Patterns for Efficient Software

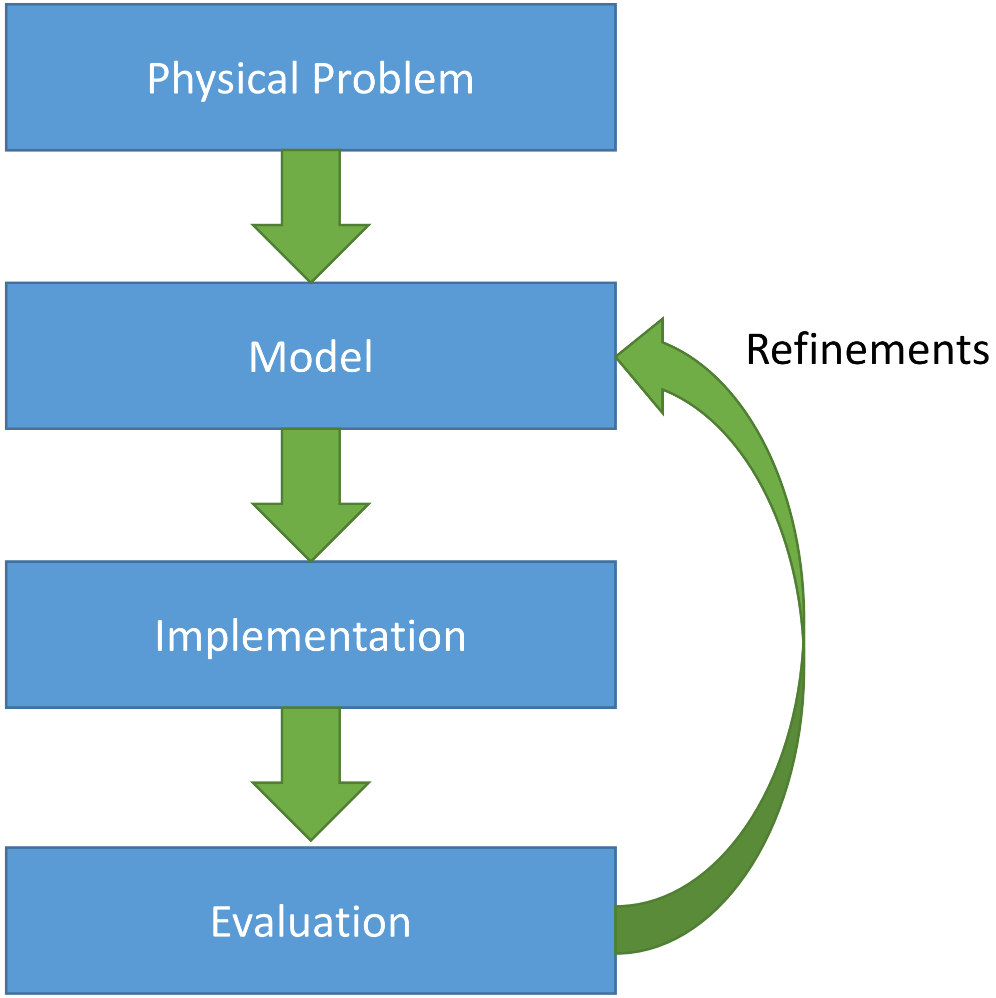
Richard Berger

richard.berger@jku.at

Johannes Kepler University Linz

What writing scientific codes looks like…

- Many of us write programs to solve specific problems in science
- We create and use models to describe our problems
- These models are implemented as code and produce results
- Evaluating these results allows us to validate our models and improve them

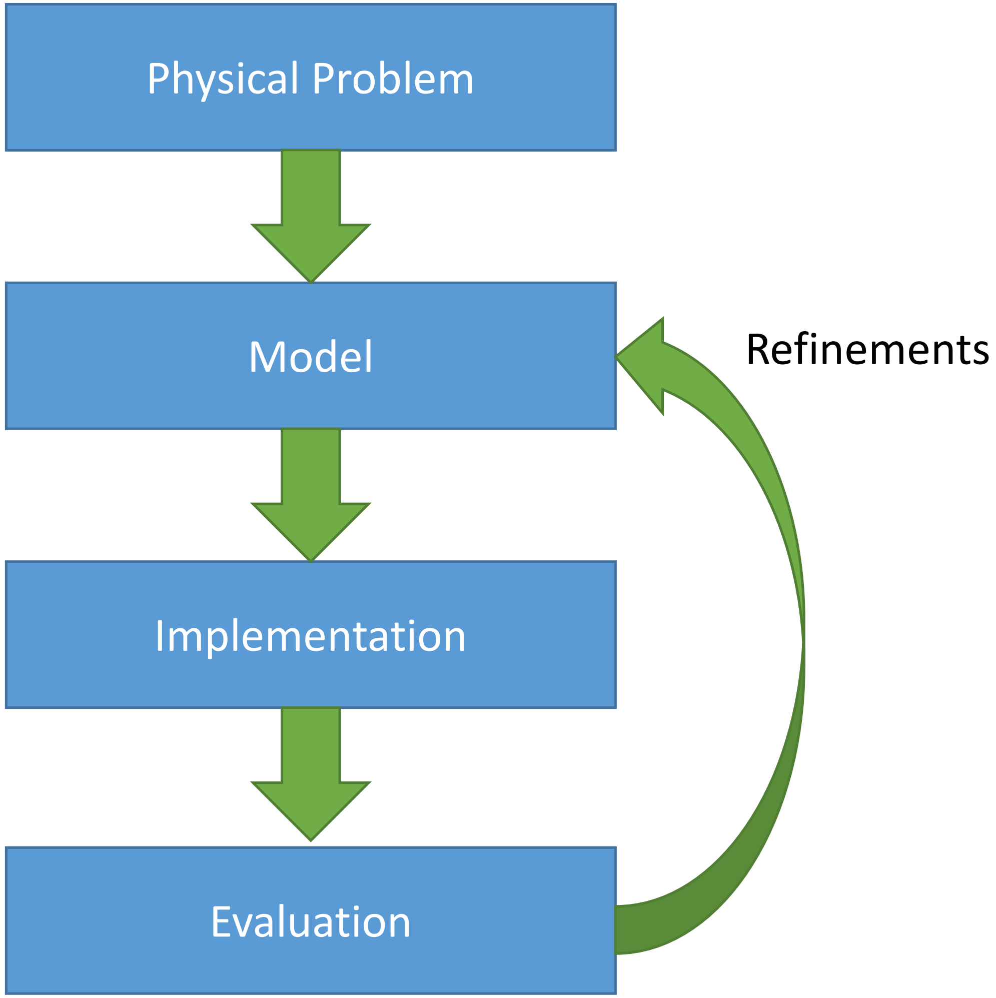


What writing scientific codes looks like…

- Performance, Maintainability, Portability of those programs is not our main concern
- What matters is that the results we compute are **correct**!

➢ First working solution often ends up in production code

➢ Sub-optimal solutions are chosen, which only get noticed later in the process (e.g. when working on bigger data sets)



There is more than one way to Rome…

- There is usually more than one way of solving your problem
- It’s very likely that someone other than you has already solved at least parts of your problem (and in a better way than you could have ever imagined)
- So the key approach to this is:
  - Know what’s out there
  - Figure out a way to choose the best solution

What this talk is about

- Choosing algorithms
  - How to compare algorithms
  - Example: Sorting

- Choosing data structures
  - Performance Characteristics
  - Types

also covered:
Generic Programming,
Examples: C++ STL

Algorithms

"Informally, an ***algorithm*** is any well-defined computational procedure that takes some value, or set of values, as ***input*** and produces some value, or set of values, as ***output***.

We can also view an algorithm as a tool for solving a well-specified ***computational problem***. The statement of the problem specifies in general terms the desired input/output relationship. The algorithm describes a specific computational procedure for achieving that input/output relationship."

from *Introduction to Algorithms* (3. Edition), by Thomas H. Cormen et. al.

Comparing Algorithms

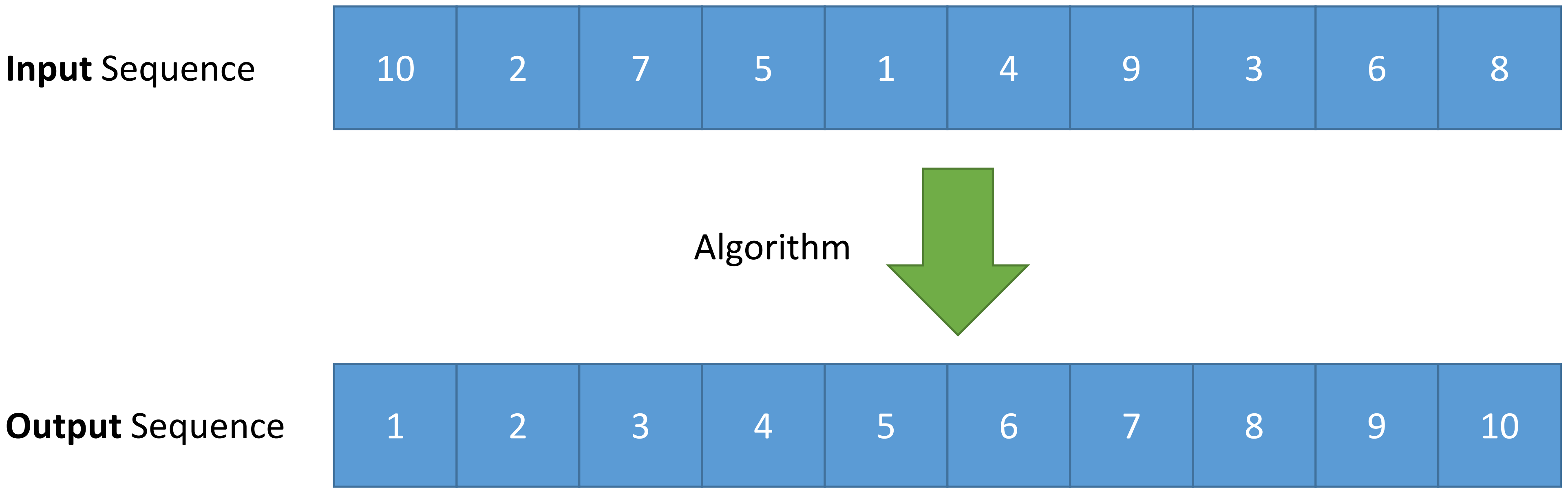
Theoretically:

- by **runtime complexity**
- by **memory usage**

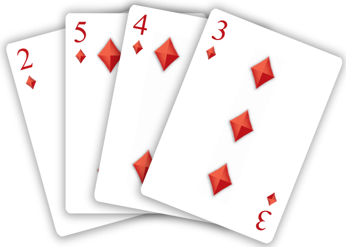
Practically:

- by **measurement (profiling)**

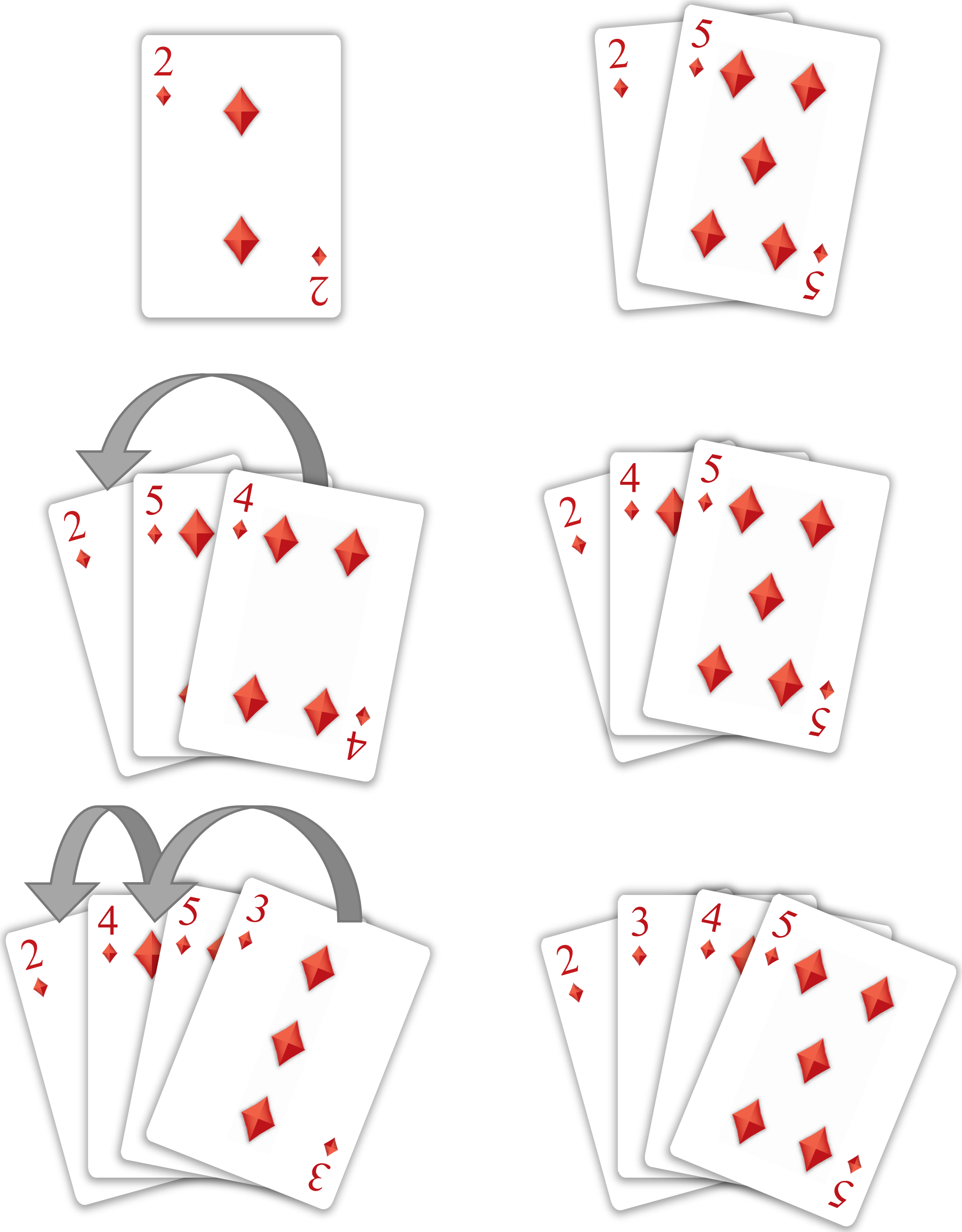
Sorting


Input Sequence
10 2 7 5 1 4 9 3 6 8
Algorithm
Output Sequence
1 2 3 4 5 6 7 8 9 10

