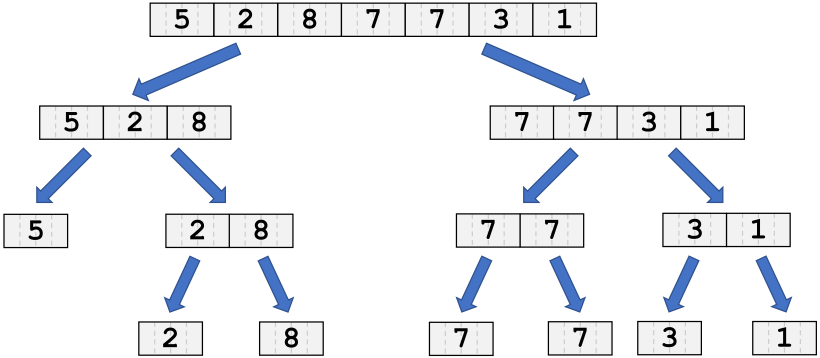
Merge-Sort

- Divide:



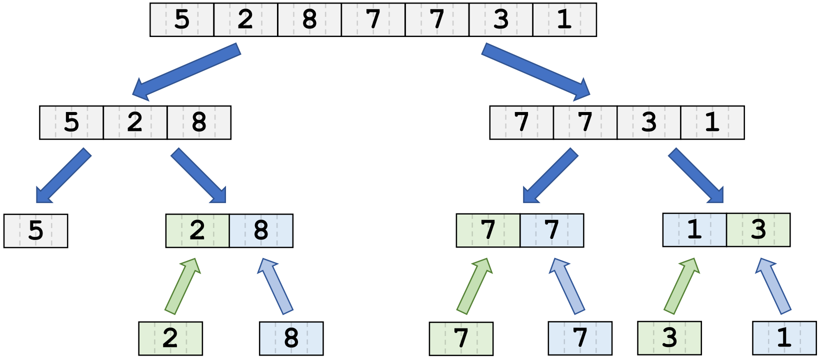
Merge-Sort

- Divide:



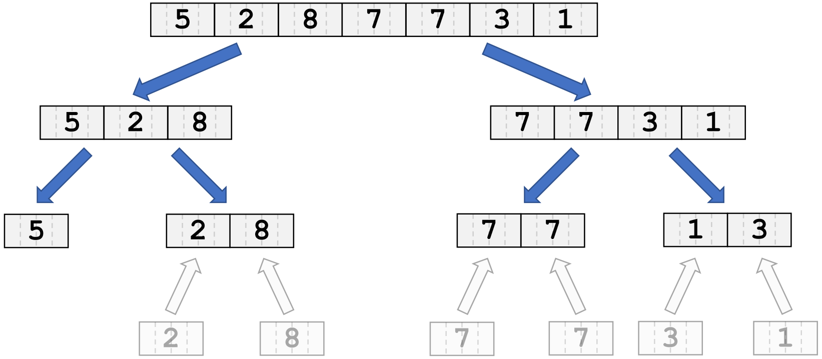
Merge-Sort

- Conquer:



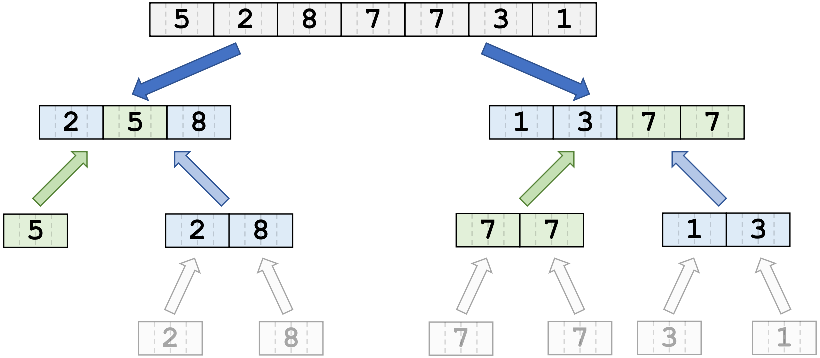
Merge-Sort

- Conquer:



