[SWE2015-41] Introduction to Data Structures (자료구조개론)

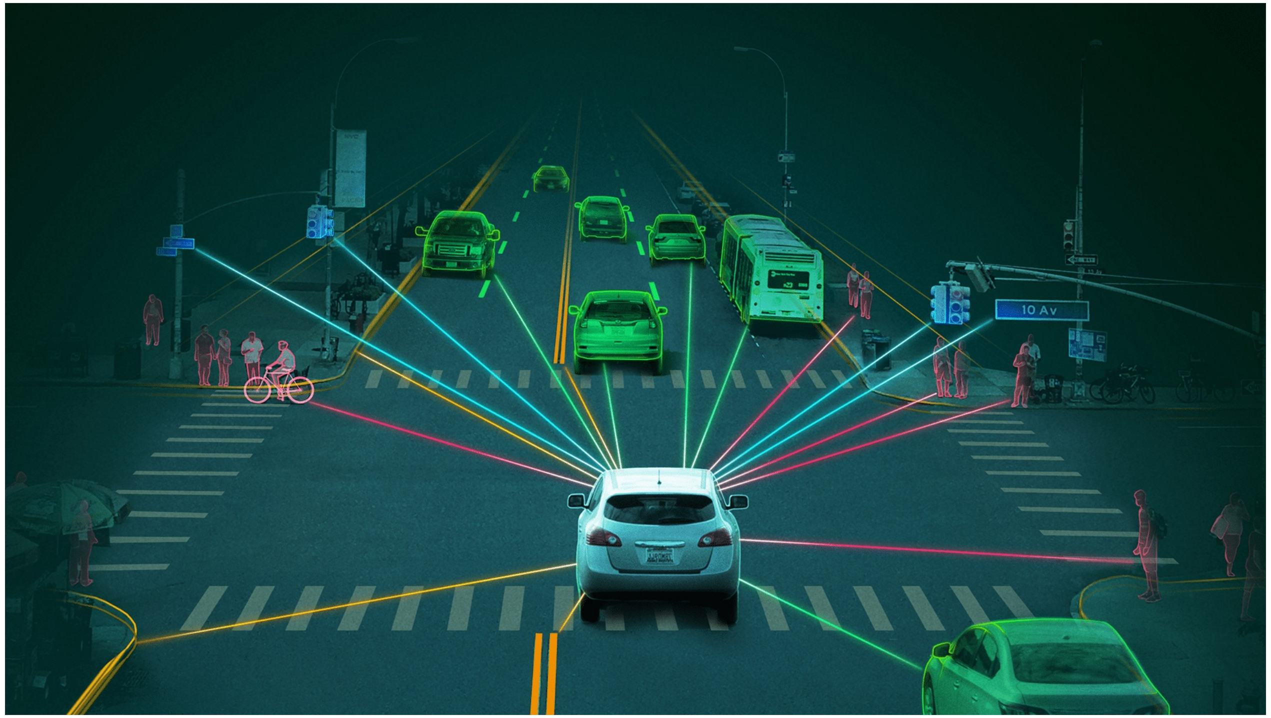
Introduction to Data Structures & Algorithms

Department of Computer Science and Engineering

Instructor: Hankook Lee (이한국)

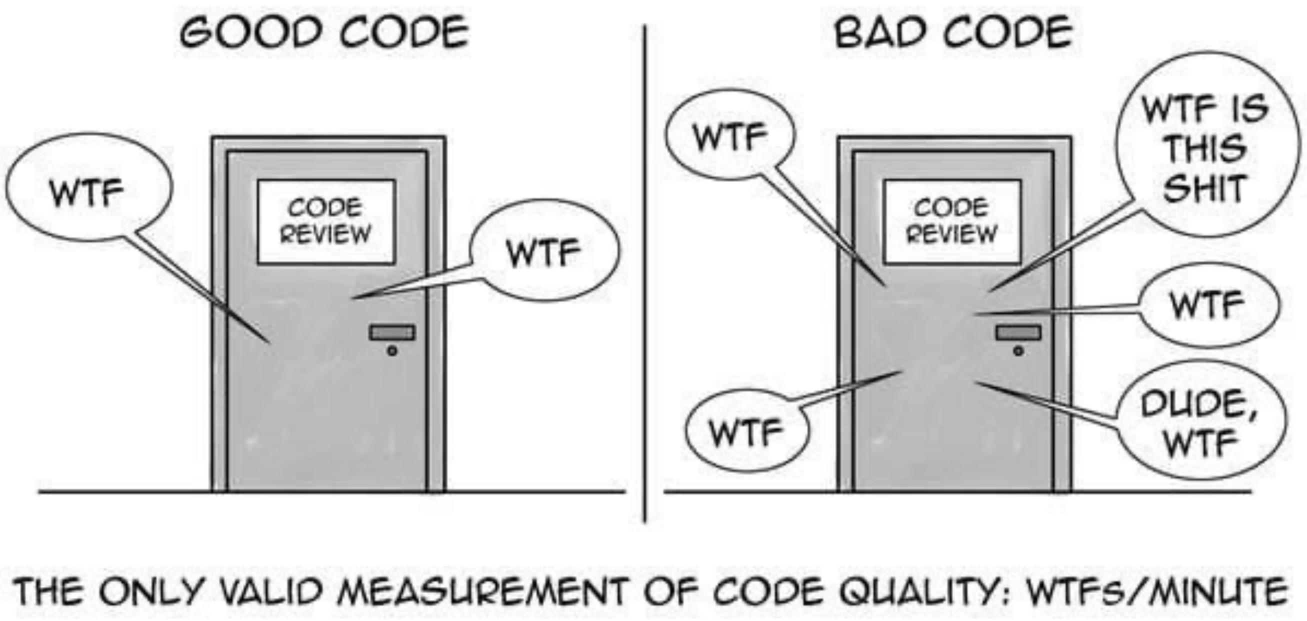
What is a Good Program?