Insertion Sort

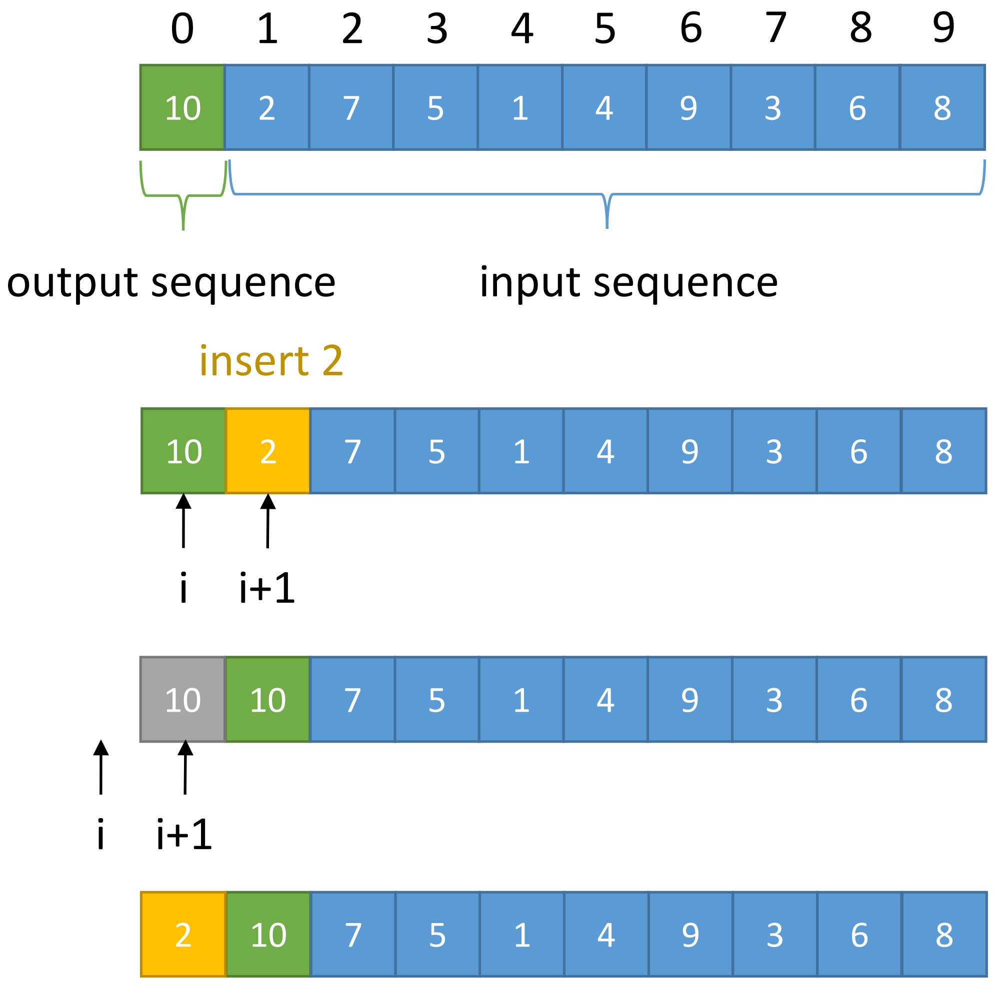


- Start out with an unordered input sequence of elements and an empty output sequence
- You then remove one element from the input sequence and insert it into the output sequence from right to left at the correct location
- Repeat until the input sequence is empty



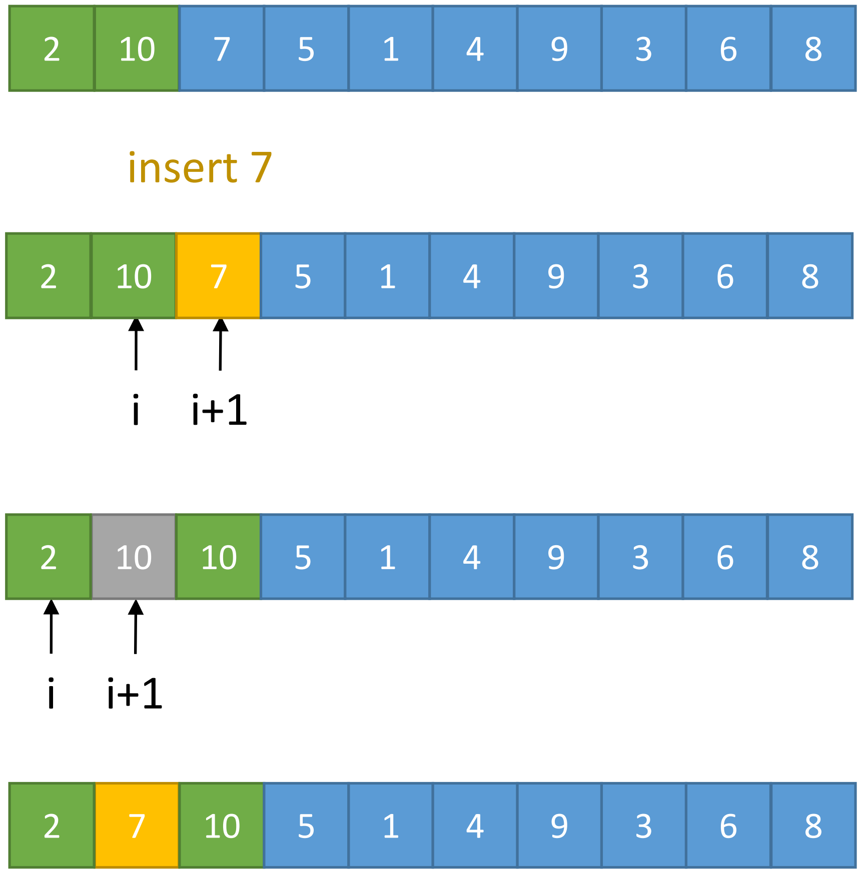
Insertion Sort – Implementation (C++)

```cpp
void insertion_sort(std::vector<int> & v) {
    for(int j = 1; j < v.size(); j++) {
        int key = v[j];
        int i = j - 1;
        while(i >= 0 && v[i] > key) {
            v[i+1] = v[i];
            i--;
        }
        v[i+1] = key;
    }
}
```



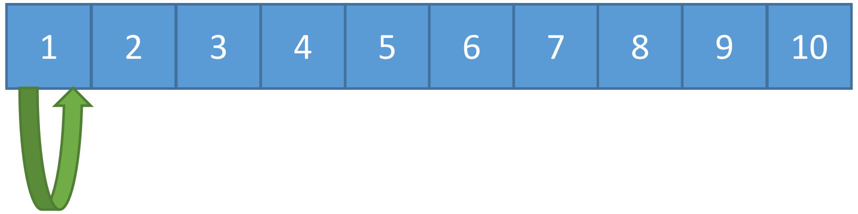
Insertion Sort – Implementation (C++)

```
void insertion_sort(std::vector<int> & v) {
    for(int j = 1; j < v.size(); j++) {
        int key = v[j];
        int i = j - 1;
        while(i >= 0 && v[i] > key) {
            v[i+1] = v[i];
            i--;
        }
        v[i+1] = key;
    }
}
```

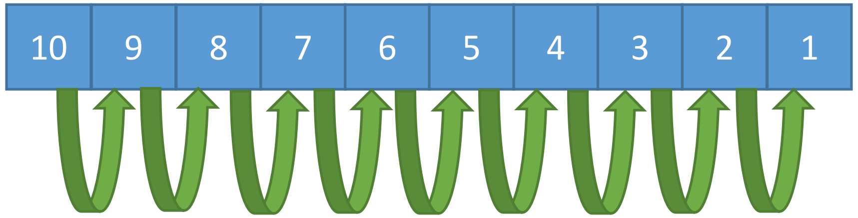


Insertion Sort

Best case: Sequence already in sorted order



Worst case: Sequence in reverse order



Insertion Sort

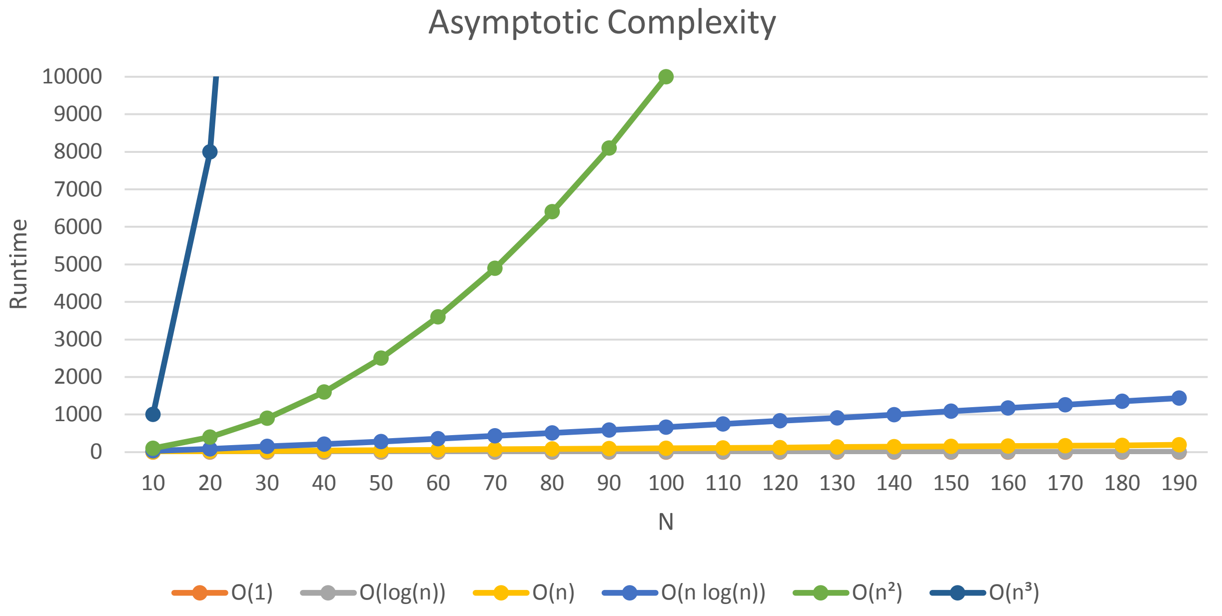
```
void insertion_sort(std::vector<int> & v) {
    for(int j = 1; j < v.size(); j++) {
        int key = v[j];
        int i = j - 1;

        while(i >= 0 && v[i] > key) {

            v[i+1] = v[i];

            i--;
        }
        v[i+1] = key;
    }
}
```

times	Best case ($t_j = 1$)	Worst case ($t_j = j$)
n	**n**	**n**
n-1	**n-1**	**n-1**
n-1	**n-1**	**n-1**
$\sum_{j=1}^{n-1} t_j$	**n-1**	$\frac{n(n+1)}{2} - 1$
$\sum_{j=1}^{n-1} (t_j - 1)$	**0**	$\frac{n(n-1)}{2}$
$\sum_{j=1}^{n-1} (t_j - 1)$	**0**	$\frac{n(n-1)}{2}$
n-1	**n-1**	**n-1**
	O(n)	**O(n²)**