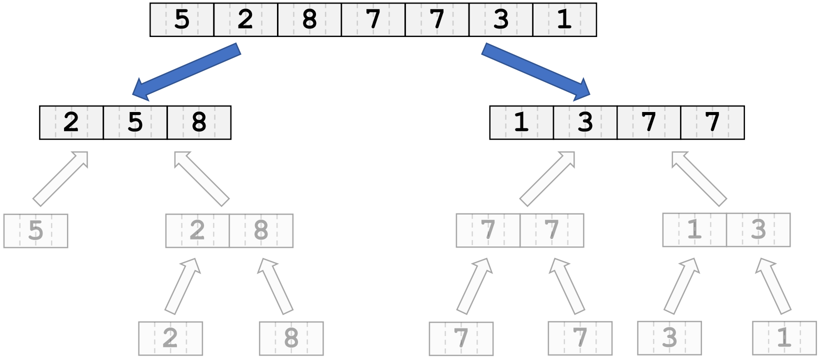
Merge-Sort

- Conquer:



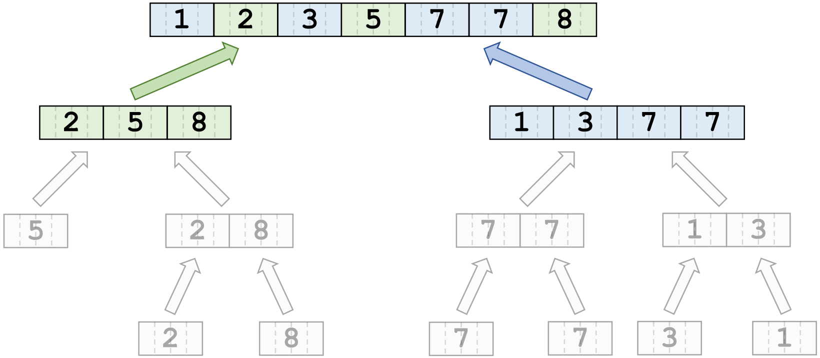
Merge-Sort

- Conquer:



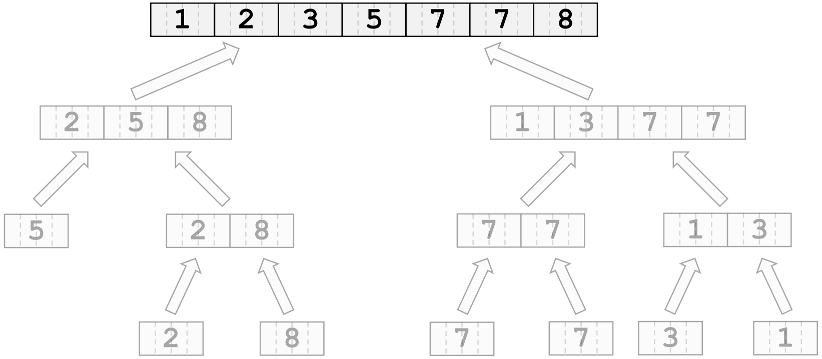
Merge-Sort

- Conquer:



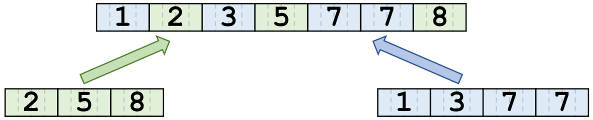
Merge-Sort

- Conquer:



Merge-Step

- How does

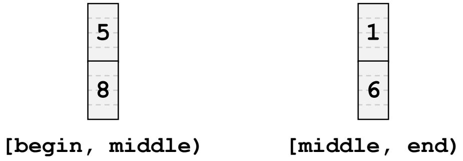


  work?