- A good program should run **correctly**, **reliably**, and **efficiently** as expected
- Example: **Autonomous Driving**
  - Detect traffic conditions and obstacles in real-time
  - Figure out the optimal driving route



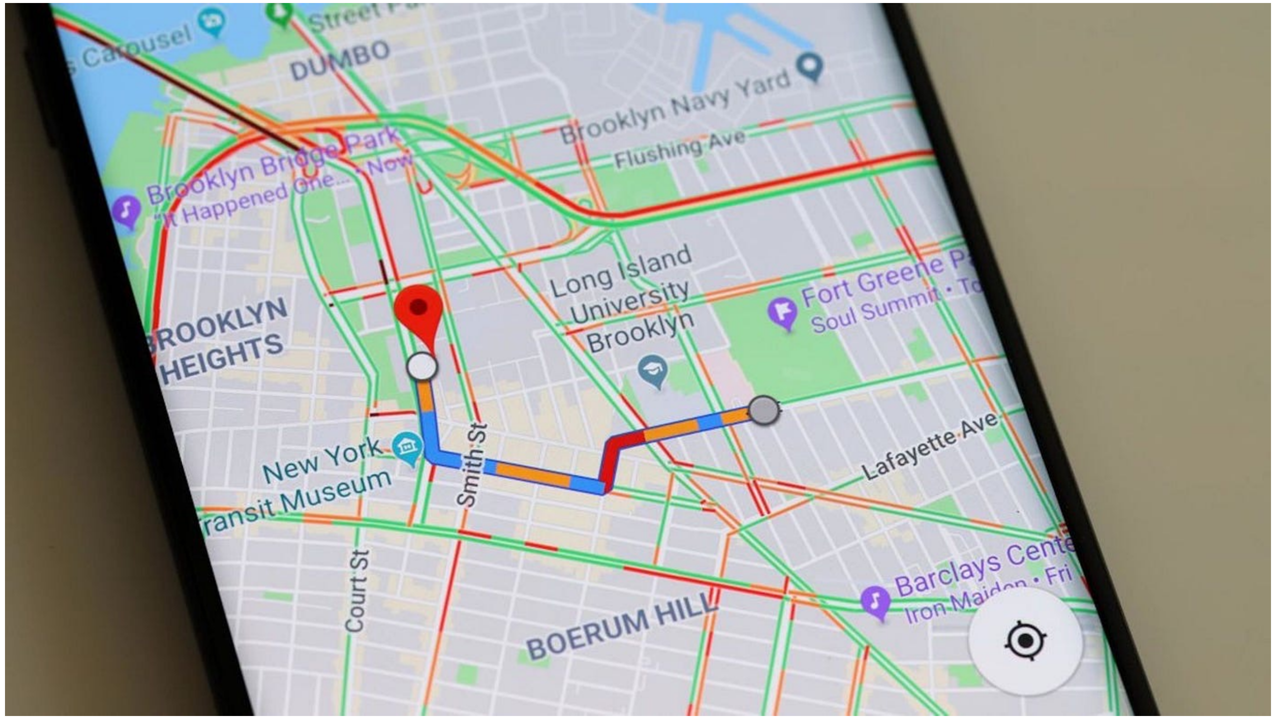
What is a Good Code?

- A good code should be …
  - **Simple** - Simple solution, Efficiency, …
  - **Readable** - Clear naming, Clear formatting, Comments, Documentation, …
  - **Maintainable** - Modularity, Reusability, Portability, …
  - **Reliable** - Error handling, Testing, Security, …



Problem Solving

- A problem is formulated by a **goal** and **requirements**
  - **Goal:** produce the desired outcome (e.g., shortest path between two places)
  - **Requirements:** time/memory limits, data/information access, devices, …