Asymptotic Complexity & O-Notation

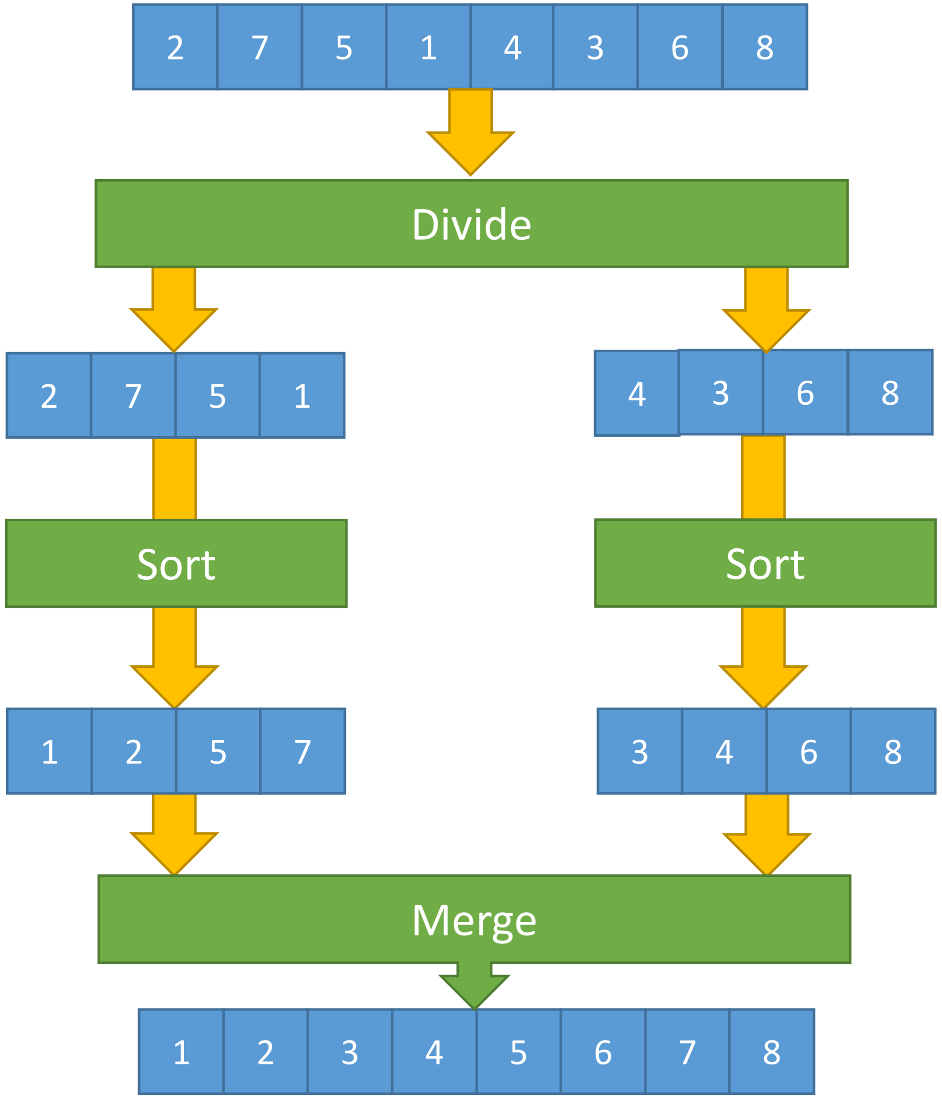
- $O(1)$ … constant
- $O(\log n)$ … logarithmic
- $O(n)$ … linear
- $O(n \log n)$ … quasi-linear
- $O(n^2)$ … quadratic
- $O(n^3)$ … cubic
- $O(c^n)$… exponential

n	O(1)	O(log(n))	O(n)	O(n log(n))	O(n²)	O(n³)
10	const	3	10	33	100	1000
1000	const	10	1000	9966	1E+06	1E+09
100000	const	17	100000	1660964	1E+10	1E+15
1000000	const	20	1000000	19931569	1E+12	1E+18



Merge Sort

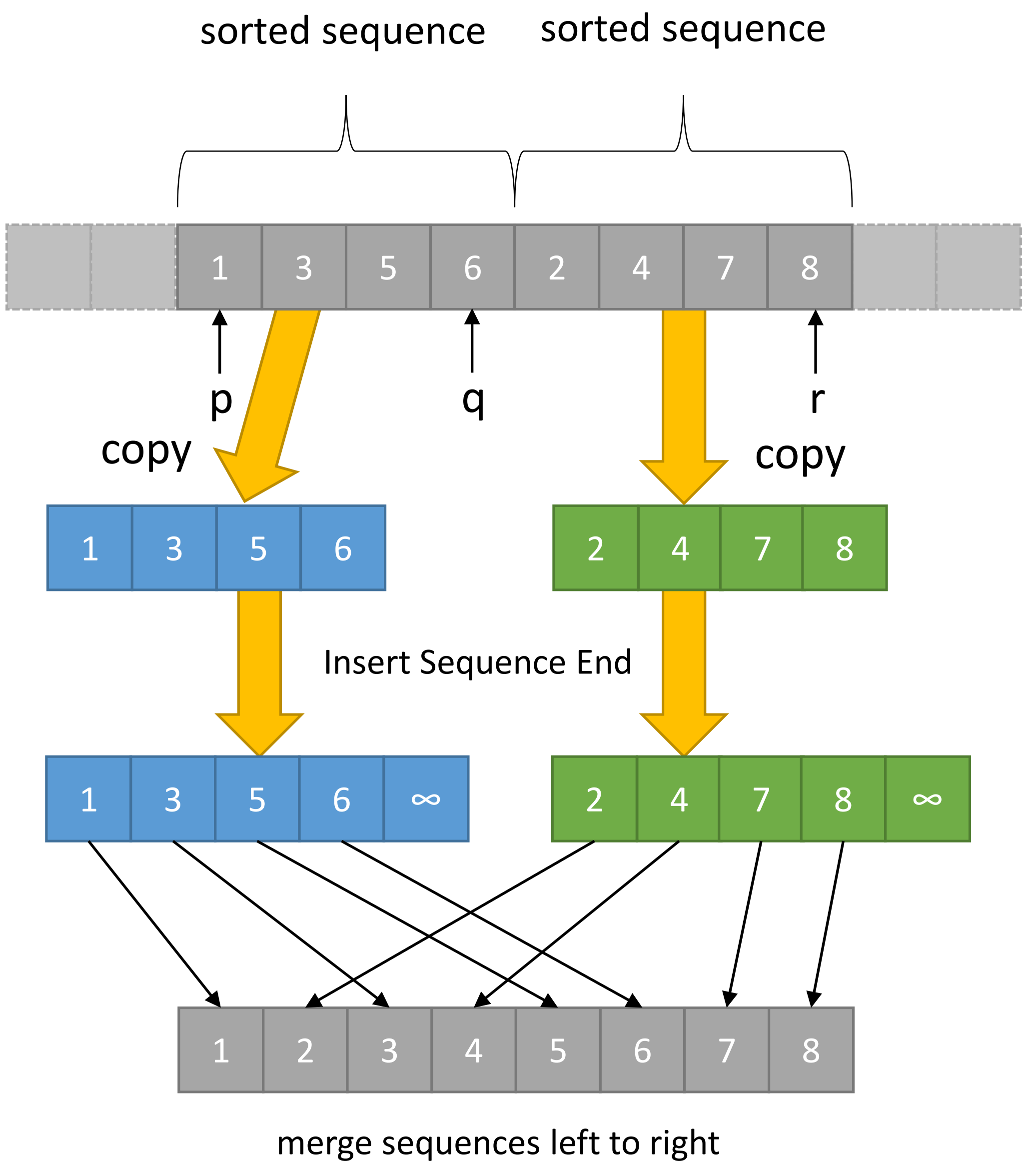
- Divide-And-Conquer Approach
- Subdivide sequence into smaller subsequences
- Sort each subsequence
- Merge ordered subsequences to final sequence



Merge Sort

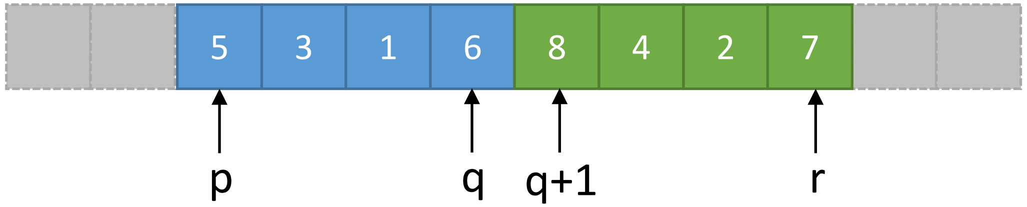
```
void merge(vector<int> & v, int p, int q, int r) {
    vector<int> left(v.begin()+p, v.begin()+q+1);
    vector<int> right(v.begin()+q+1, v.begin()+r+1);

    left.push_back(/* infinity */)
    right.push_back(/* infinity */);
    int i = 0;
    int j = 0;

    for(int k = p; k <= r; k++) {
        if(left[i] <= right[j])
            v[k] = left[i++];
        else
            v[k] = right[j++];
    }
}
```

O(n)



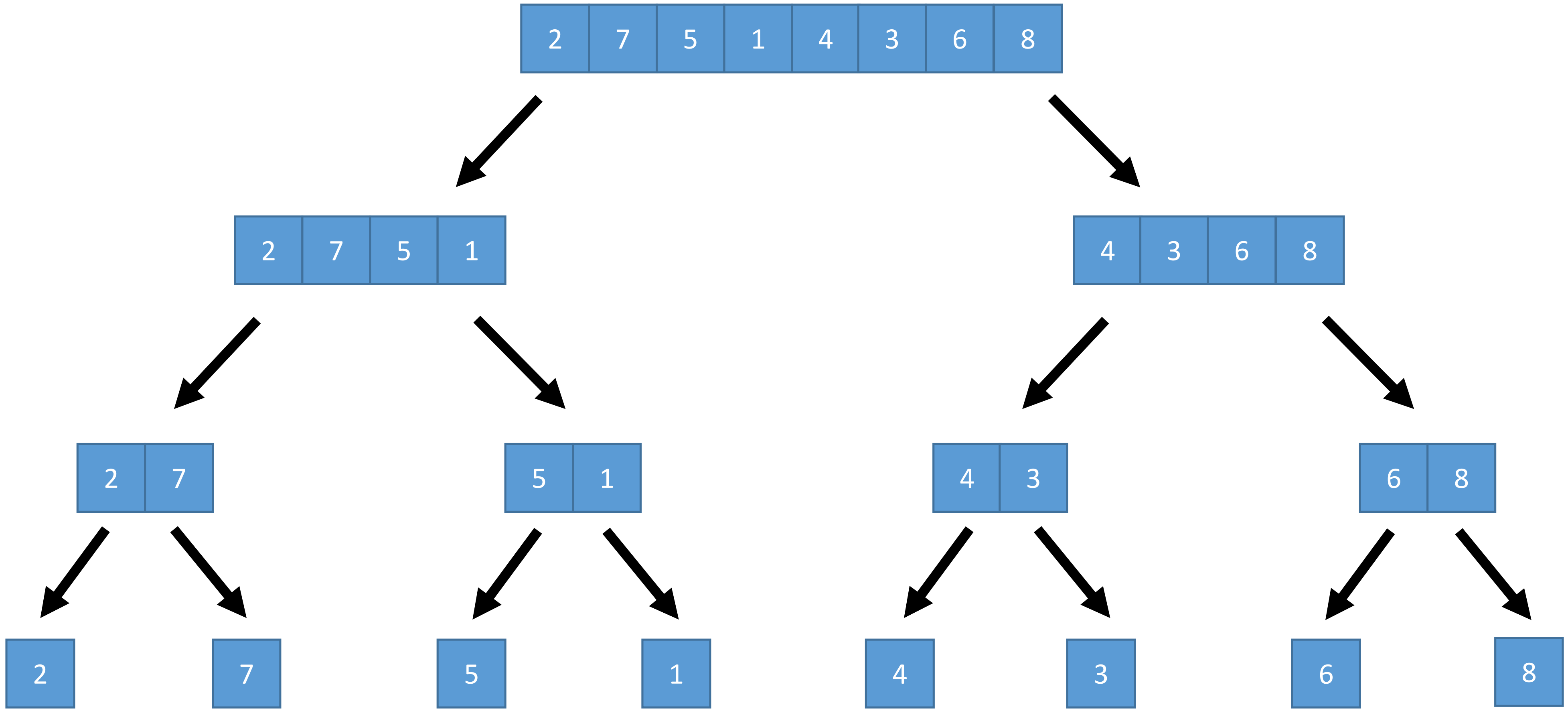
Merge Sort

```
void merge_sort(std::vector<int> & v, int p, int r) {
    if(p < r) {
        int q = (p+r)/2;
        merge_sort(v, p, q);
        merge_sort(v, q+1, r);
        merge(v, p, q, r);
    }
}
```