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

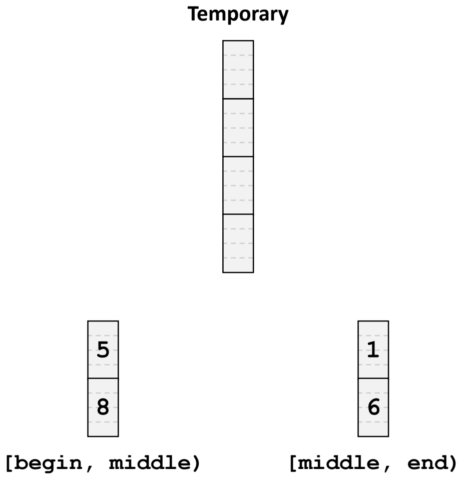
Merge-Sort – Exercise 1

- Idea:



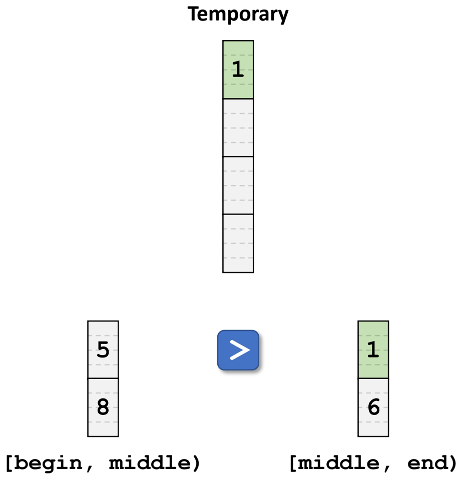
Merge-Sort – Exercise 1

- Idea:



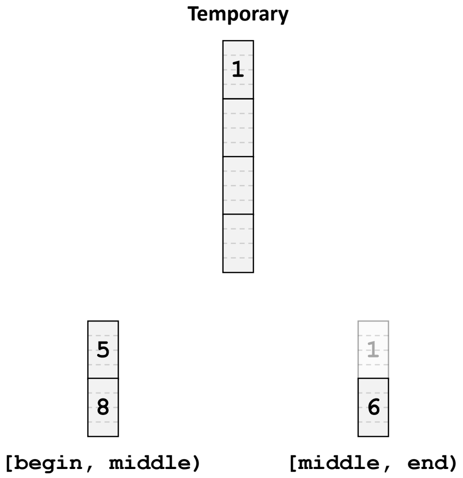
Merge-Sort – Exercise 1

- Idea:



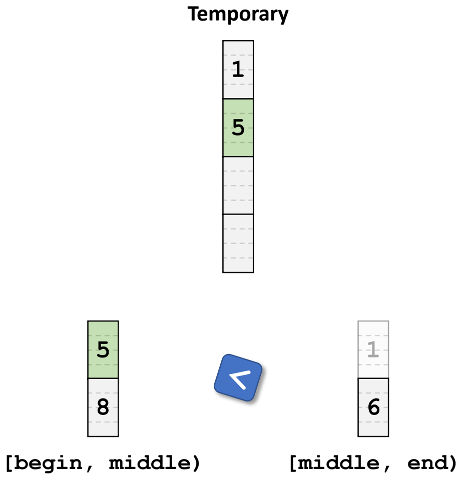
Merge-Sort – Exercise 1

- Idea:



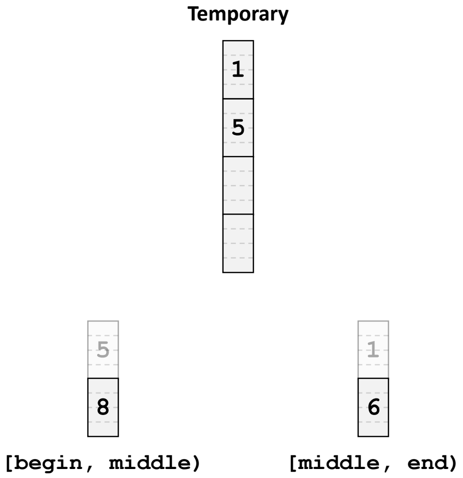
Merge-Sort – Exercise 1

- Idea:



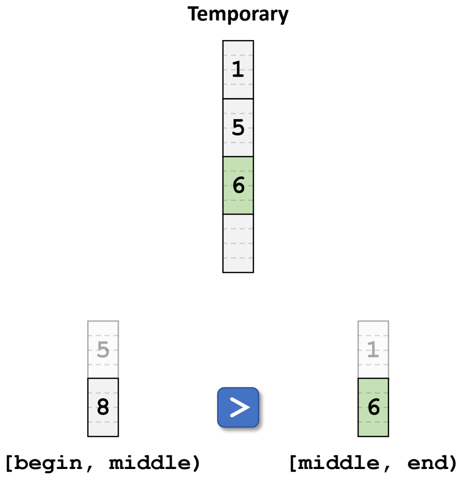
Merge-Sort – Exercise 1

- Idea:



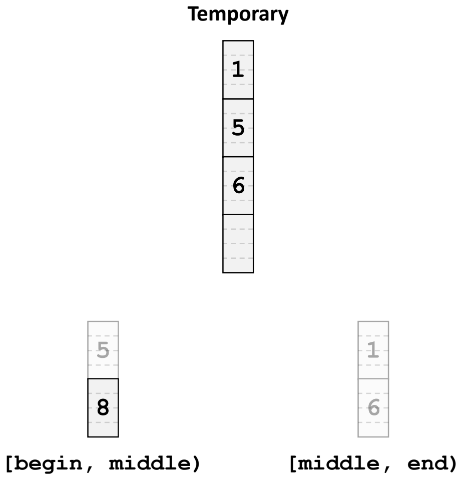
Merge-Sort – Exercise 1

- Idea:



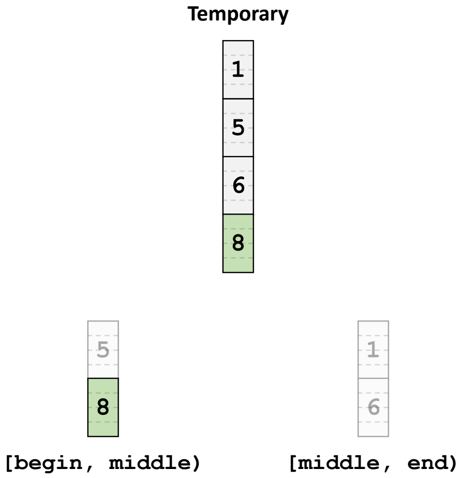
Merge-Sort – Exercise 1

- Idea:



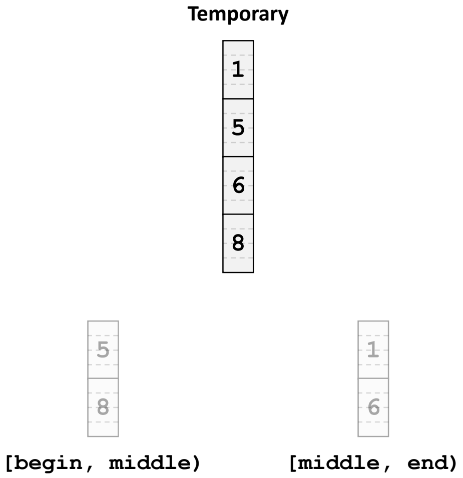
Merge-Sort – Exercise 1

- Idea:



Merge-Sort – Exercise 1

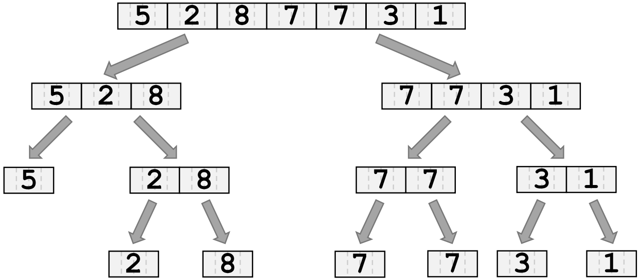
- Idea:



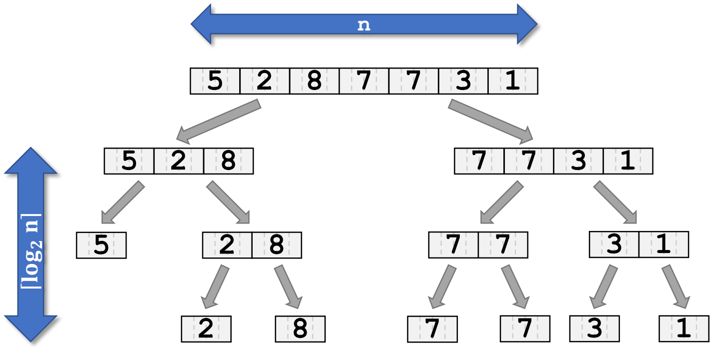
Runtime

(Intuition)

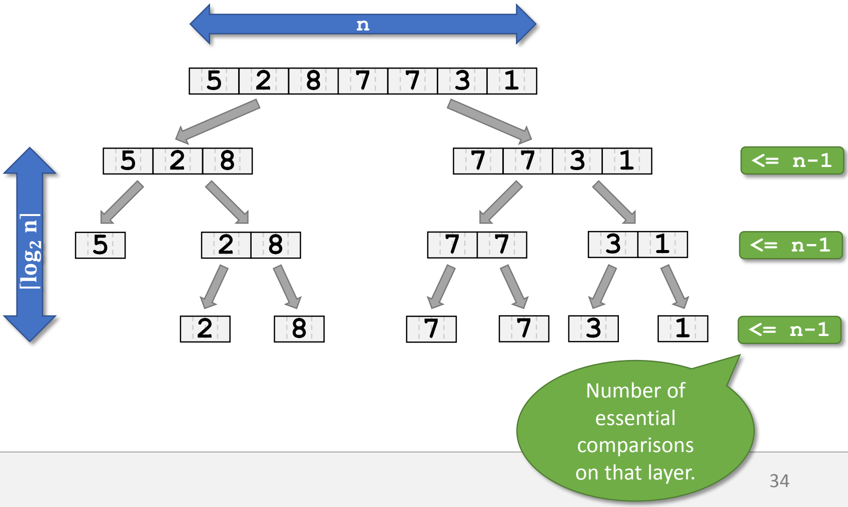
Alternative Proof



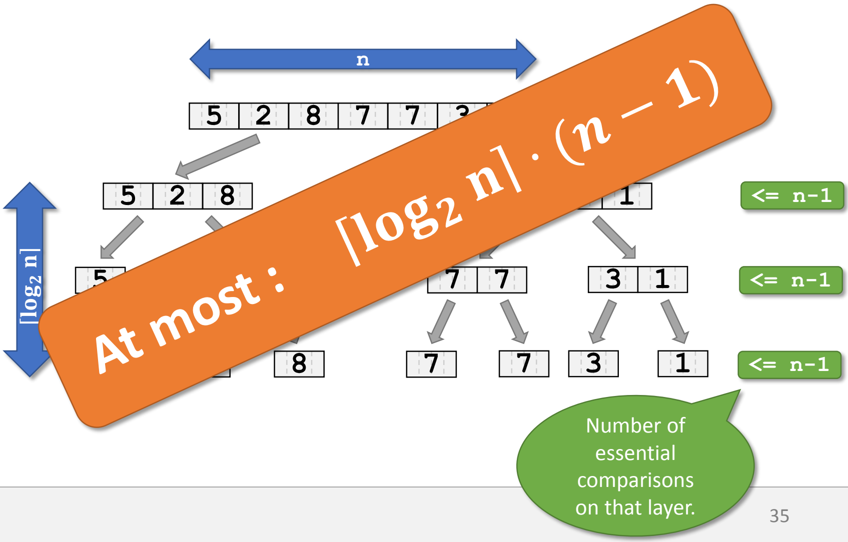
Alternative Proof



Alternative Proof



Alternative Proof



n
5 2 8 7 7 3
5 2 8
1
$\lceil \log_2 n \rceil$
5
7 7
3 1
8
7 7 3 1
<= n-1
<= n-1
<= n-1
At most : $\lceil \log_2 n \rceil \cdot (n-1)$
Number of essential comparisons on that layer.

Exercise "`our_list::merge_sorted`" (Difficult)

Exercise "`our_list::merge_sorted`" (Difficult)

- Open "`our_list::merge_sorted`" on **code** expert

Exercise "`our_list::merge_sorted`" (Difficult)

- Open "`our_list::merge_sorted`" on **code** expert
- Think about how you would approach the problem with pen and paper

Exercise "`our_list::merge_sorted`" (Difficult)

- Open "`our_list::merge_sorted`" on **code** expert
- Think about how you would approach the problem with pen and paper
- Implement a solution (optionally in groups)

Exercise "`our_list::merge_sorted`" (Solution)

Exercise "`our_list::merge_sorted`" (Solution)

See **code** expert

Questions?

6. Outro

General Questions?

See you next time

Have a nice week!