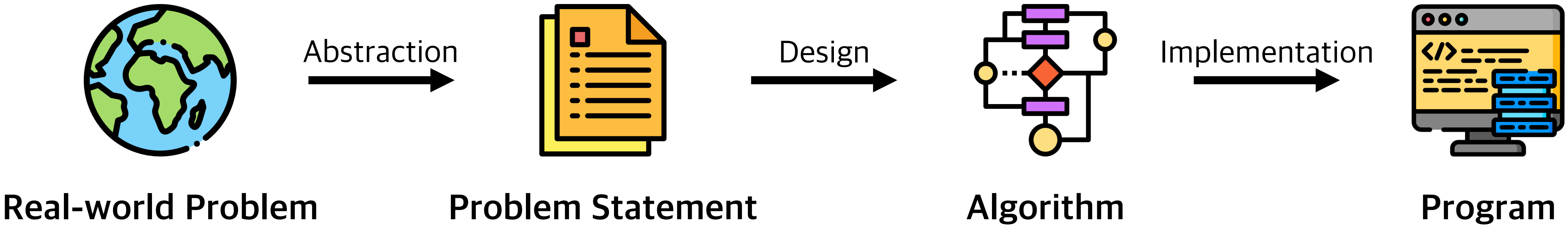
Path Finding

Face Recognition

Problem Solving

- A problem is formulated by a **goal** and **requirements**
  - **Goal:** produce the desired outcome (e.g., shortest path between two places)
  - **Requirements:** time/memory limits, data/information access, devices, …

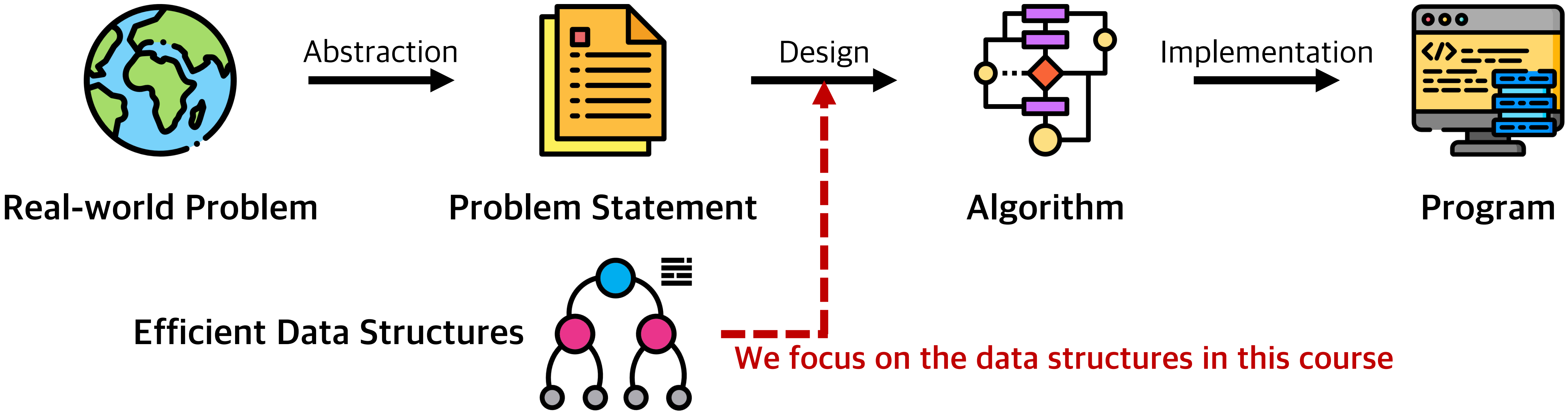
- Build an efficient program that can solve the problem:



Problem Solving

- A problem is formulated by a **goal** and **requirements**
  - **Goal:** produce the desired outcome (e.g., shortest path between two places)
  - **Requirements:** time/memory limits, data/information access, devices, …

- Build an efficient program that can solve the problem:



Problem Solving - Sub-sequential Sum

- **Problem:** calculate the sum between a-th and b-th integers (inclusive)

main.c

```c
int main() {
    int arr[5] = { 2, 1, 4, 3, 0 }, a, b;
    scanf("%d %d\n", &a, &b);

    int sum = 0;
    for (int i = a; i <= b; i ++) {
        sum += arr[i];
    }
    return 0;
}
```

Data Structure (Array)

Algorithm

Problem Solving - Sub-sequential Sum

- **Problem:** calculate the sum between a-th and b-th integers (inclusive)

main.c

```
int main() {
    int arr[5] = { 2, 1, 4, 3, 0 }, a, b;
    scanf("%d %d\n", &a, &b);

    int sum = 0;
    for (int i = a; i <= b; i ++) {
        sum += arr[i];
    }
    return 0;
}
```

Data Structure (Array)

Algorithm

- **(Q)** Is this the best algorithm?
  - Scenario #1: Given multiple queries, how to efficiently calculate?
  - Scenario #2: If data modification is available, how to efficiently calculate?