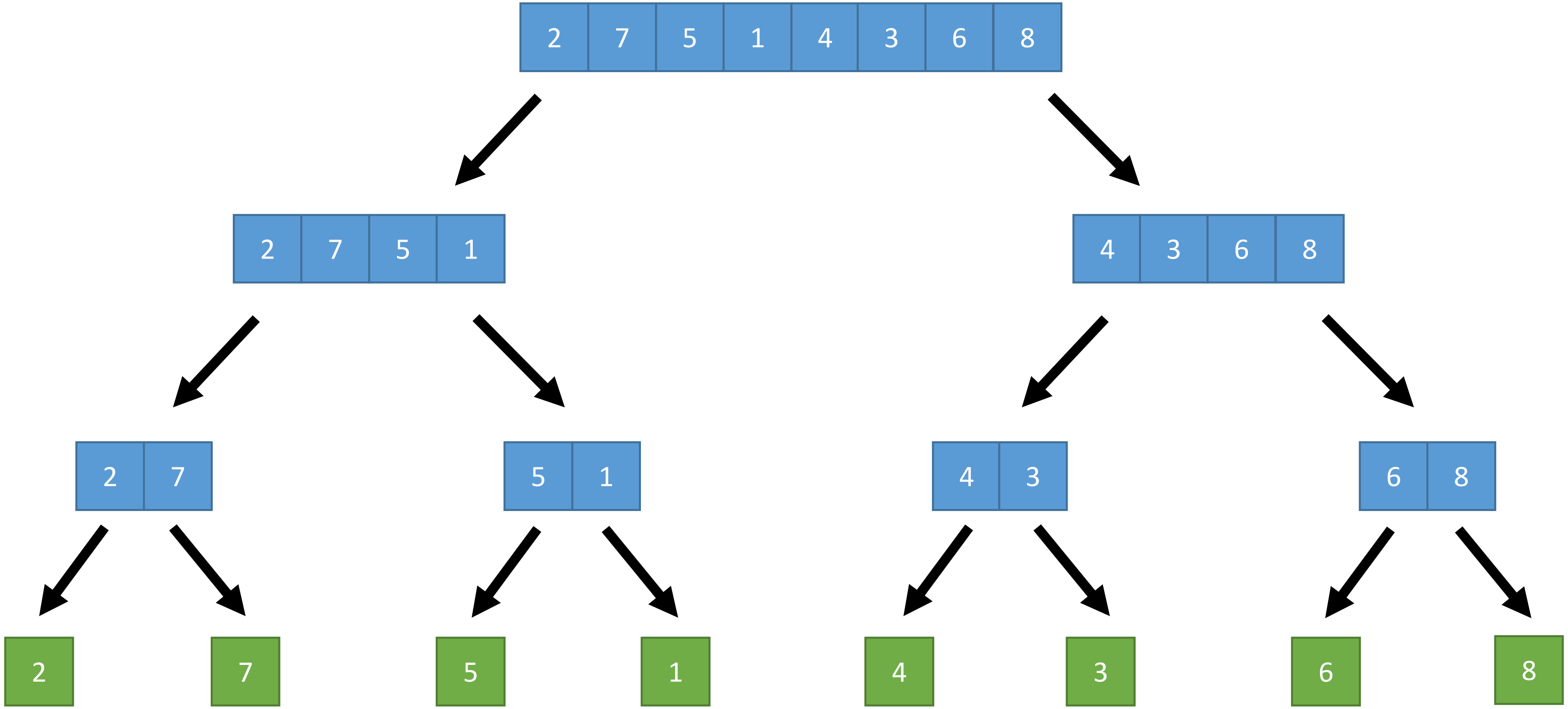



```
void merge_sort(std::vector<int> v) {
    merge_sort(v, 0, v.size());
}
```

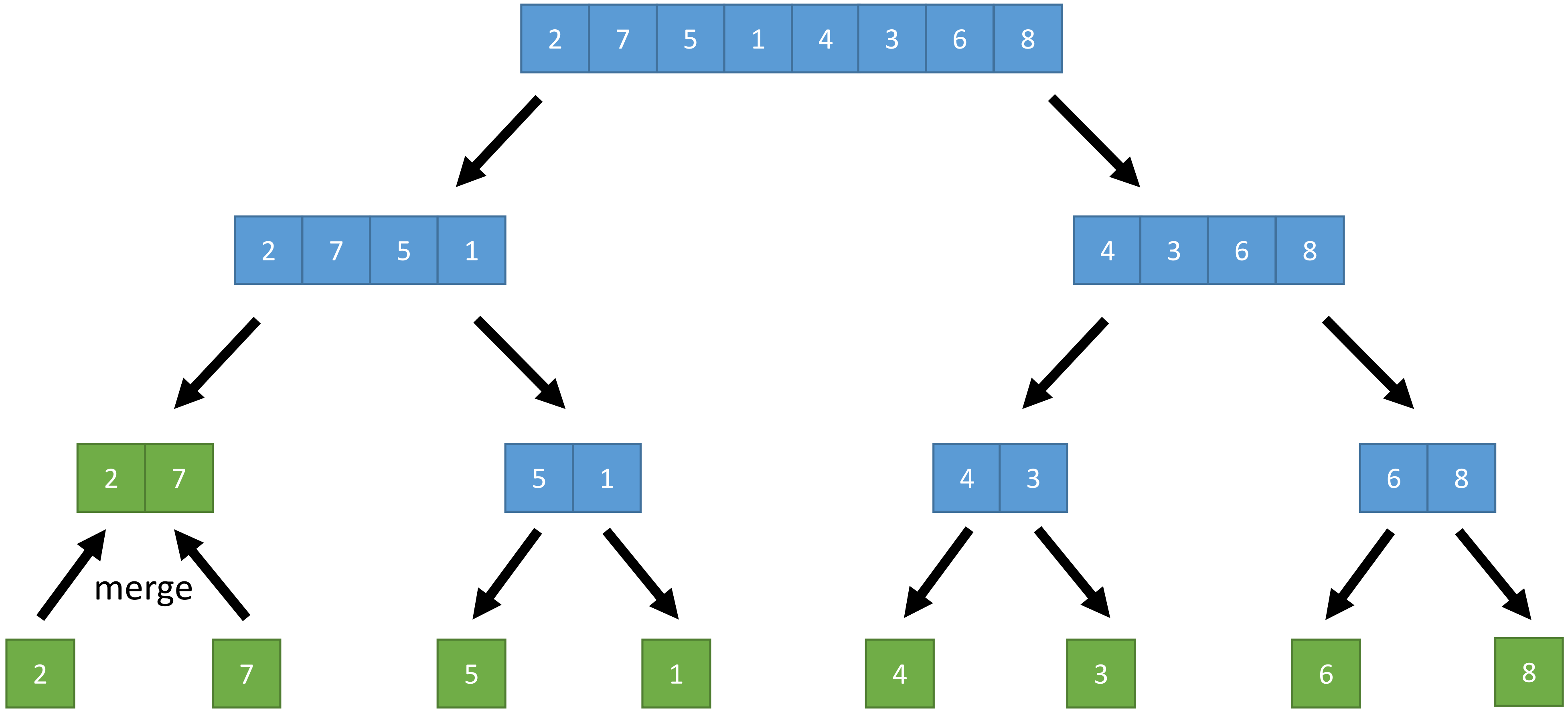
Merge Sort



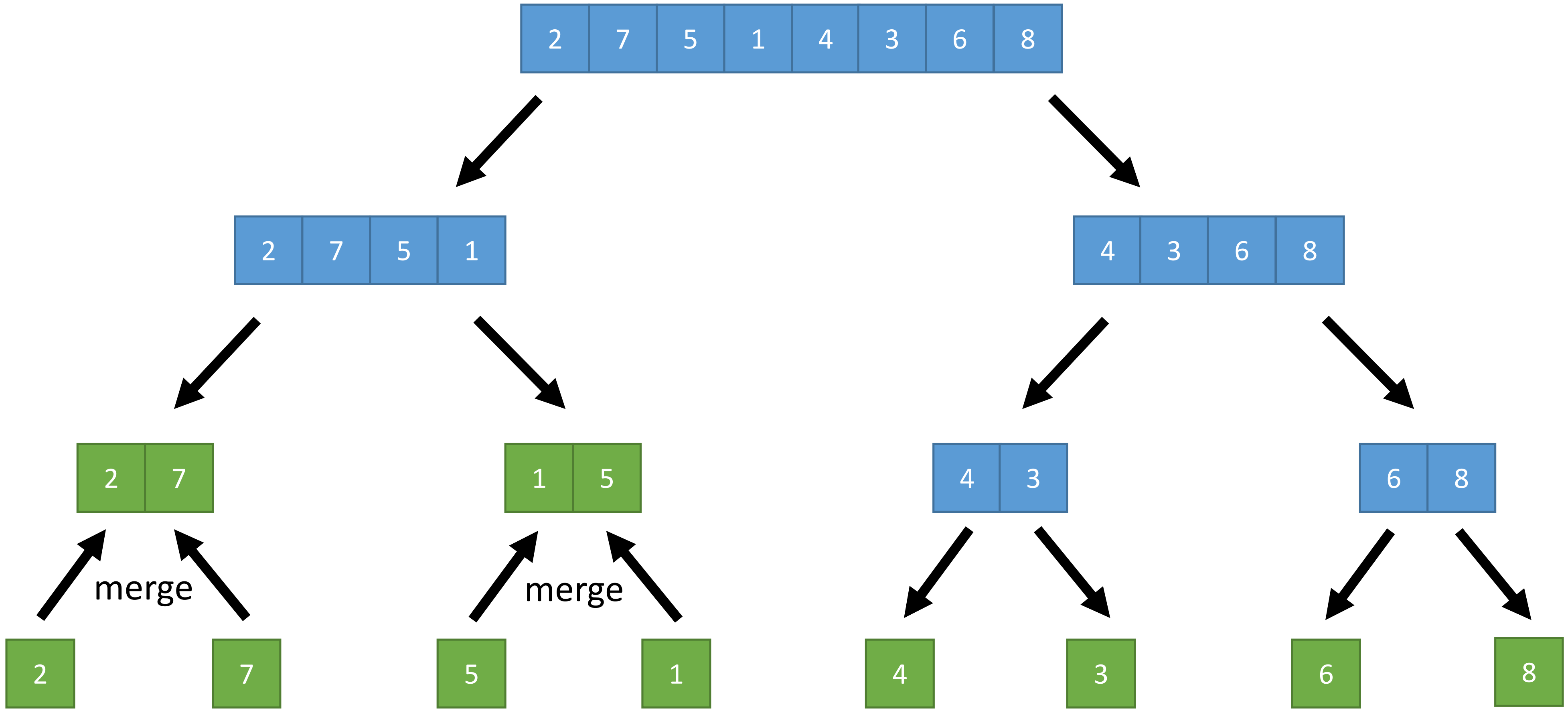
Merge Sort



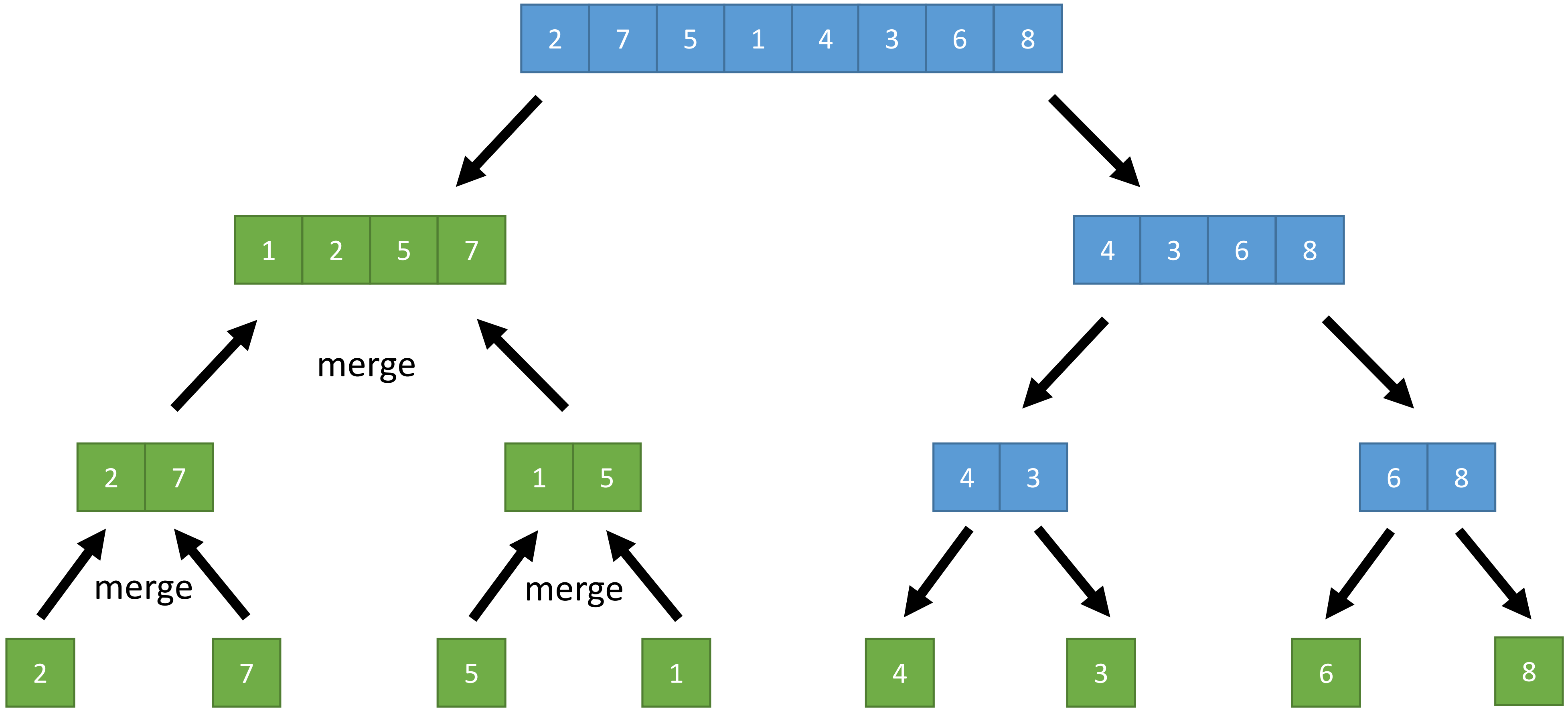
Merge Sort



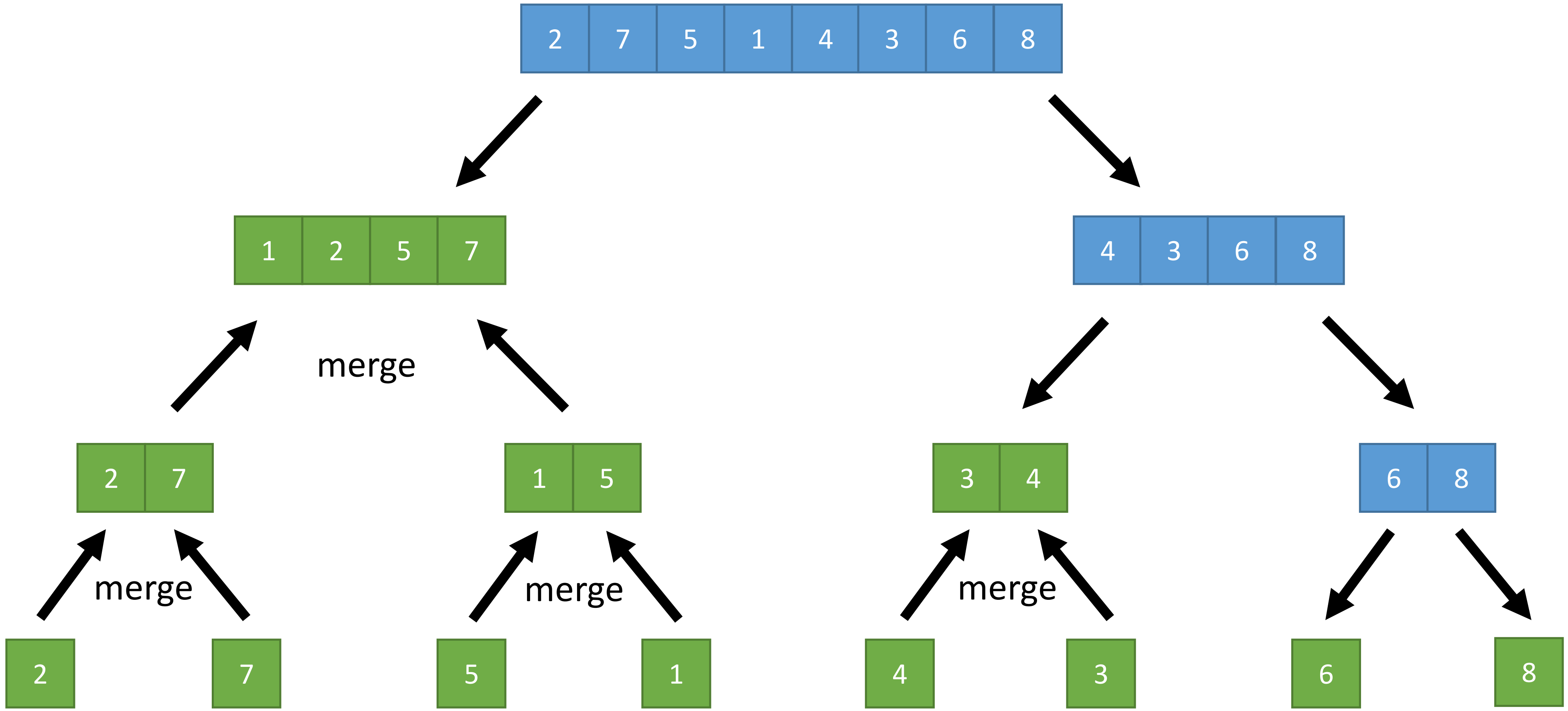
Merge Sort



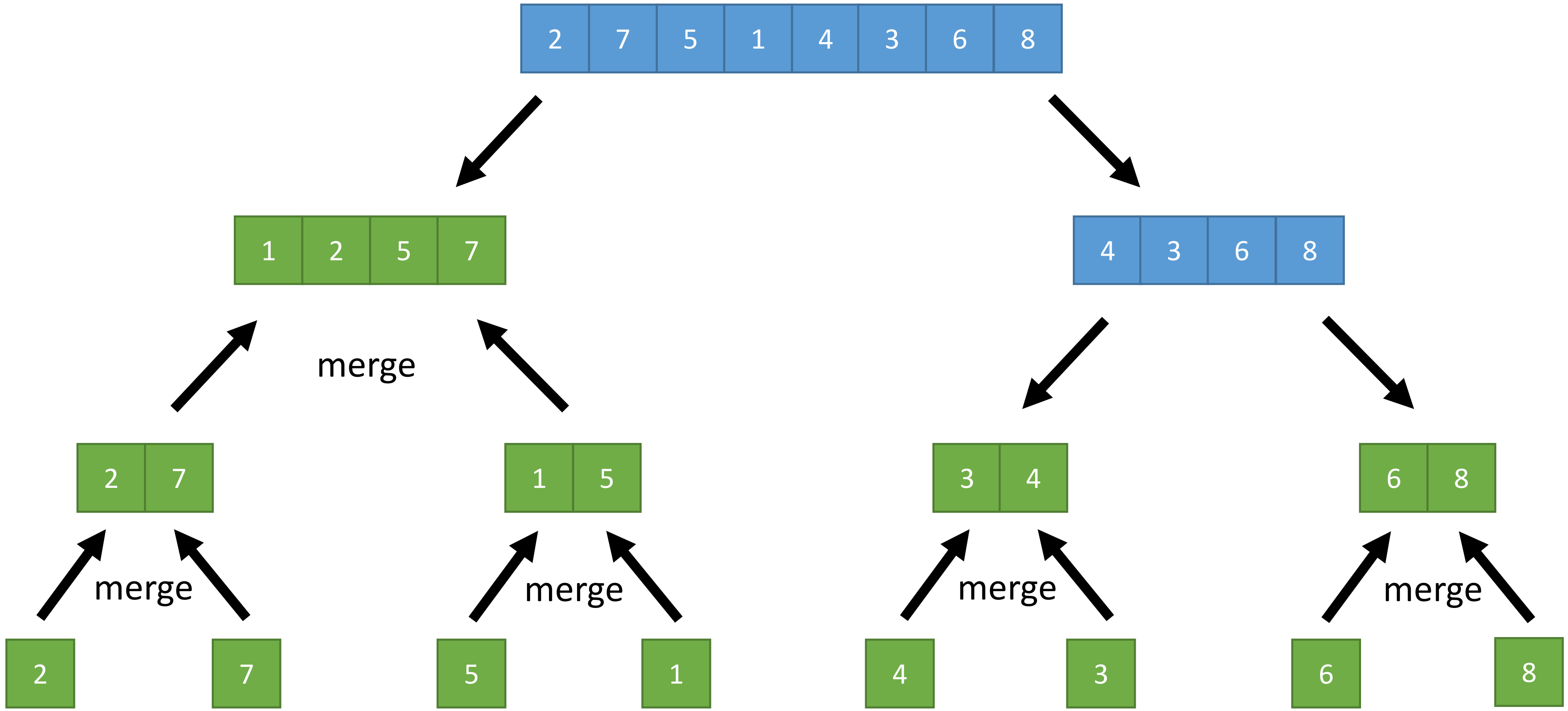
Merge Sort



Merge Sort

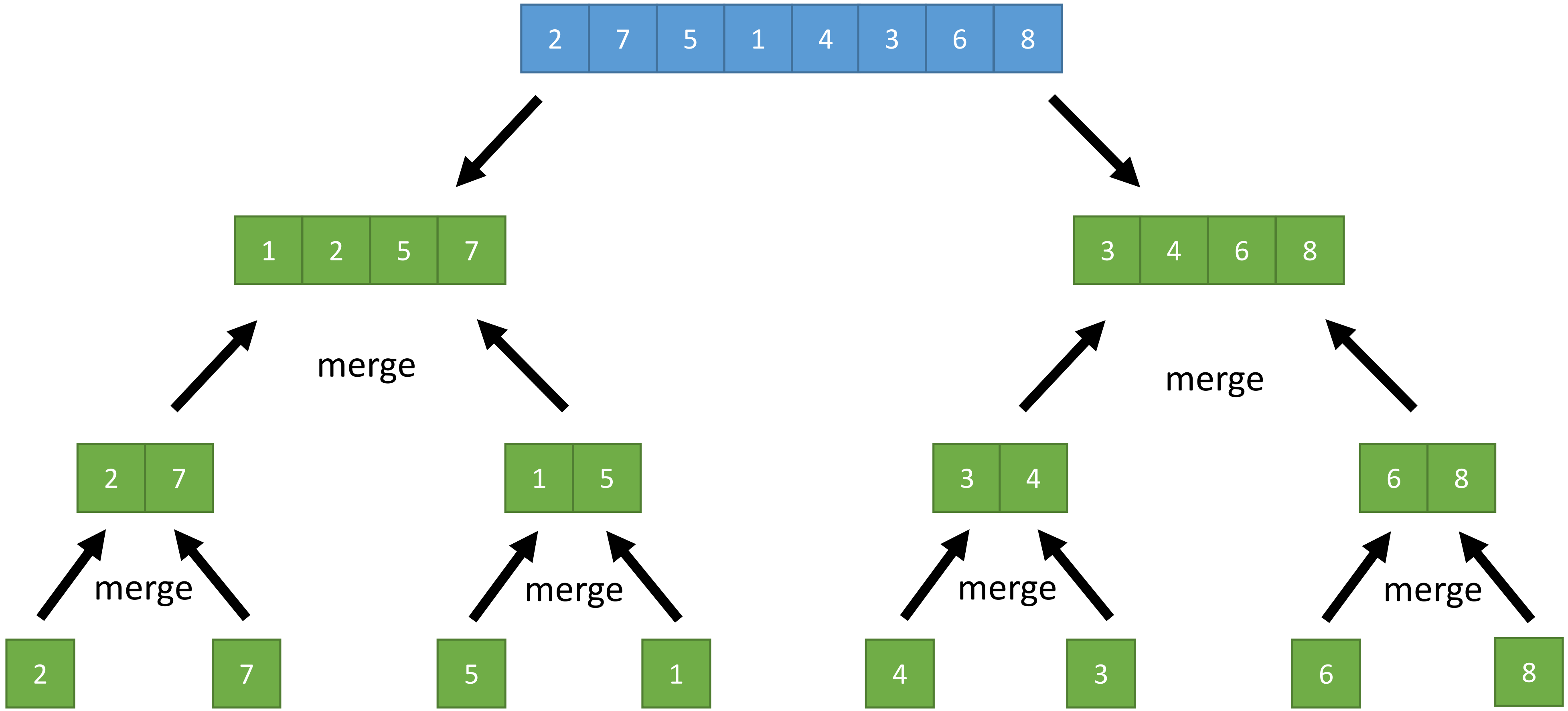

2 7 5 1 4 3 6 8
1 2 5 7
4 3 6 8
merge
2 7
1 5
3 4
6 8
merge
merge
merge
2
7
5
1
4
3
6
8

Merge Sort



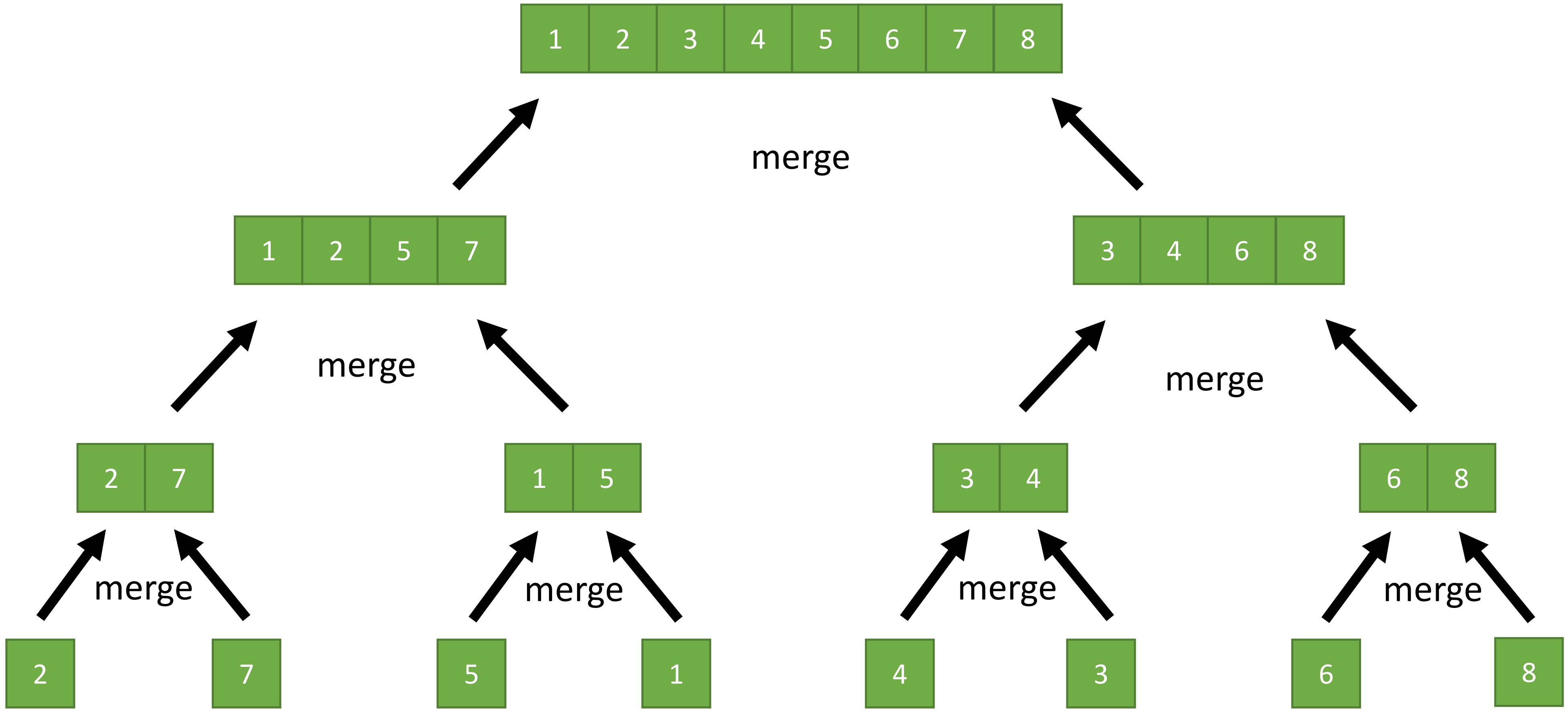
2 7 5 1 4 3 6 8
1 2 5 7
4 3 6 8
merge
2 7
1 5
3 4
6 8
merge
merge
merge
merge
2
7
5
1
4
3
6
8