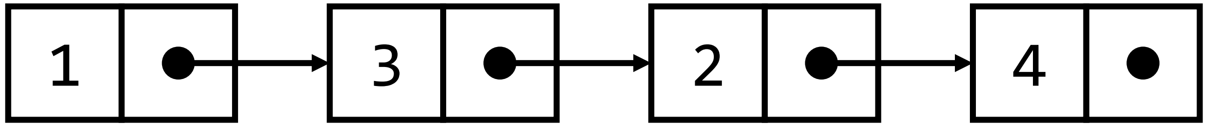
Data Structures

- A data structure is designed for efficient operations on a specific data type
  - Operations: addition, deletion, search, sorting, …
  - Example: find the maximum value among one million integers

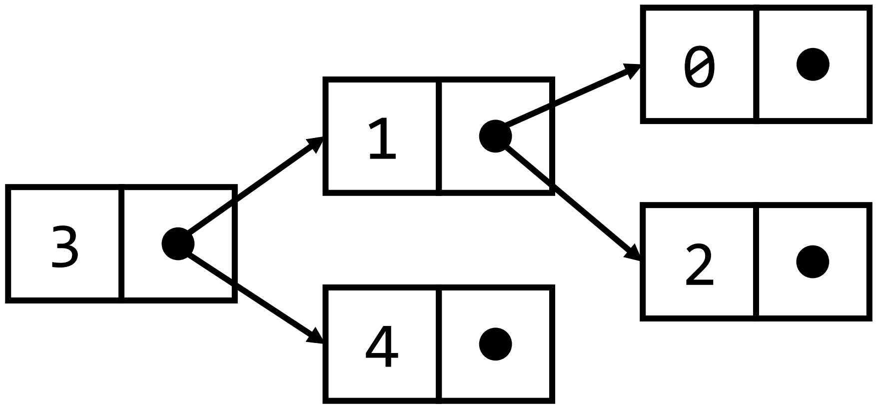
- Types of data structures

  Primitive types: characters (`char`), integers (`int`), floating-point numbers (`float`)

  Non-primitive types:
  - **Linear:** arrays, linked lists, stacks, queues, …
  - **Non-linear:** trees, graphs, …



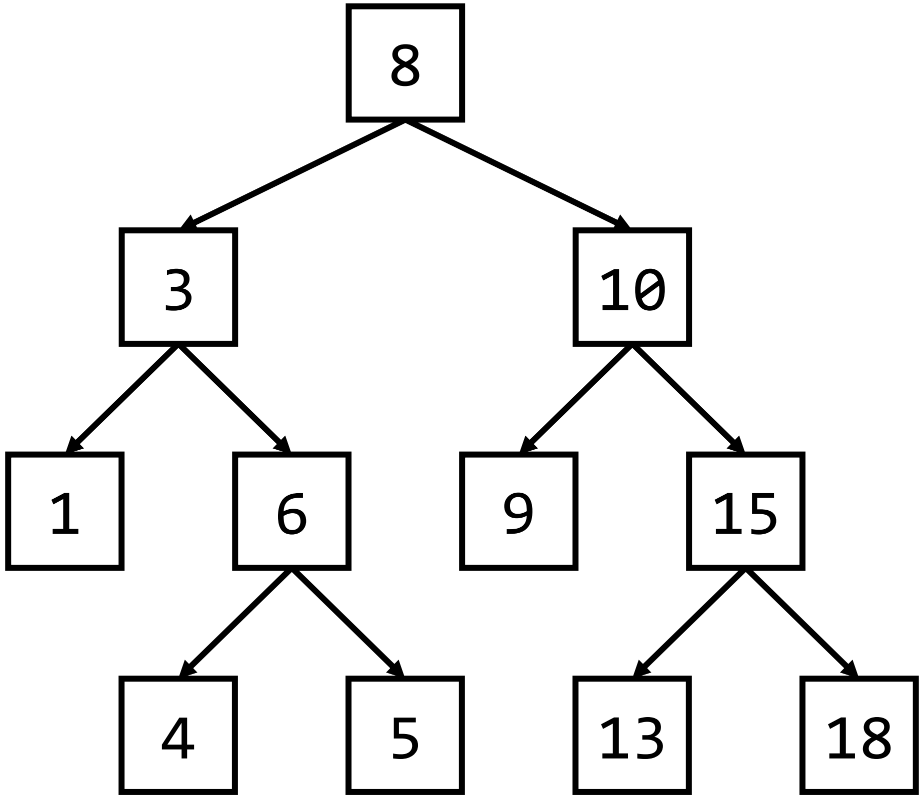
Linear Structure (Linked List)



Non-linear Structure (Binary Tree)