Merge Sort



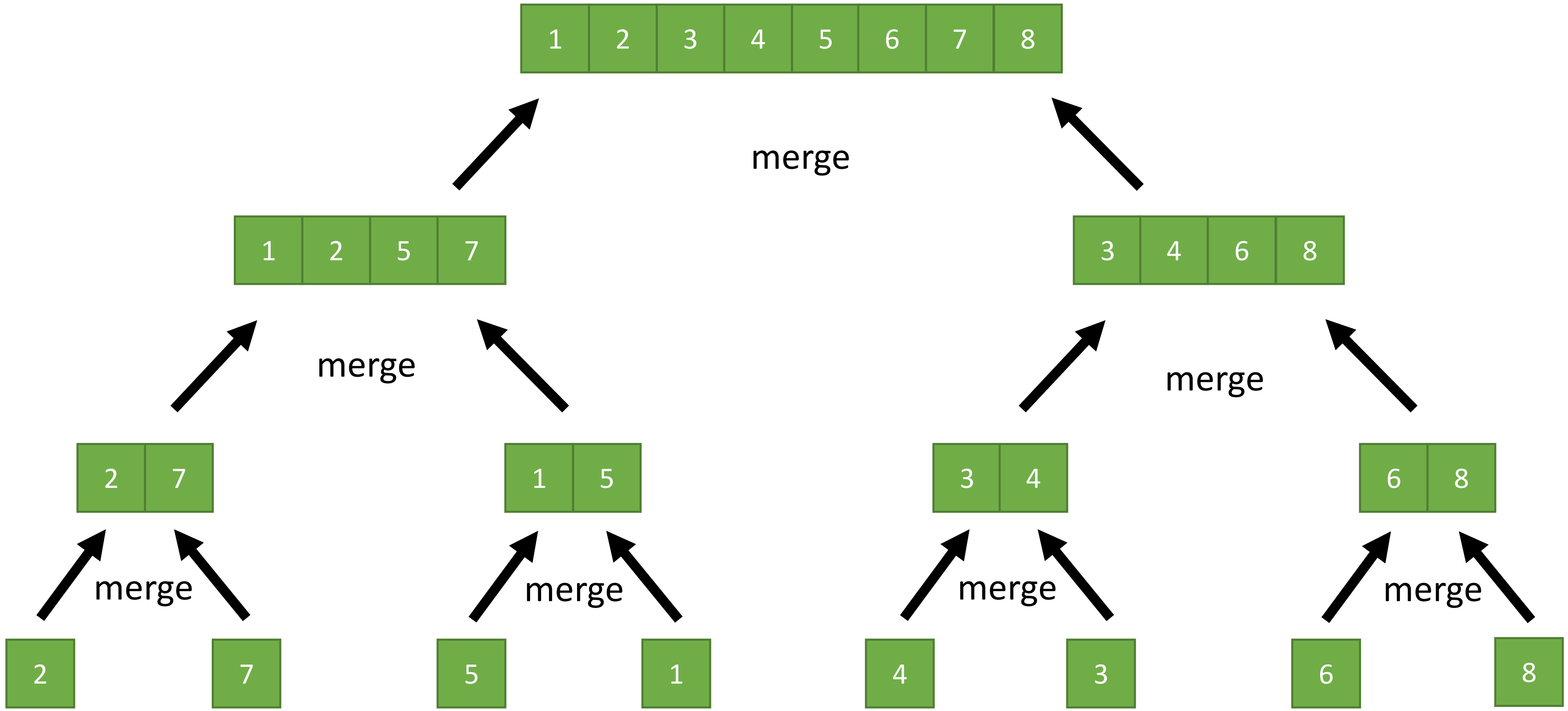
2 7 5 1 4 3 6 8
1 2 5 7
3 4 6 8
merge
merge
2 7
1 5
3 4
6 8
merge
merge
merge
merge
2
7
5
1
4
3
6
8

Merge Sort


1
2
3
4
5
6
7
8
merge
1
2
5
7
3
4
6
8
merge
merge
2
7
1
5
3
4
6
8
merge
merge
merge
merge
2
7
5
1
4
3
6
8

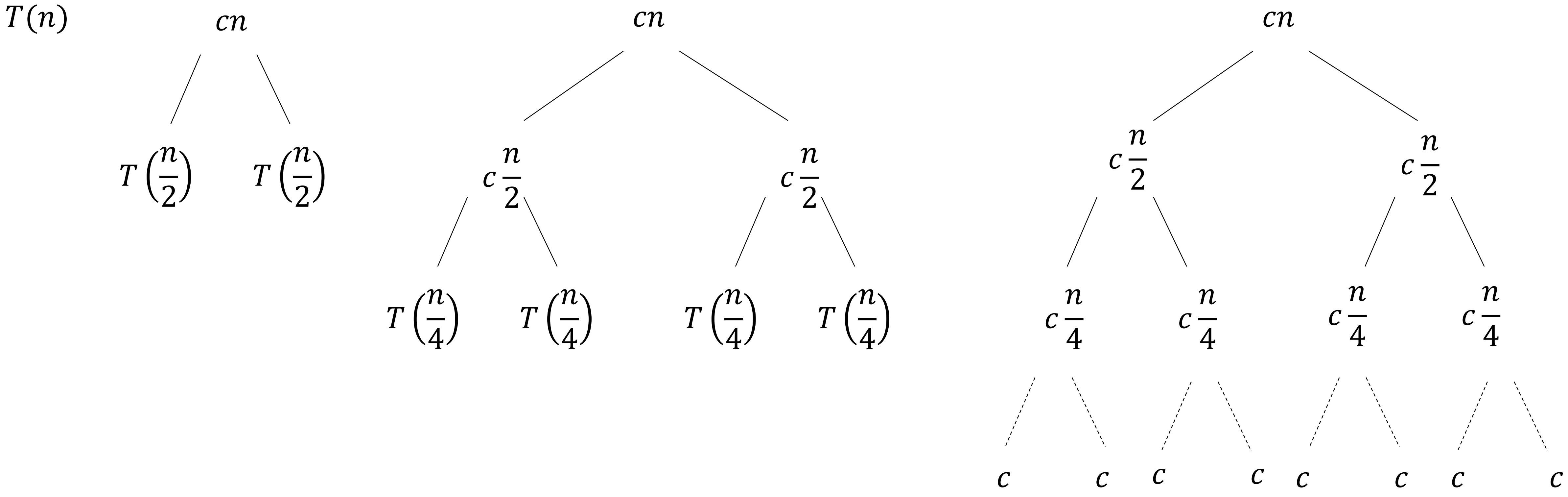
Merge Sort

$$T(n) = \begin{cases} c & if\ n = 1 \\ 2T\left(\frac{n}{2}\right) + cn & if\ n > 1 \end{cases}$$



Merge Sort – Recursive Tree

$$T(n) = \begin{cases} c & if\ n = 1 \\ 2T\left(\frac{n}{2}\right) + cn & if\ n > 1 \end{cases}$$

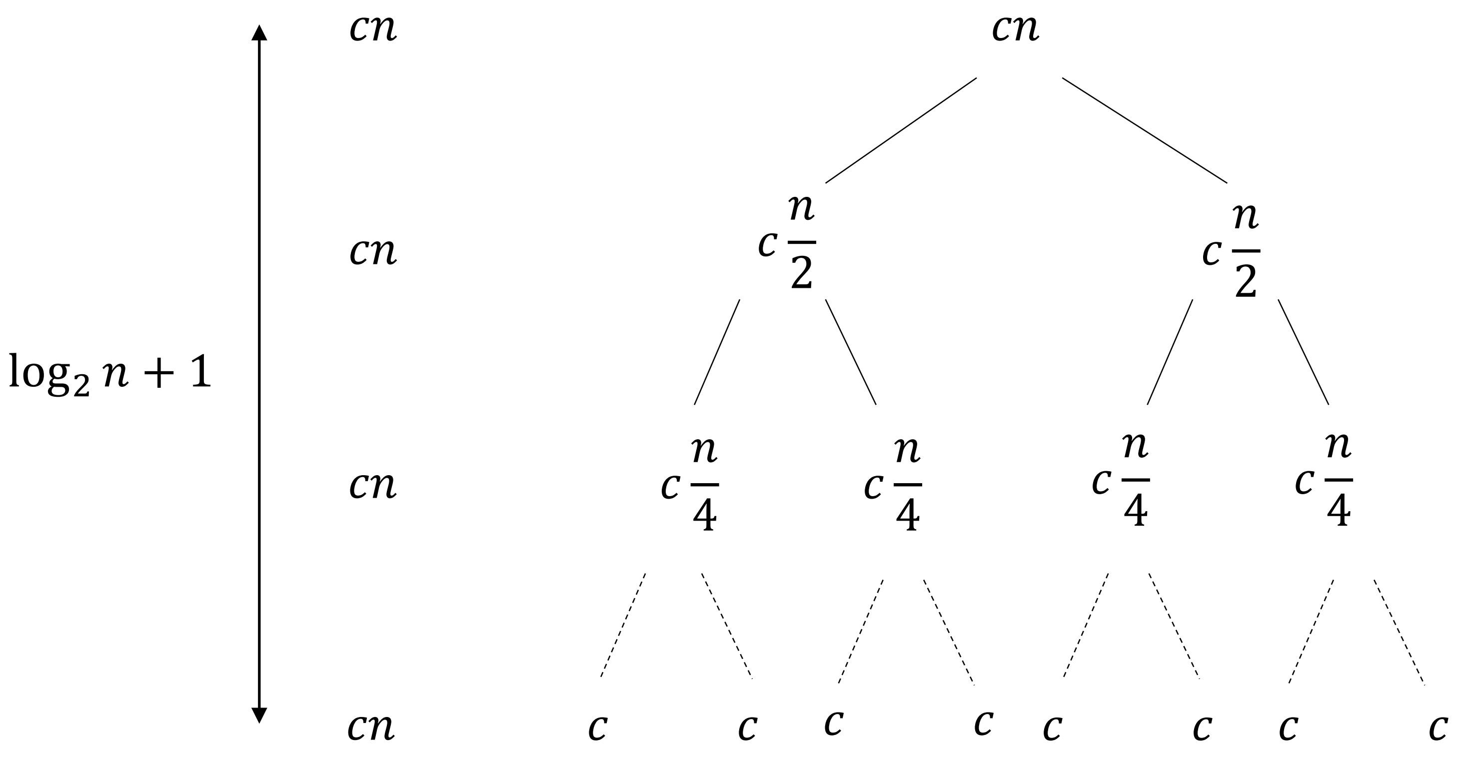


Merge Sort - Complexity

$$T(n) = \begin{cases} c & if\ n = 1 \\ 2T\left(\frac{n}{2}\right) + cn & if\ n > 1 \end{cases}$$

$\mathrm{T(n)} = \mathrm{cn}(\log_2 n + 1)$

O(n log n)



Comparing Algorithms

Theoretically:

- by **runtime complexity**
  - Insertion Sort $O(n^2)$ slower than Merge Sort $O(n \log n)$
- by **memory usage**
  - Insertion Sort can be implemented as in-place algorithm (work on same memory block)
  - Merge Sort creates copies of sub-sequences → 2X maximum memory usage

Practically:

- by **measurement (profiling)**

Sorting

- Quick Sort:
  - Worst case: $O(n^2)$
  - Best case: O(n log n)
  - Average case: O(*n* log *n*)
- Heap Sort:
  - Worst case: O(n log n)
  - Best case: O(n log n)
  - Average case: O(n log n)

Visualization of Sorting

- 15 Sorting Algorithms in 6 Minutes
  http://youtu.be/kPRA0W1kECg

Keep this in mind when optimizing

Finding a **better** algorithm > Optimizing an algorithm

$O(n)$ beats $O(n^2)$
$O(1)$ beats $O(n)$
…

Generic Programming

- Defining functions and classes which work with many different types

C++: strict type system

```
void sort(int* ar, int length);
void sort(float* ar, int length);
void sort(double* ar, int length);
```

C++ templates:
allow you to write generalized algorithms that can work with any type

```
template<typename T>
void sort(T* ar, int length);

template<typename T>
class SomeClass {
   T memberVariable;
}
```

Python: duck typing

```
def sort(ar, length):
    …
```

Python doesn't care what types come in as parameters. It will try to use them. Type checking is done at runtime.