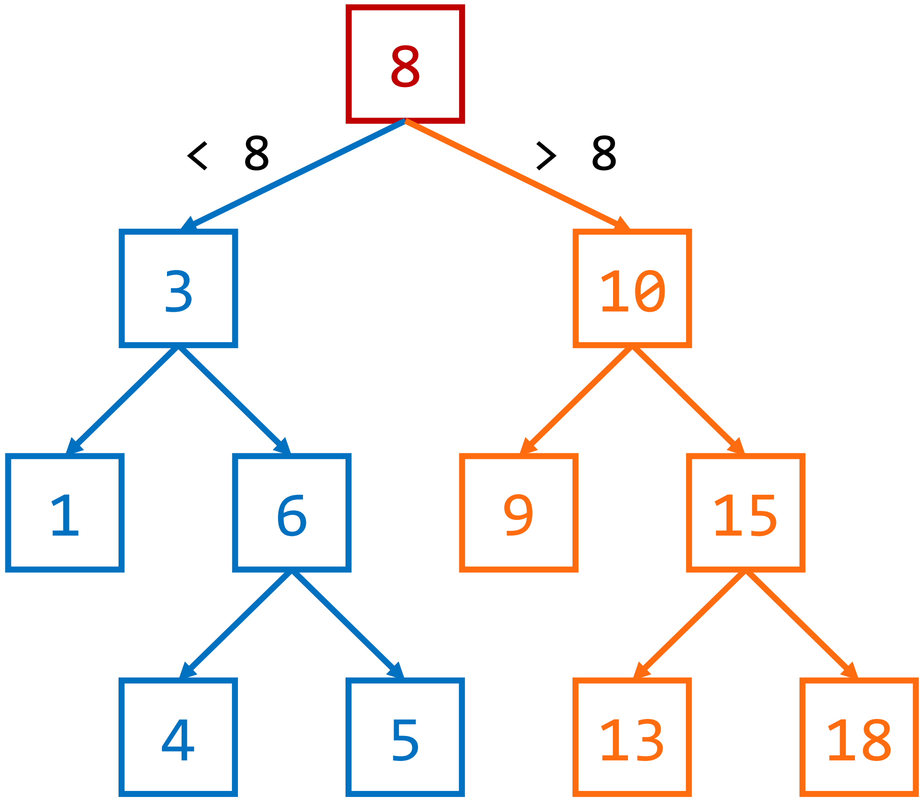
Data Structures - Binary Search Tree

- Binary Search Tree (BST) is designed for efficient search
  - Property: `left sub-tree < current node < right sub-tree`



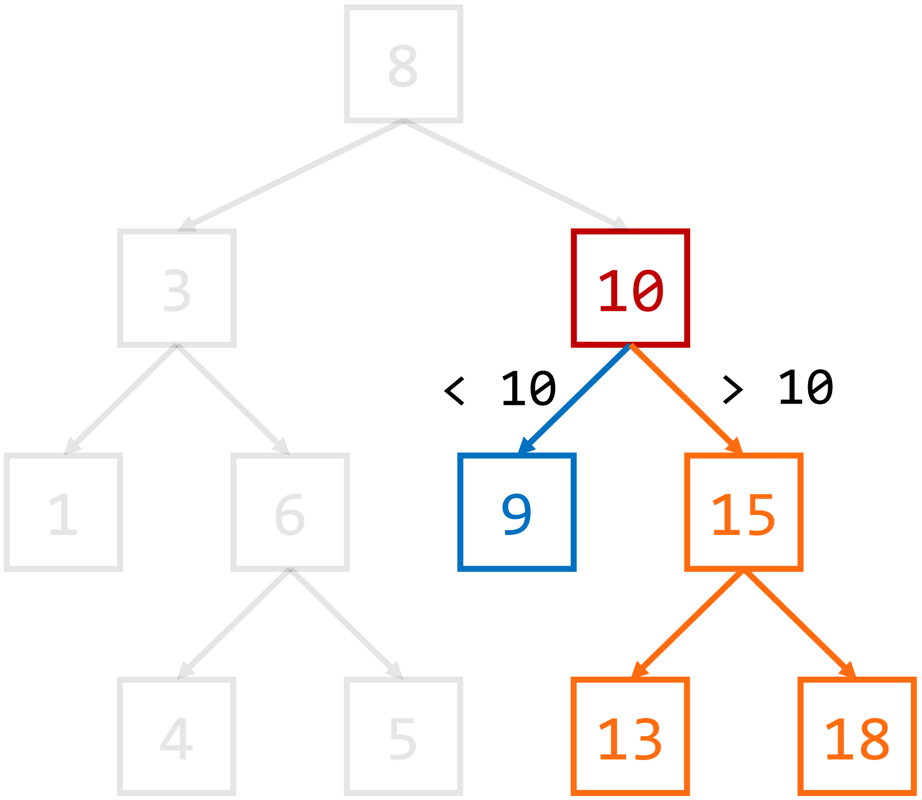
Data Structures - Binary Search Tree

- Binary Search Tree (BST) is designed for efficient search
  - Property: `left sub-tree < current node < right sub-tree`



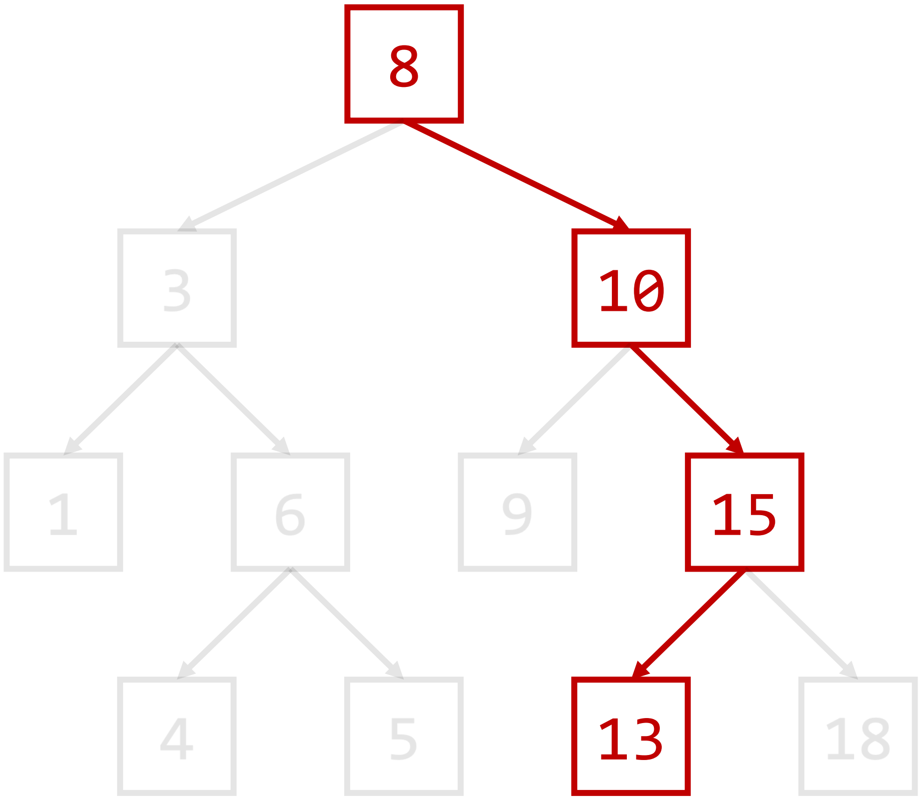
Data Structures - Binary Search Tree

- Binary Search Tree (BST) is designed for efficient search
  - Property: `left sub-tree` < `current node` < `right sub-tree`



Data Structures - Binary Search Tree

- Binary Search Tree (BST) is designed for efficient search
  - Property: `left sub-tree < current node < right sub-tree`
  - You can search a specific value (e.g., 13) efficiently using the property



Algorithms

- **Algorithm:** a formally defined procedure for performing some calculation
  - It can be implemented using a programming language (e.g., C, Python, …)
  - It provides a blueprint to write a program to solve a particular problem
  - A well-defined algorithm always provides an answer and is guaranteed to terminate

Algorithms

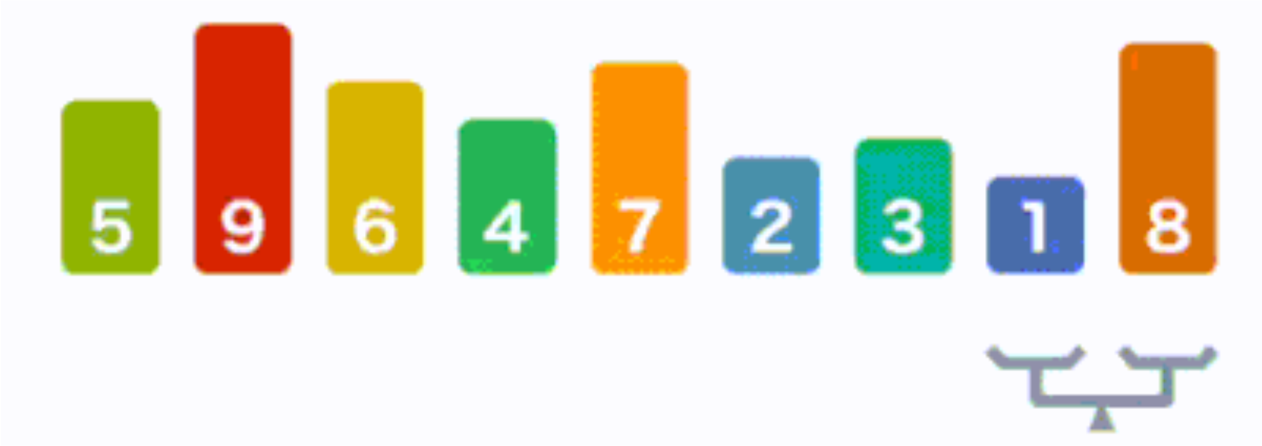
- **Algorithm:** a formally defined procedure for performing some calculation
  - It can be implemented using a programming language (e.g., C, Python, …)
  - It provides a blueprint to write a program to solve a particular problem
  - A well-defined algorithm always provides an answer and is guaranteed to terminate

- Well-known algorithms
  - **Sort:** Bubble Sort, Insertion Sort, Quick Sort, …
  - **Search:** Binary Search, Depth-First Search (DFS), Breadth-First Search (BFS)
  - **Graph:** Matching, Shortest Path, Minimum Spanning Tree
  - **Dynamic Programming**

Algorithms

- How to describe an algorithm?
  - Inputs → A set of formally-defined instructions → Outputs
  - Instructions can be written by a specific programing language or pseudo code

  - An example: Bubble Sort
    - This first (i) compares adjacent elements and then (ii) swap them if reversed



Algorithms

- How to describe an algorithm?
  - Inputs → A set of formally-defined instructions → Outputs
  - Instructions can be written by a specific programing language or pseudo code

  - An example: Bubble Sort
    - This first (i) compares adjacent elements and then (ii) swap them if reversed

C

```c
void bubble_sort(int *arr, int n) {
    int i, j;
    for (i = 0; i < n; i ++) {
        for (j = n-1; j > i; j --) {
            if (arr[j-1] > arr[j])
                swap(&arr[j-1], &arr[j]);
        }
    }
}
```

Algorithms

- How to describe an algorithm?
  - Inputs → A set of formally-defined instructions → Outputs
  - Instructions can be written by a specific programing language or pseudo code

  - An example: Bubble Sort
    - This first (i) compares adjacent elements and then (ii) swap them if reversed

Pseudo Code

```
BubbleSort(arr[], n):
    FOR i ← 0 to n-1
      FOR j ← n-1 to i+1
        IF arr[j-1] > arr[j]
          swap arr[j-1] and arr[j]
```

    - Pseudo code is more readable and simpler
    - There is no ground rule for pseudo code syntax/style, so recommend to refer someone’s one

Algorithm Performance Analysis

- How to evaluate performance of an algorithm?
  - **Correctness** - Whether the algorithm is accurate
  - **Efficiency** - How efficient the algorithm is

Algorithm Performance Analysis

- How to evaluate performance of an algorithm?
  - **Correctness** - Whether the algorithm is accurate
  - **Efficiency** - How efficient the algorithm is

- **Efficiency** is the important metric for comparison between algorithms
  - Time efficiency - How long does the algorithm take?
  - Memory efficiency - How much memory does the algorithm occupy?

Algorithm Performance Analysis

- How to evaluate performance of an algorithm?
  - **Correctness** - Whether the algorithm is accurate
  - **Efficiency** - How efficient the algorithm is

- **Efficiency** is the important metric for comparison between algorithms
  - Time efficiency - How long does the algorithm take?
  - Memory efficiency - How much memory does the algorithm occupy?

- **Complexity** is a machine-independent metric for efficiency comparison
  - Time complexity - the number of machine instructions used in the algorithm
  - Space complexity - the number of primitive variables used in the algorithm
  - They are expressed by input sizes (e.g., the number of integers, n)

Asymptotic Notation

- Consider $f(n) = n^2 + 10n + \log_{10} n$
- If n is large enough, which term is more important?

n	n^2	$10n$	$\log_{10} n$	$f(n)$
1	1	10	0	11
10	100	100	1	201
100	10000	1000	2	11002
1000	1000000	10000	3	1010003
10000	100000000	100000	4	100100004

- When $n \to \infty$, one can roughly say …
  - $f(n) \approx n^2$
  - $g(n) = 100n$ is increasing slower than $f(n)$
  - $g(n) = 2^n$ is increasing faster than $f(n)$

Asymptotic Notation

- **Big-O notation** $O(\cdot)$ - Asymptotic Upper Bound

$$f(n) = O\big(g(n)\big) \text{ if } \exists c > 0, \exists n_0 \in \mathbb{N} \text{ such that } f(n) \leq cg(n) \text{ for all } n \geq n_0$$

$f(n)$ is not increasing faster than $g(n)$

- Examples
  - $n^2 + 10n = O(n^2)$
  - $n^2 + 10n = O(n^3)$
  - $n^3 + n + 5 \neq O(n^2)$

- Exercises
  - Prove $n = O(2^n)$ and $\log n = O(n)$
  - Prove $f(n) = O\big(h(n)\big)$ when $f(n) = O\big(g(n)\big)$ and $g(n) = O\big(h(n)\big)$.

Asymptotic Notation

- **Omega notation** $\Omega(\cdot)$ - Asymptotic Lower Bound

$$f(n) = \Omega\big(g(n)\big) \text{ if } \exists c > 0, \exists n_0 \in \mathbb{N} \text{ such that } f(n) \geq cg(n) \text{ for all } n \geq n_0$$

$f(n)$ is not increasing slower than $g(n)$

- Examples
  - $n^2 + 10n = \Omega(n^2)$
  - $n^2 + 10n = \Omega(n)$
  - $n^3 + n + 5 \neq \Omega(n^4)$

- Exercises
  - Prove $n! = n \times (n-1) \times \cdots \times 1 = \Omega(2^n)$
  - Prove $f(n) = \Omega\big(h(n)\big)$ when $f(n) = \Omega\big(g(n)\big)$ and $g(n) = \Omega\big(h(n)\big)$.

Asymptotic Notation

- **Theta notation** $\Theta(\cdot)$ - Asymptotic Tight Bound

$$f(n) = \Theta\big(g(n)\big) \text{ if } f(n) = O\big(g(n)\big) \text{ and } f(n) = \Omega\big(g(n)\big)$$

$f(n)$ is equally increasing to $g(n)$

- Examples
  - $n^2 + 10n = \Theta(n^2)$
  - $n^3 + n + 5 = \Theta(n^3)$
  - $n^3 + 2^n = \Theta(2^n)$

- Exercises
  - Prove $\log_2 n = \Theta(\log_3 n)$
  - Prove $2^n \neq \Theta(3^n)$

Asymptotic Notation

- Asymptotic notations
  - **Big-O notation** $O(\cdot)$ - Asymptotic Upper Bound
  - **Omega notation** $\Omega(\cdot)$ - Asymptotic Lower Bound
  - **Theta notation** $\Theta(\cdot)$ - Asymptotic Tight Bound

  - They allow us to calculate and compare time/space complexities easier

- **Big-O notation** (as tight as possible) is commonly used
  - This is because it is hard to compute the tight bound
  - Lower bound is often not informative for algorithm performance analysis
  - E.g., say $n^2 + 10n = O(n^2)$ rather than $n^2 + 10n = O(n^3)$

Time Complexity

- Constant-time operations → $O(1)$
  - Arithmetic operations: addition, subtraction, multiplication, division, …
  - Comparison: equality, inequality, …
  - Variable declaration and assignments

C

```c
void constant_operations() {
    int a = 16, b = 2, c;
    c = a/4 + b*2 + 3;
    if (c > 10) c -= 10;
}
```

Time Complexity

- Linear-time complexity → $O(n)$

C

```c
int sum(int n) {
    int sum = 0;
    for (int i = 0; i < n; i ++)
        sum += i;
    return sum;
}
```

1. [`int sum = 0;`] → $O(1)$ → c_1 time complexity
2. [`int i = 0; i < n; i ++`] → $O(1)$ → c_2 time complexity
3. [`sum += i;`] → $O(1)$ → c_3 time complexity
4. [2] and [3] are repeated `n` times

- Therefore, the total time complexity is $(c_2 + c_3) \times n + c_1 = O(n)$

Time Complexity - Exercise

- Exercise - Print n-by-n identity matrix

C

```c
void print_identity_matrix(int n) {
    int i, j;
    for (i = 0; i < n; i ++) {
        for (j = 0; j < n; j ++) {
            if (i == j) printf("%d ", 1);
            else printf("%d ", 0);
        }
        printf("\n");
    }
}
```

n = 4

1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

Time Complexity - Exercise

- Exercise - Compute the bit length of an integer `n`

C
```c
int bit_length(int n) {
    int length = 0;
    while (n > 0) {
        length += 1
        n /= 2;
    }
    return length;
}
```

```
n      bits     bit_length(n)
4   = 100       3
5   = 101       3
10  = 1010      4
127 = 1111111   7
```

- When compute all bit lengths from `1` to `N`, what is the time complexity?

C
```c
for (int i = 1; i <= N; i ++)
    printf("%d\n", bit_length(i));
```

Space Complexity

- One primitive variable → 1~8 bytes (`char`, `int`, `double`, …) → $O(1)$
- An array of `n` variables → $O(n)$

C

```c
int fibonacci_sum(int n) {
    int i, sum = 0, *arr;
    arr = malloc(sizeof(int)*n);
    arr[0] = 1; arr[1] = 1;
    for (i = 2; i < n; i ++) arr[i] = arr[i-1] + arr[i-2];
    for (i = 0; i < n; i ++) sum += arr[i];
    free(arr);
    return sum;
}
```

```
n = 7
1 1 2 3 5 8 13
fibonacci_sum(n) = 33
```

- This uses $12 + 4n = O(n)$ bytes
- **(Q)** Can we reduce the memory usage to $O(1)$ bytes?

Space Complexity

- One primitive variable → $O(1)$
- An array of n variables → $O(n)$

C

```c
int fibonacci_sum(int n) {
    int i, sum = 2, arr[3] = { 1, 1, 0 };
    for (i = 2; i < n; i ++) {
        arr[i%3] = arr[(i+1)%3] + arr[(i+2)%3];
        sum += arr[i%3];
    }
    return sum;
}
```

```
n = 7
1 1 2 3 5 8 13
fibonacci_sum(n) = 33
```

- This uses only $18 = O(1)$ bytes!

Algorithm Comparison

- Which algorithm is better?
  - **Average-case** - The expected complexity when the input is randomly drawn
  - **Worst-case** - The complexity w.r.t. the worst-possible case of the input instance

Sort Algorithm	Time Complexity (Average)	Time Complexity (Worst)	Space Complexity (without input array)
Insertion Sort	$O(n^2)$	$O(n^2)$	$O(1)$
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(1)$
Quick Sort	$O(n \log n)$	$O(n^2)$	$O(\log n)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n)$

- **Note.** Running time requirements are more critical than memory requirements

Any Questions?