"If it quacks like a duck, looks like a duck, then it is a duck."

Generic Algorithms

```
void insertion_sort(std::vector<int> & v) {
    for(int j = 1; j < v.size(); j++) {
        int key = v[j];
        int i = j - 1;
        while(i >= 0 && v[i] > key) {
            v[i+1] = v[i];
            i--;
        }
        v[i+1] = key;
    }
}
```

```
template<typename T>
void insertion_sort(std::vector<T> & v) {
    for(int j = 1; j < v.size(); j++) {
        T key = v[j];
        int i = j - 1;
        while(i >= 0 && v[i] > key) {
            v[i+1] = v[i];
            i--;
        }
        v[i+1] = key;
    }
}
```

Usage:

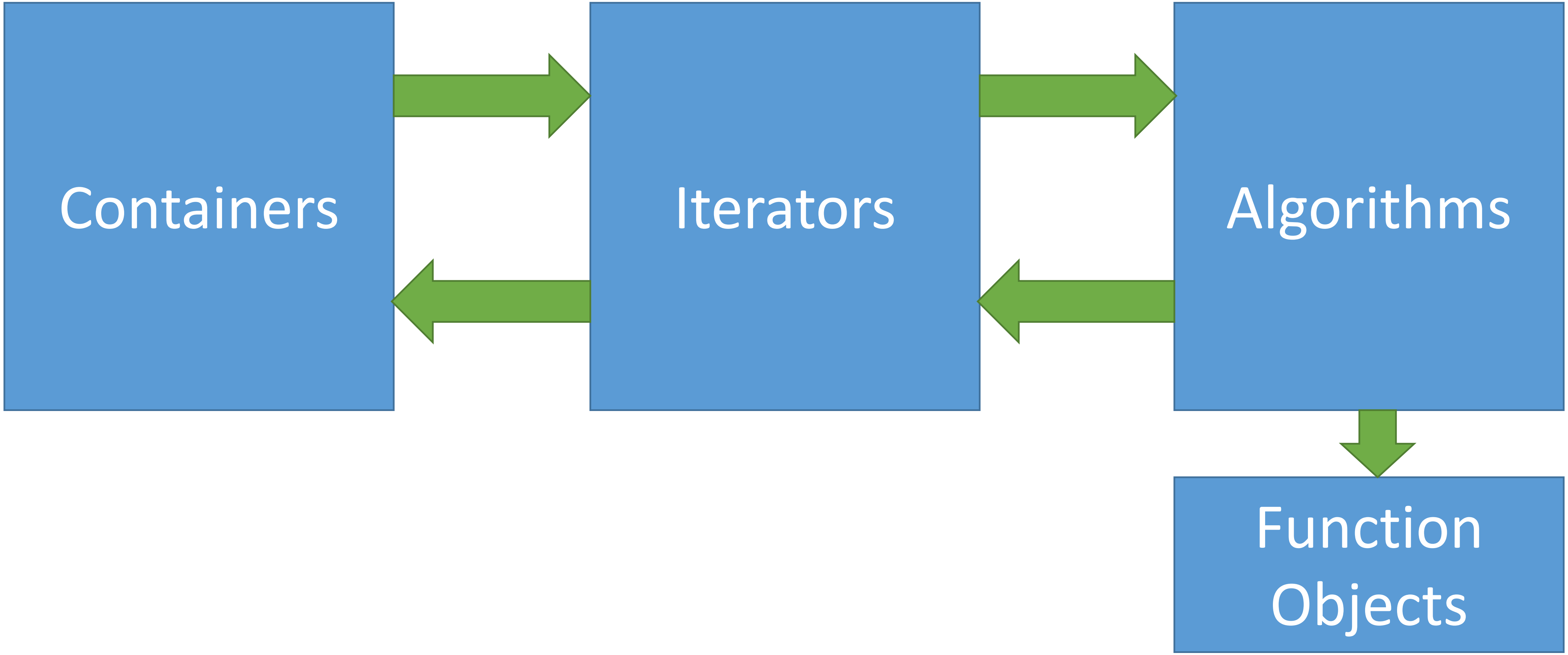
```
std::vector<int> i_list;
std::vector<double> fp_list;
insertion_sort(i_list);
insertion_sort(fp_list);
```

C++ Standard Template Library

- Common Algorithms and Data Structures
- Optimized for General Computing
- Uses C++ template mechanisms extensively
- No inhertance, no virtual calls

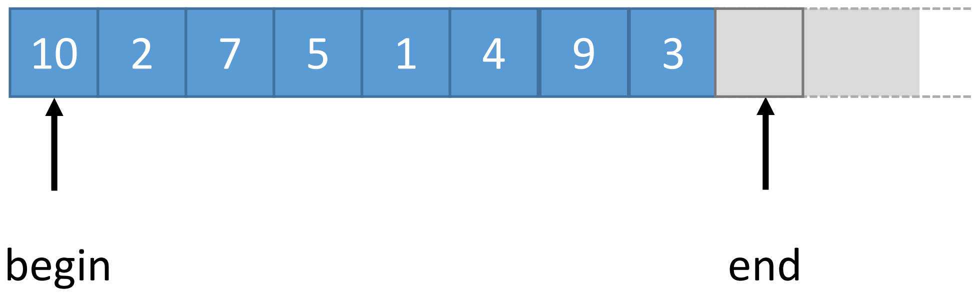
C++ Standard Template Library (STL)

Part of the C++ Standard Library



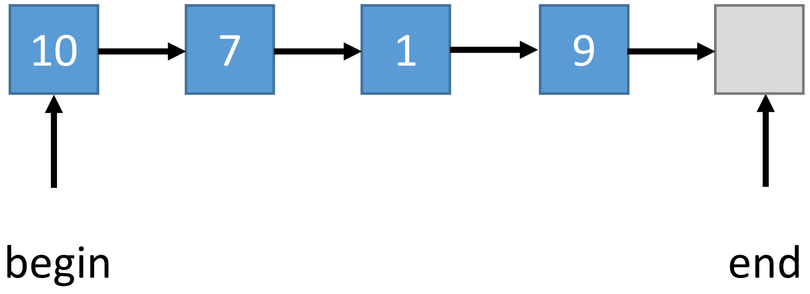
Iterators

„Generalized Pointers“




```
int myArray[100];

int * b = &myArray[0];
int * e = &myArray[0] + 100;


for(int i = 0; i < 100; ++i) {
    myArray[i]  = /* calculation */;
}


for(int * a = b; a != e; ++a) {
    *a  = /* calculation */;
}
```

```
vector<int> v(100, 0);


for(vector<int>::iterator it = v.begin(); it != v.end(); ++it)
    *it  = /* calculation */;
}
```



Function Objects

- Python
  - any function can be used as an object
  - lambda functions: anonymous functions (functions without a name)

```python
def less_than(a, b):
    return a < b

# store function in a variable
a_variable = less_than

# call function through variable
a_variable(a, b)
```

- C++
  - Something like C-callbacks, but it can store its state
  - Definition of a class which implements special operator() member function
  - C++11 lambda functions

```cpp
struct less_than {
    bool operator()(int a, int b) {
        return a < b;
    }
}

less_than a_object;
// use that object like a function
bool result = a_object(10, 20);
```

std::**sort** and std::**stable_sort**

stable_sort ensures the order of objects which are equal is not changed after sorting.

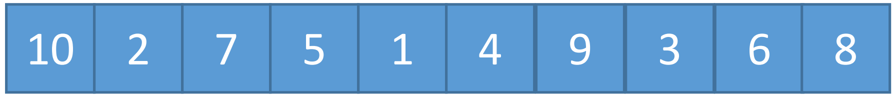
```
#include <vector>
#include <list>
#include <algorithm> // sort
#include <functional> // function objects

std::vector<int> myVector;

// sort in ascending order
std::sort(myVector.begin(), myVector.end());

// equivalent
std::sort(myVector.begin(), myVector.end(), std::less_than())

// descending order
std::sort(myVector.begin(), myVector.end(), std::greater_than())
```

Function Object

STL Algorithms

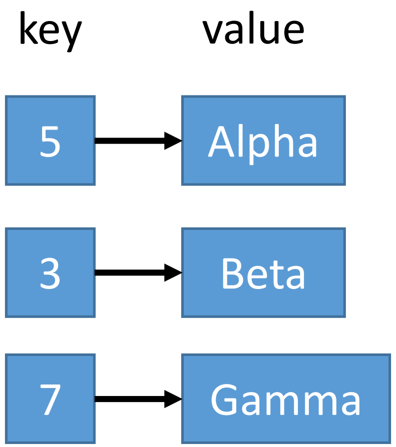
- sort
- stable_sort
- swap
- find_if
- any
- rotate
- lower_bound

Data Structures



Sequence

e.g., C-Arrays, std::vector, std::deque, std::list

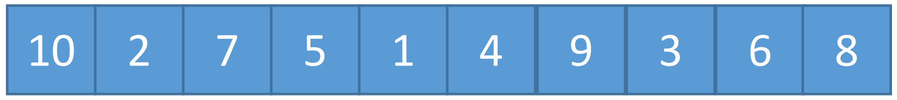


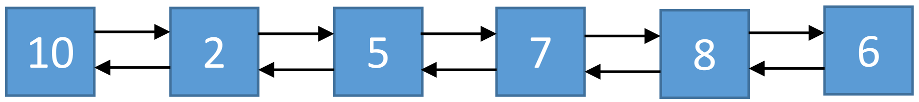
Associative

e.g., C-Arrays, std::map, std::set, std::unordered_map

Data Structures

- Operations:
  - Insertion
  - Searching
  - Deletion
- Variants:
  - Ordered
  - Unordered





Sequential Containers

Arrays, Lists, Queues, Stacks

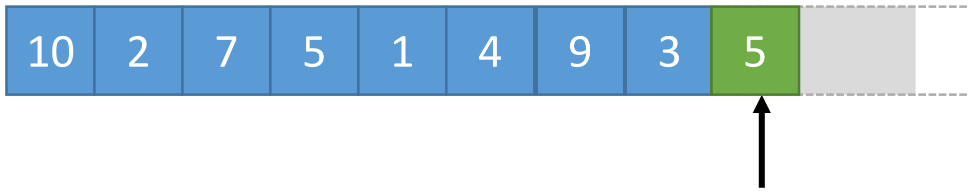
STL Containers

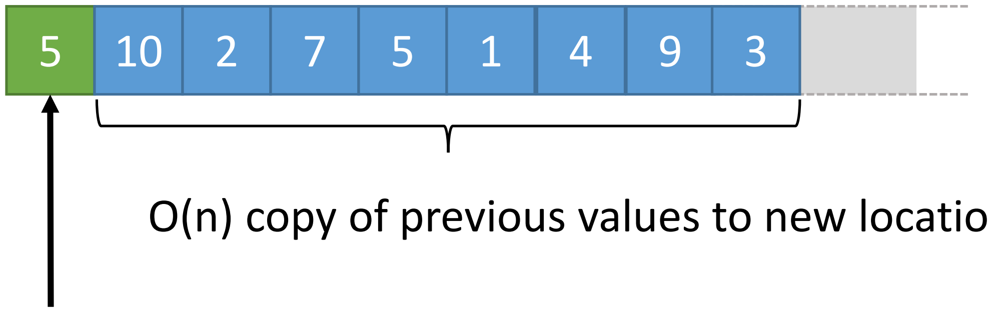
- Sequence Containers
  - vector (flexible sequence)
  - deque (double-ended queue)
  - list (double linked list)
  - array (fixed sequence, C++11)
  - forward_list (single linked list, C++11)

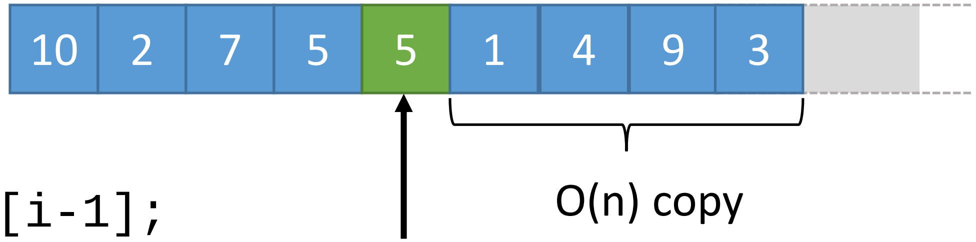
C-Arrays

- Simplest Sequence Data Structure
- Data stored in range [0, numElements)
- Fixed Size, Wasteful
- Consecutive Memory (efficient access)

```
int a[10000];
int numElements = 0;

// insertion at end O(1)
a[numElements++] = new_value;

// insertion at beginning O(n)
for(int i = numElements; i > 0; i--) a[i] = a[i-1];
a[0] = new_value;
numElements++;
```







std::vector

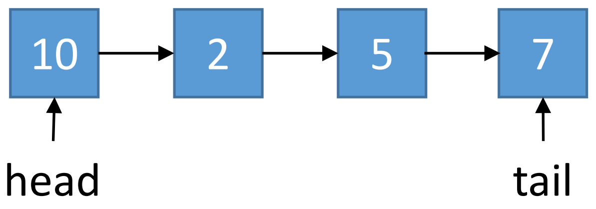
```
#include <iostream>
#include <vector>

using namespace std;

// empty construction
vector<int> a;
// sized construction
vector<int> a(10);
// sized construction with initial value
vector<int> a(100, -1);
// C++ 11 initializer lists
vector<int> a { 3, 5, 7, 9, 11 };

// insertion at end
a.push_back(3);
a.push_back(5);
a.push_back(7);

// delete at end
a.pop_back();

// insertion at beginning
a.insert(a.begin(), new_value);
```

```
// accessing elements just like arrays
for(int i = 0; i < a.size(); i++) {
   cout << a[i] << endl;
}

// using iterators
for(auto i = a.begin(); i != a.end(); ++i) {
   cout << *i << endl;
}

// C++11 for each
for(auto element : a) {
   cout << element << endl;
}
```

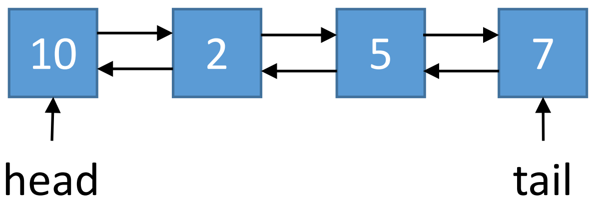
Linked-List

- List Elements connected through pointers
- First Element (head) and last element (tail) are always known
- Insertion/Deletion at **both** ends in O(1)
- Insertion in the middle is also cheaper
  - Finding insertion location is O(n) compared to O(1) with C-Arrays
  - But insertion itself happens in O(1) instead of O(n) copies
- Dynamic Size
- Distributed in memory

Single Linked-List:
only pointer of next element



Double Linked-List:
pointer of previous and next element




```
struct Node {
    Node * prev;
    Node * next;
    int data;
}
```

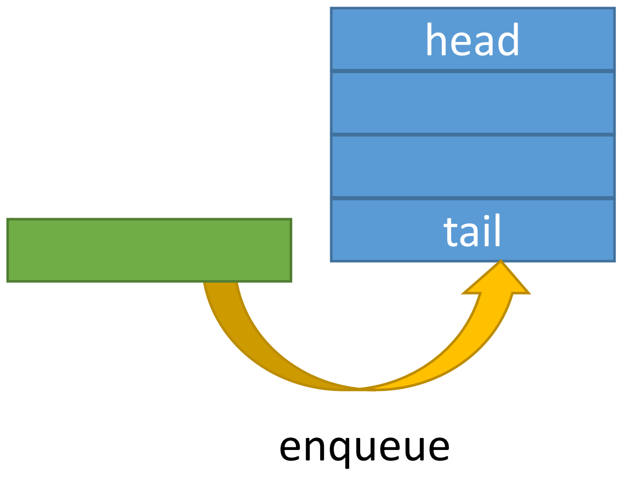
std::list

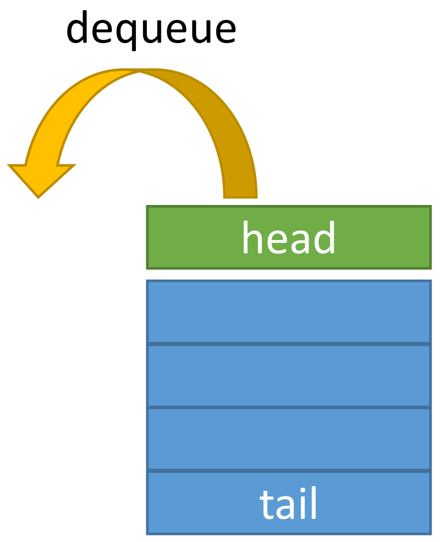
```cpp
#include <iostream>
#include <list>

using namespace std;

// empty construction
list<int> a;
// sized construction
list<int> a(10);
// sized construction with initial value
list<int> a(100, -1);
// C++ 11 initializer lists
list<int> a { 3, 5, 7, 9, 11 };

// insertion at beginning
a.push_front(3);

// insertion at end
a.push_back(3);

// delete at beginning
a.pop_front();

// delete at end
a.pop_back();
```

```cpp
// access front element
int first = a.front();

// access last element
int last = a.back();

// using iterators
for(auto i = a.begin(); i != a.end(); ++i) {
   cout << *i << endl;
}

// C++11 for each
for(auto element : a) {
   cout << element << endl;
}
```

Queue

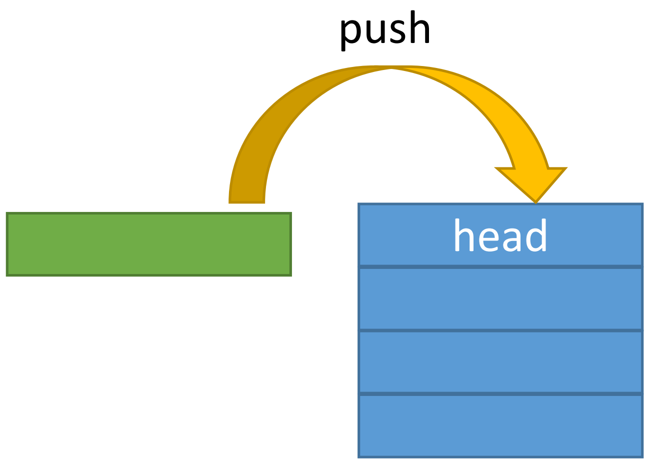
- First-In-First-Out (FIFO) data structure
- Implementations:
  - Double-Linked-List
- Operations:
  - **enqueue**: put element in queue (insert at tail)
  - **dequeue**: get first element in queue (remove head)

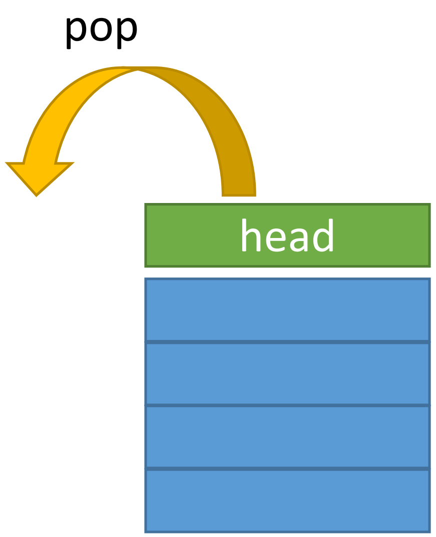


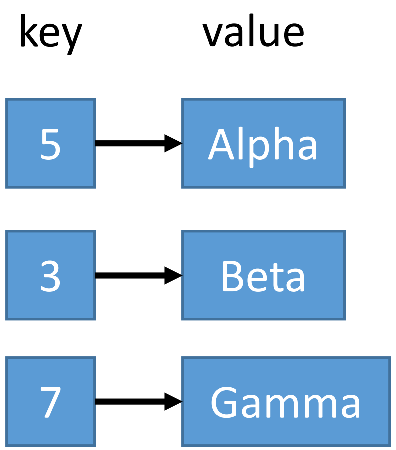


Stack

- Last-In-First-Out (LIFO) data structure
- Implementations:
  - C-Array
  - Single-Linked-List
- Operations:
  - **push**: put element on stack (insert as first element)
  - **pop**: get first element on stack (remove head)







Associative Containers

Dictionaries, Maps, Sets

Associative Containers

- Map a key to a value
- Searching for a specific element in unsorted sequential containers takes **linear** time O(n)
- Getting a specific element from an associative container can be as fast as **constant** time O(1)

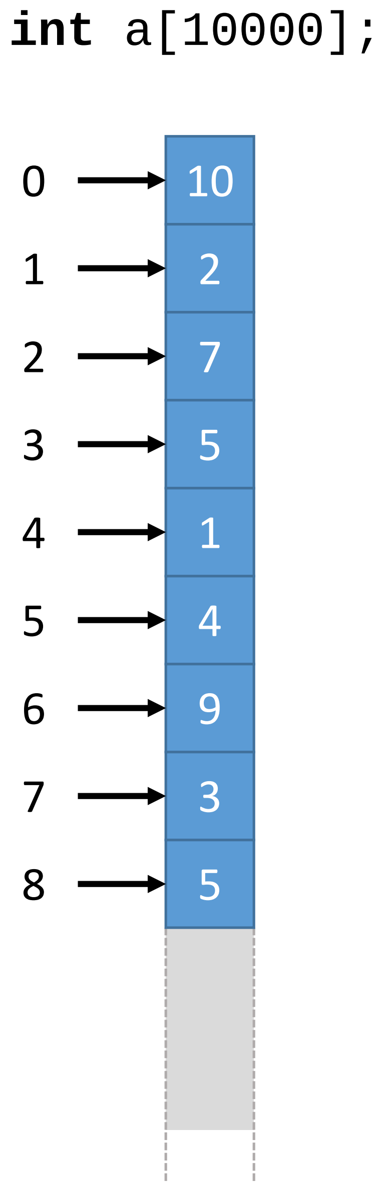
STL Containers

- Associative Containers
    - map
    - set
    - multimap
    - multiset

    - unordered_map (C++11)
    - unordered_set (C++11)
    - unordered_multimap (C++11)
    - unordered_multiset (C++11)

C-Array as Associative Container

- Simplest associative data structure
- maps **integer number** to data
  - 0 -> a[0]
  - 1 -> a[1]
  - …
- **efficient access in O(1)**
- **inefficient storage**
- **limited to positive integer numbers as keys**

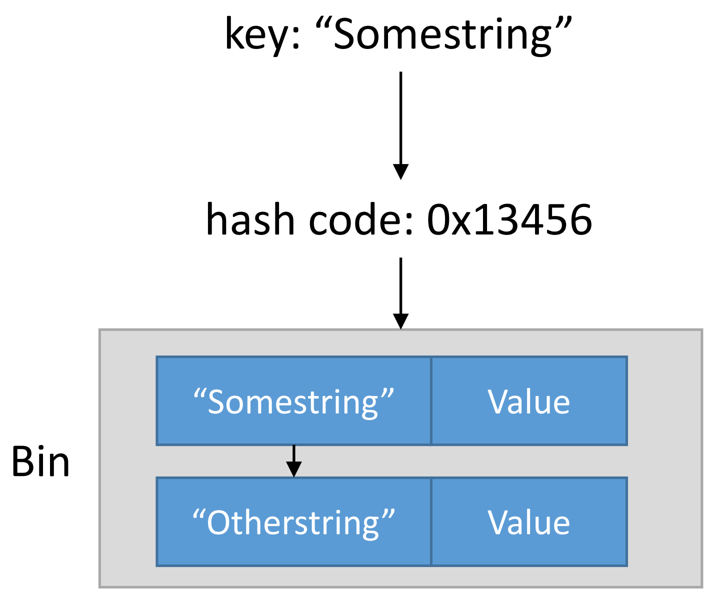


Unordered maps / Hash maps

On average lookup times are O(1). But this is only true if there are only few hash collisions.

Hash maps require **a good hashing function**, which reduces the amount of hash collisions.

- Maps **arbitrary keys** (objects, basic types) to **arbitrary values** (objects, basic types)
- **On average accessing a hash map through keys takes O(1)**
- In general unordered structure - you can't get out objects in the same order you inserted them.
- a number, called a **hash code**, is generated using a **hash function** based on key in O(1)
- Each hash code can be mapped to a location called a bin
- A bin stores nodes with keys which map to the same hash code
- Lookup therefore consists of:
  - Determining the hash code of the key O(1)
  - Selecting the correct node inside the bin is in the worst case O(n)



Ordered maps

- Maps **arbitrary keys** (objects, basic types) to **arbitrary values** (objects, basic types)
- Basic idea: if keys are sortable, we can store nodes in a data structure sorted by its keys. Sorted data structures can be searched more quickly, e.g. with binary search in O(log(n))
- Elements ordered by key
- **Worst case lookup time is O(log(n))**

std::map

```cpp
#include <iostream>
#include <map>
#include <string>

using namespace std;

map<string, string> capitals;

// setting value for key
capitals["Austria"] = "Vienna";
capitals["France"] = "Paris";
capitals["Italy"] = "Rome";

// getting value from key
cout << "Capital of Austria: " << capitals["Austria"] << endl;
string & capital_of_france = capitals["France"];
cout << "Capital of France: " << capitals << endl;


// check if key is set
if (capitals.find("Spain") != capitals.end()) {
    cout << "Capital of Spain is " << capitals["Spain"]  << endl;
else {
    cout << "Capital of Spain not found!" << endl;
}
```

std::map

```
// iterate over all elements
for (map<string, string>::iterator it = capitals.begin(); it != capitals.end(); ++it) {
    string & key = it->first;
    string & value = it->second;
    cout << "The capitol of " << key << " is " << value << endl;
}

// C++11: iterate over all elements
for (auto it = capitals.begin(); it != capitals.end(); ++it) {
    string & key = it->first;
    string & value = it->second;
    cout << "The capitol of " << key << " is " << value << endl;
}

// C++11: iterate over all elements
for (auto & kv : capitals) {
    string & key = kv.first;
    string & value = kv.second;
    cout << "The capitol of " << key << " is " << value << endl;
}
```

C++ Resources

- (Unofficial) C++ STL online reference: http://www.cplusplus.com/reference/stl/
- C++11 working draft: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf