EE2331 Data Structures and Algorithms

Introduction and a sorting algorithm

Today's agenda:

1) Introduce basic concepts of data structure and algorithm

2) Insertion sort

Two major topics

- Basic data structures
- Algorithms

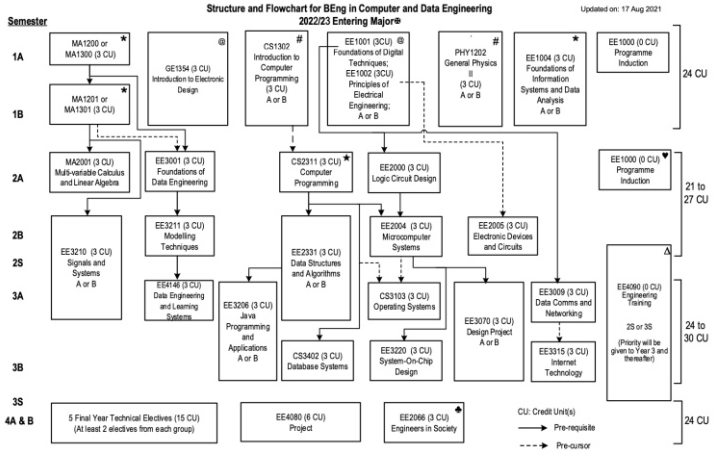
You are given as set of cards with students' names on them. How to **organize them** so that we can **easily** find a student, remove a student, add a student, sort the student by names?

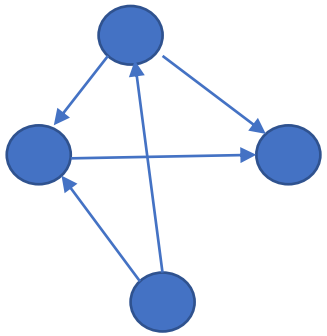
I have about 100 students in EE2331. How should I organize the test papers so that a student can find his/her test paper fast?

Data structures

- Data structure provides a way to organize data items. Each data structure has associated operations
  - What are the data structures in the following examples?
    - Example 1. MTR map

Example 2: flow chart of CDE



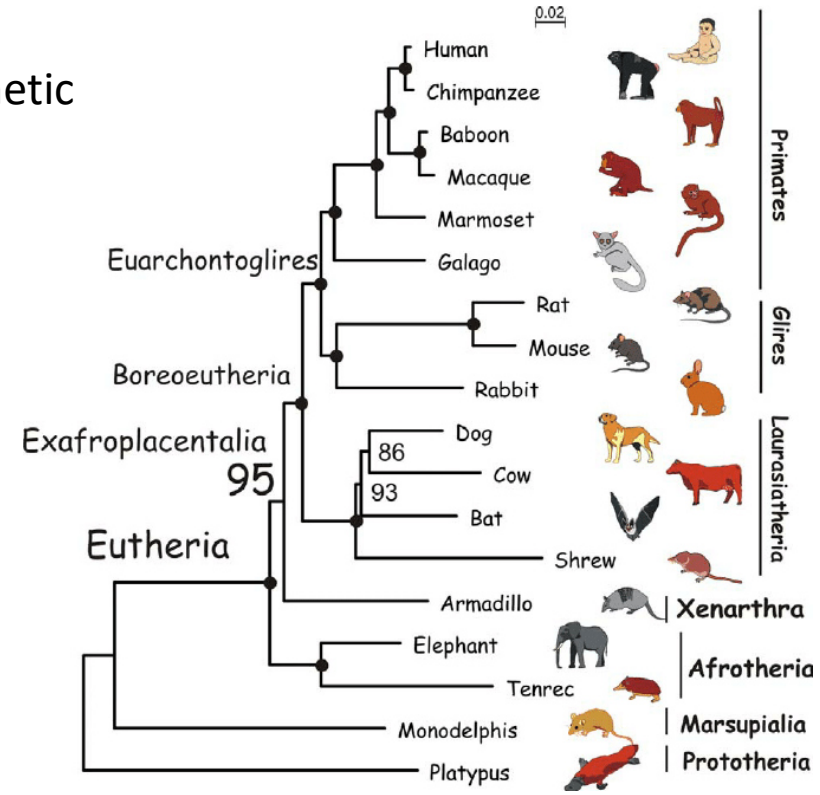


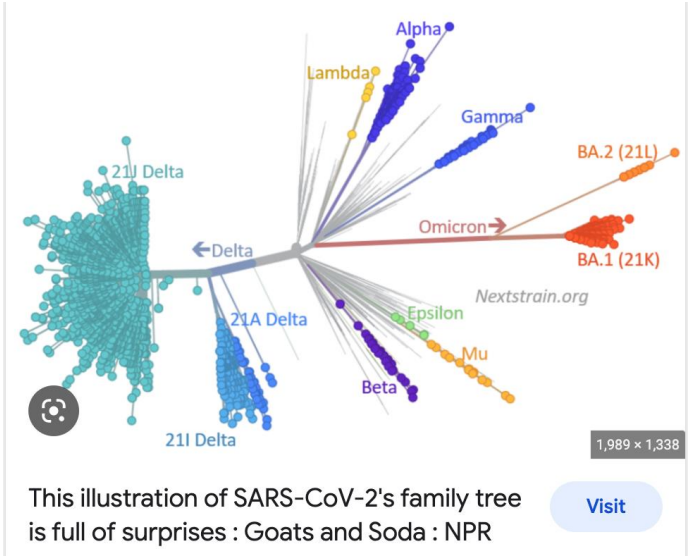
graph

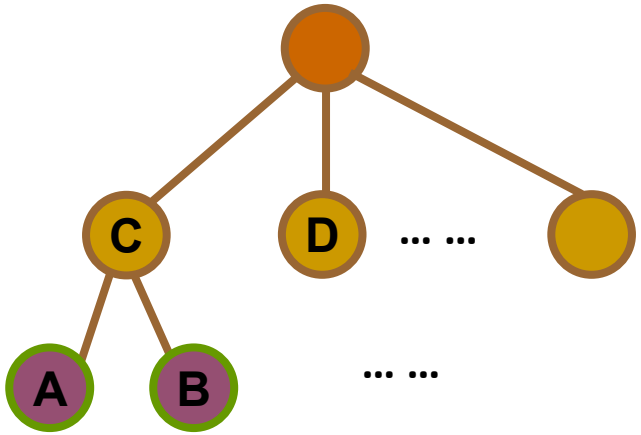
Data structures

- Data structure provides a way to organize data items. Each data structure has associated operations
  - What are the data structures in the following examples?

Example: Phylogenetic tree







Tree -> not reliable

Data structures

- Data structure provides a way to organize data items. Each data structure has associated operations
  - What are the data structures in the following examples?

Name	Age
Andy	5
Judy	6
Mathew	7
Raymond	5.5
Hayden	7.5

list

Question: How to you organize all the students' records at CityU? What data structure do you choose?

Algorithms

- "a bag of tricks", can be executed systematically by the computer

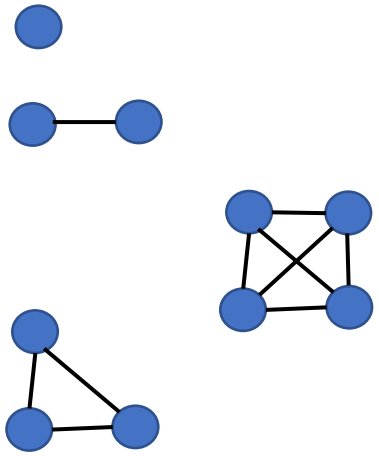
Basic algorithmic techniques:	A way to think about computation
• **sorting**, **searching**	• What is a "good" algorithm? • What does "fast/ faster" mean?

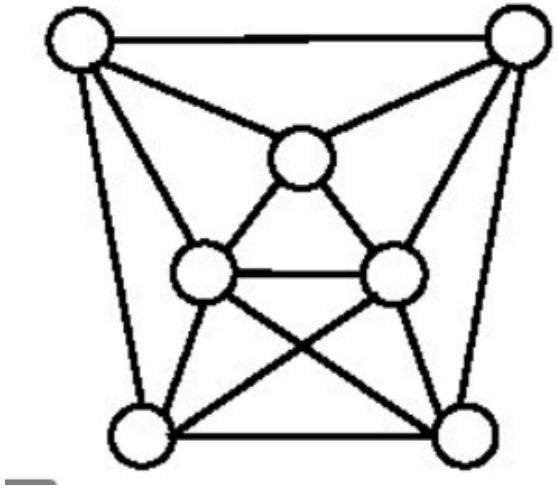
Algorithms

- Example problems.
  The **Subway Challenge** is a challenge in which participants must navigate the entire New York City Subway system in the shortest time possible.
  - Can you solve this on Hong Kong MTR?

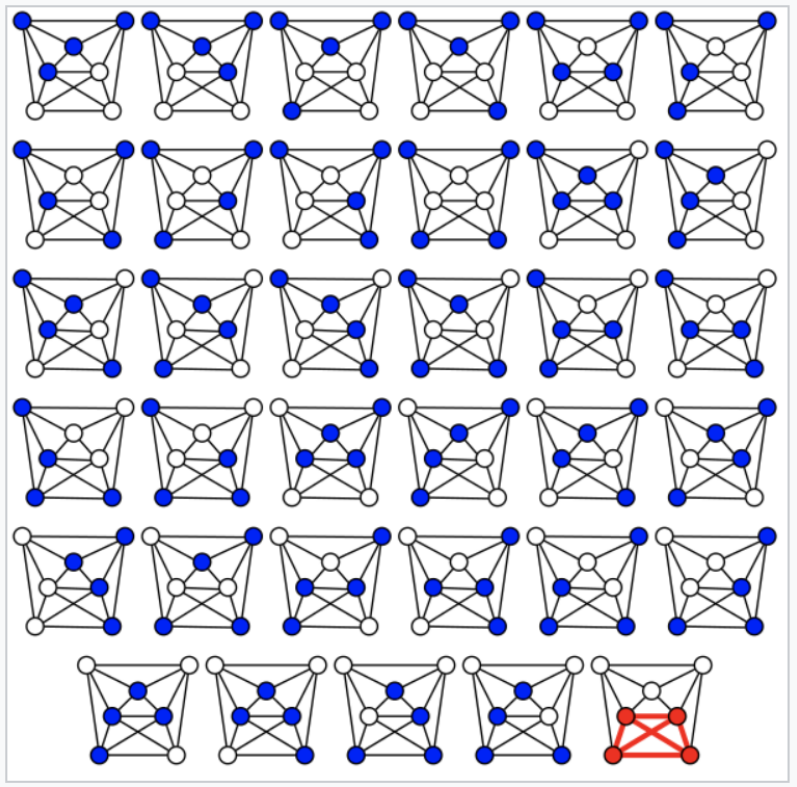
Algorithms

- Clique problem. What is a clique: subsets of vertices, all adjacent to each other, also called complete subgraphs in a graph
- **Maximum clique problem: identify the maximum clique in a graph. Can you come up with an algorithm?**





What is the maximum clique in this graph?



Brue force algorithm

Algorithms

- Analytical techniques
  - Asymptotic notation (next topic, stay tuned)

- **Summary**
  - Data structure & algorithms are practical and basic to computer science culture (more than just writing code)

First problem: Sorting

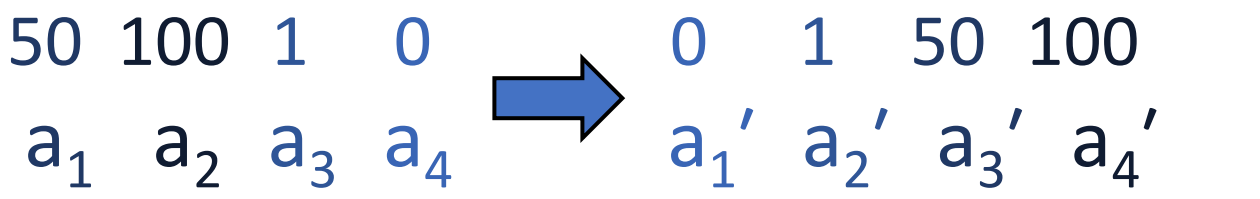
- Input: a sequence of n numbers

  $<a_1, a_2, \ldots, a_n>$

- Output: a permutation (re-ordering)

  $< a_1', a_2', \ldots, a_n'>$ of the input sequences,

  s.t. $a_1' \leq a_2' \leq a_3' \leq \ldots \leq a_n'$

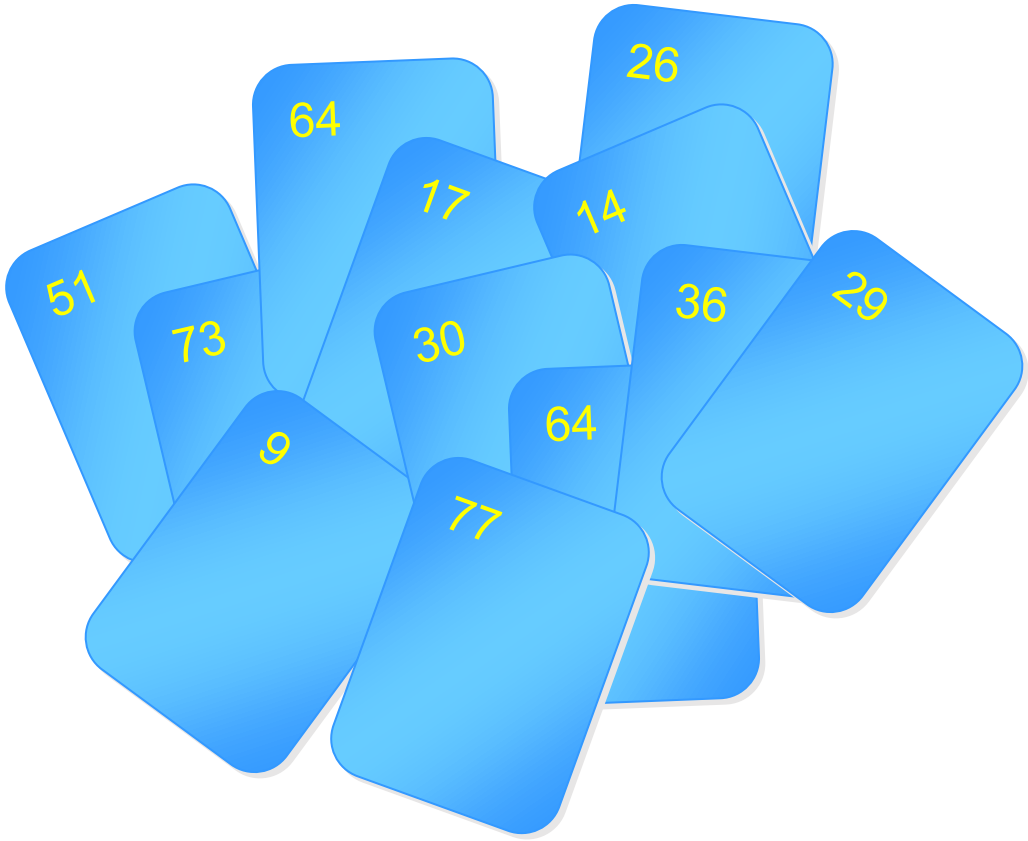


Question: can you think of some examples of sorting in real life?

(dictionary is a very good example)

If a list is sorted…

- How to find the largest number?
- How to find the smallest number?
- How to determine if an arbitrary number exists in the list?



Sorting

- To rearrange the order (ascending, descending, increasing, decreasing, non-decreasing) of data for **ease of searching**

- We will discuss various ways to sort a large amount of data and compare them by time/space efficiency.

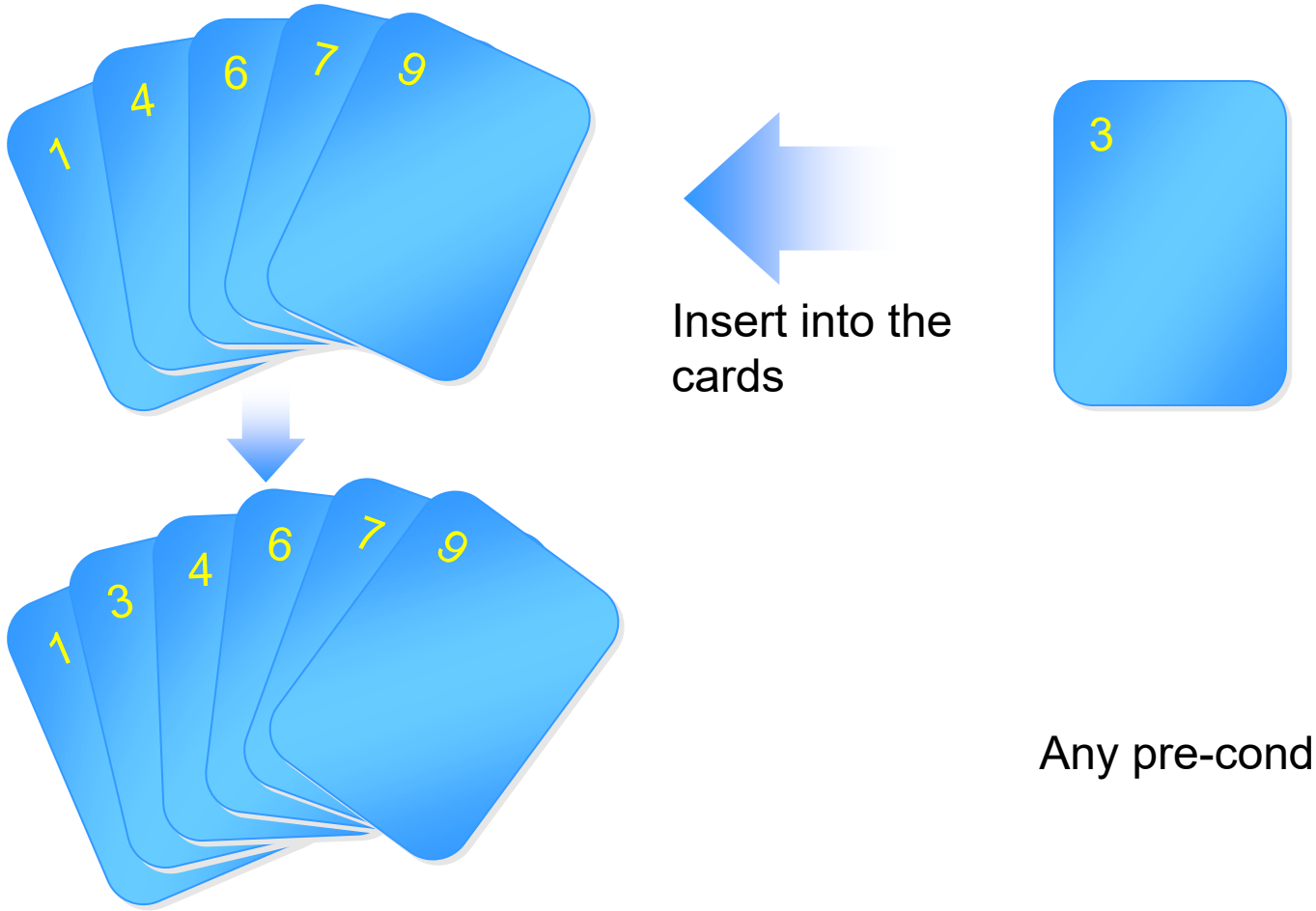
Insertion Sort

A basic sorting algorithm.

Basic operation: insert an element into a sorted list such that the final list is still sorted

Daily Life Example

- The idea of insertion is like playing cards



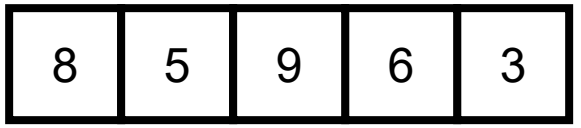
Any pre-condition?

Insertion Sort

- Insertion sort successively inserts a new element into a (sorted) sublist in each pass
- Initially 1st element may be thought of as a sorted sublist of only one element
- After each sorted-insertion, the sorted sublist's length grows by 1.
- Insertion sort makes use of the fact that elements in the sublist are already known to be in sorted order.

Insertion Sort Example

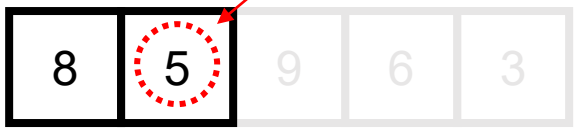
The unsorted list:



Consider the 1st element as a *sorted* sublist

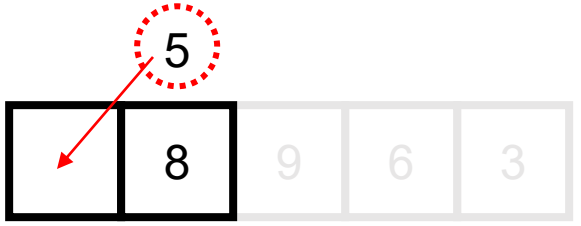
Insert this element into the left sublist such that they maintain a proper order

The 1st pass



Ignore them in current pass

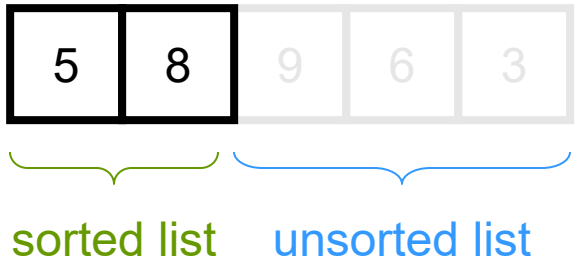
Pick up “5”. Move “8” to right



Insert “5” to the appropriate position

Insertion Sort Example

After 1st pass

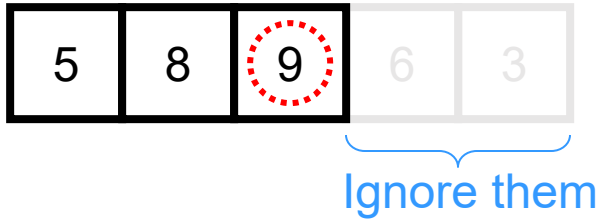


1 Move!

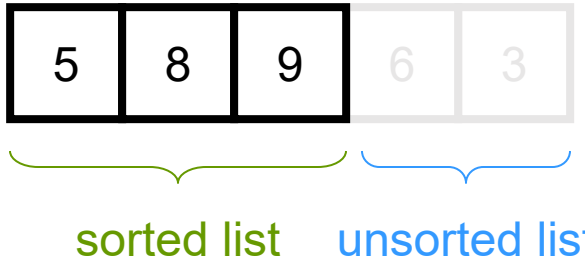
Compare with this sublist only

Insert this element into the left sublist such that they maintain a certain order

The 2nd pass

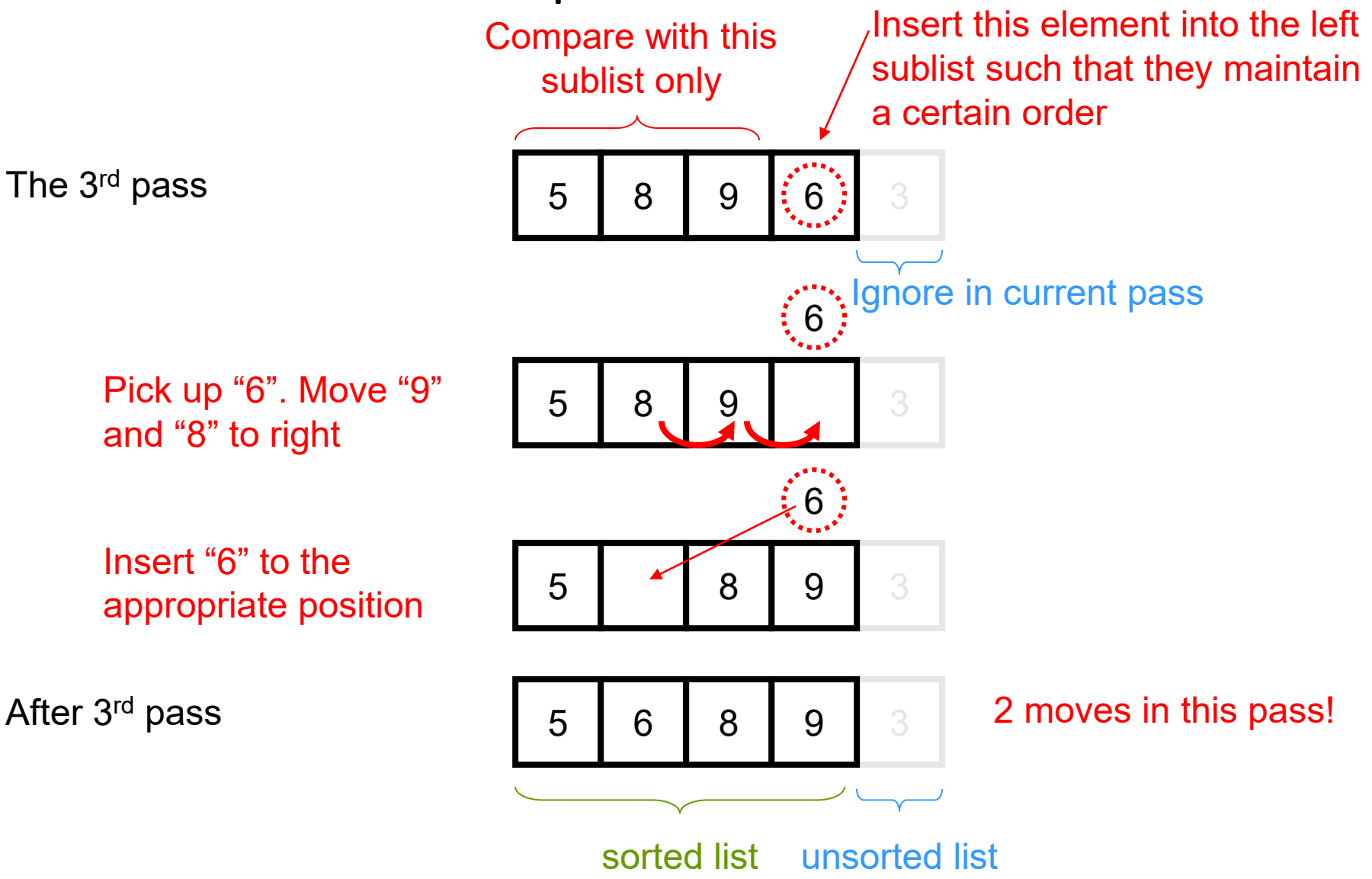


After 2nd pass

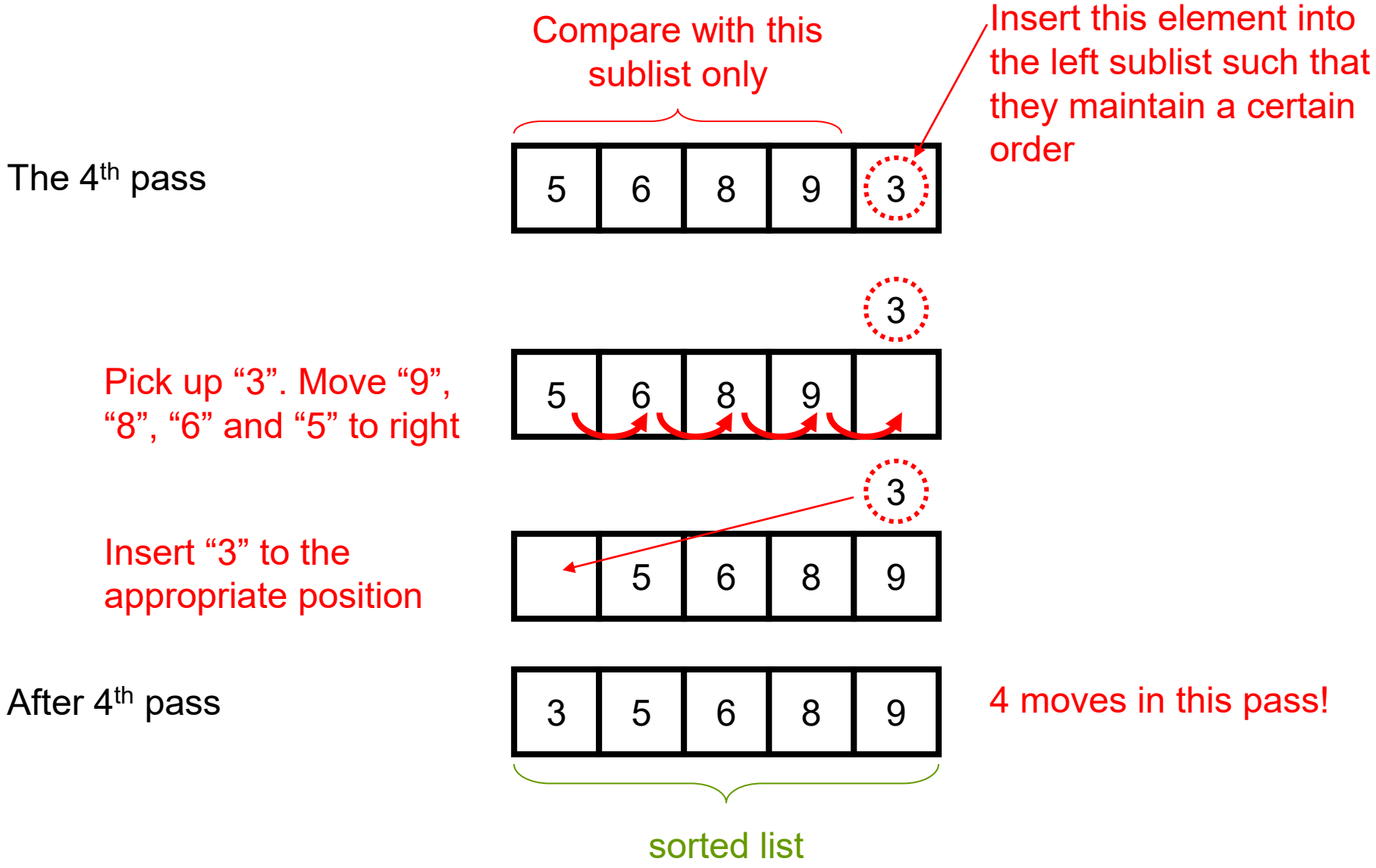


no move in this pass

Insertion Sort Example



Insertion Sort Example



Pseudo Code Review (before we introduce the pseudo code of Insertion sort)

- We need a language to express program development
  - English is too verbose and imprecise.
  - The target language, e.g. C/C++, requires too much details.
- Pseudo code resembles the target language in that
  - it is a sequence of steps (each step is precise and unambiguous)
  - it has similar control structure of C/C++
- Pseudo code is a kind of structured English for describing algorithms. It allows the designer to focus on the logic of the algorithm without being distracted by details of language syntax.

```
x = max{a, b, c}
```

Pseudo code

```
x = a;
if (b > x)  x = b;
if (c > x)  x = c;
```

C++ code

Insertion sort “detailed” pseudo code

- Basic operation: insert an element into a sorted list s.t. the final list is sorted.

```
Void INSERT_SORT(A)          // length [A] = n, A's index starts with 0
1   for (int i = 1; i < n; i++) {
2       int temp = data[i];
3       // element (data[i]) to be inserted
4       int j = i-1; //the last element in the sorted list
5       while(j >= 0 && data[j] > temp){
6           data[j+1] = data[j]; //movement operation
7           j = j-- ;}
8       data[j+1] = temp; }
```

https://yongdanielliang.github.io/animation/web/InsertionSortNew.html

Insertion sort animation game

How do you ”evaluate” an algorithm?

- E.g. there are other sorting algorithms too. Which of them is the “best”?
  - How to rank the algorithms?
  - Criteria
    - Correct?
    - **Fast?**
    - Low memory usage?

We will focus on running time analysis now

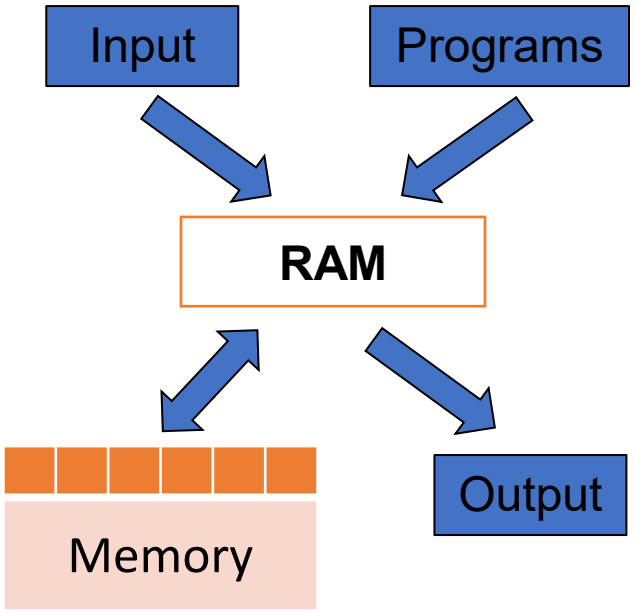
- Basic running time analysis examples
- Insertion sort’s running time

How to measure running time?

- Can we run two programs on computers and just report their actual running time?
  - Different hardware/OS can affect the running time
  - Different operators/people can run the program differently
- Thus, we must be able to measure the running time independent of the hardware, OS, and the users.
  - RAM model

RAM model

- Running time analysis using random-access machine (RAM) model
  - RAM: a generic one-processor instruction is executed one after another; no concurrent operations.



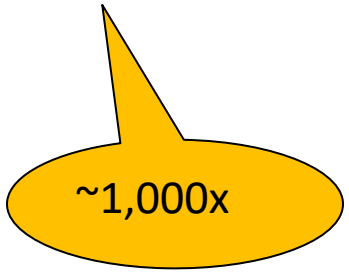
  - Each “simple” operation (+, *, -, /, ==, if, else, =(←)) takes exactly 1 step (most arithmetic operations)

Algorithms Analysis Example

- Find the sum of 1 + 2 + 3 + 4 + … + 998 + 999

- Method 1:
  - 1 + 2 = 3
  - 3 + 3 = 6
  - 6 + 4 = 10
  - …
  - 498,501 + 999 = 499,500

998 addition



- Method 2:
  - ((1 + 999) x 999) / 2
  - = 499,500

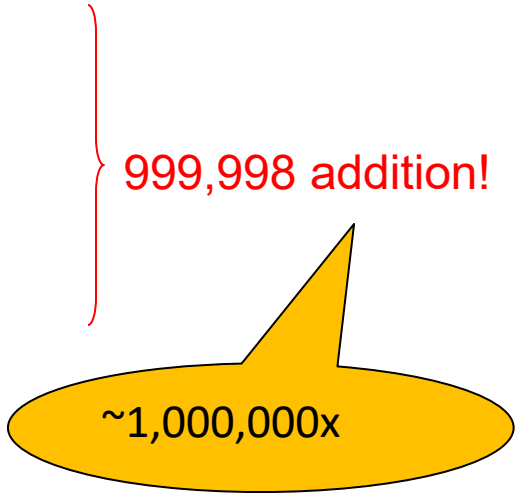
1 addition, 1 multiplication, 1 division

Algorithms Analysis Example

- Find the sum of 1 + 2 + 3 + 4 + … + 999,999

- Method 1:
  - 1 + 2 = 3
  - 3 + 3 = 6
  - 6 + 4 = 10
  - …
  - 498,998,500,001 + 999,999 = 499,999,500,000

- Method 2:
  - (1 + 999,999) x 999,999 / 2
  - = 499,999,500,000



Still 1 addition, 1 multiplication, 1 division! (independent of the input size)

Algorithms Analysis Example

Method 1:

```
int sumOfSeries(int n) {
    int i, sum = 0;
    for (i = 1; i < n; i++)
        sum += i;
    return sum;
}
```

n is the size of the elements.
For example, if n=10, the sum is 1+2+3+…+9.

n - 1 addition

Which one is better?

Method 2:

```
int sumOfSeries(int n) {
    return (1 + n) * n / 2;
}
```

1 addition, 1 multiplication, 1 division

An Example Program

```
#include <iostream>

int main(int argc, char *argv[]) {

   int i, n, sum = 0;

   cin >> n;

   for(i = 0; i < n; i++)
      sum += i;

   return 0;
}
```

Constant time, C_1

Constant time, C_2

Variable time, depends on n
$= C_3 \times n$

Constant time, C_4

Total execution time
$= C_1 + C_2 + C_3 \times n + C_4$
$\approx C_3 \times n$ (if n is very large)

Analysis

- The exact value of C_i is not important, but the order of magnitude is important
- Usually C_i is a very small number
- $n * C_i$ would be a very significant number if n is a very large number
- e.g. suppose C_i is 1ms
  - If n is 1, execution time is 1ms
  - If n is 10, execution time is 10ms
  - If n is 1 million, execution time is 1,000s
- C_i is machine dependent
- To simplify our analysis, simply count how many times each instruction is executed in the algorithm.

Count the No. of Operations

```
int i, n, sum = 0;
```

This instruction being executed once

```
cin >> n;
```

This instruction being executed once

```
for(i = 0; i < n; i++)
    sum += i;
```

This block being executed *n* times

Total execution
= 1 + 1 + *n*
= *n* + 2
≈ *n* (if *n* is very large)

Count the No. of Operations

Practice time: how many times is each code executed?

```
//Code A
sum += i;
```

This instruction being executed once

```
//Code B
for(i = 0; i < n; i++)
        sum += i;
```

This code being executed n times

```
//Code C
for(i = 0; i < n; i++)
    for(j = 0; j < n; j++)
        sum += i * j;
```

This code being executed n^2 times!

Count the total number of operations of each statement

Practice time: how many times is each statement executed?

```
//Code A
sum += i;                          1 time in total
```

Total running time: 1

```
//Code B
for(i = 0; i < n; i++)             ~n times in total
    sum += i;                      n times in total
```

Total running time:
n+n

```
//Code C
for(i = 0; i < n; i++)             ~n times in total
  for(j = 0; j < n; j++)           ~n^2 times in total
    sum += i * j;                  ~n^2 times in total
```

Total running time:
$n+n^2+n^2$

Insertion sort analysis

- Basic operation: insert an element into a sorted list s.t. the final list is sorted.

t_i is the number of running time for the inner loop for a given *i* (*i* is the index in the outer loop)

	INSERT_SORT(A) // length [A] = n	Cost	Times
1	for (int i = 1; i < n; i++) {	C_1	n
2	int temp = data[i];	C_2	n
3	// element to be inserted	0	
4	int j = i-1;	C_4	n
5	while (j >= 0 && data[j] > temp){	C_5	$\sum_{i=1\ to\ n-1} t_i$
6	data[j+1] = data[j];	C_6	$\sum_{i=1\ to\ n-1} t_i$
7	j = j-- ;}	C_7	$\sum_{i=1\ to\ n-1} t_i$
8	data[j+1] = temp;	C_8	n

Ci is the cost of the corresponding statement. Times: how many times each statement is executed

Insertion sort analysis

- Total running time (ignore all C_i)

$$T(n) = \text{n} + \text{n} + \text{n} + (\sum_{i=1}^{n-1} t_i) + (\sum_{i=1}^{n-1} t_i) + (\sum_{i=1}^{n-1} t_i) + \text{n}$$

- Minimum running time (best case: $t_i = 1$)

$$\begin{aligned} T(n) &= \text{n} + \text{n} + \text{n} + (\sum_{i=1}^{n-1} 1) + \text{n} \\ &= \text{n} + \text{n} + \text{n} + (n-1) + \text{n} \\ &= 5\text{n} - 1 \\ &= a \cdot n + b \end{aligned}$$

Question: give me an example of the best-case input

Insertion sort analysis

- Maximum running time (worst case: $t_i = ?$)

$$\sum_{i=1}^{n-1} t_i = \sum_{i=1}^{n-1} i = 1 + 2 + 3 + \cdots + n - 1 = \frac{n(n-1)}{2}$$

$$T(n) = \text{n} + \text{n} + \text{n} + (\frac{n(n-1)}{2}) + (\frac{(n)(n-1)}{2}) + (\frac{(n)(n-1)}{2}) + \text{n}$$

$$= a'n^2 + b'n + c'$$

Question: provide a worst-case input?

Insertion sort analysis

- Average case analysis ($t_i = \frac{i}{2}$)

$$\sum_{i=1}^{n-1} t_i = \sum_{i=1}^{n-1} \frac{i}{2}$$

Insertion sort analysis

- Running time analysis using random-access machine (RAM) model
  - RAM: a generic one-processor instruction is executed one after another; no concurrent operations
  - Each “simple” operation (+, *, -, /, ==, if, else, =(←)) takes exactly 1 step
  - Loops and subroutines are not simple operations but depends on the size of input data & the contents of a subroutine.

    “sort”, “matrix multiplication”, “length of an array”
  - Each memory access takes 1 step
  - Now, RAM model: $C_i = 1$

Complexity Analysis

- Space complexity: O(1)
- the temp. variable is used to **hold** the element that going to be inserted into the sublist

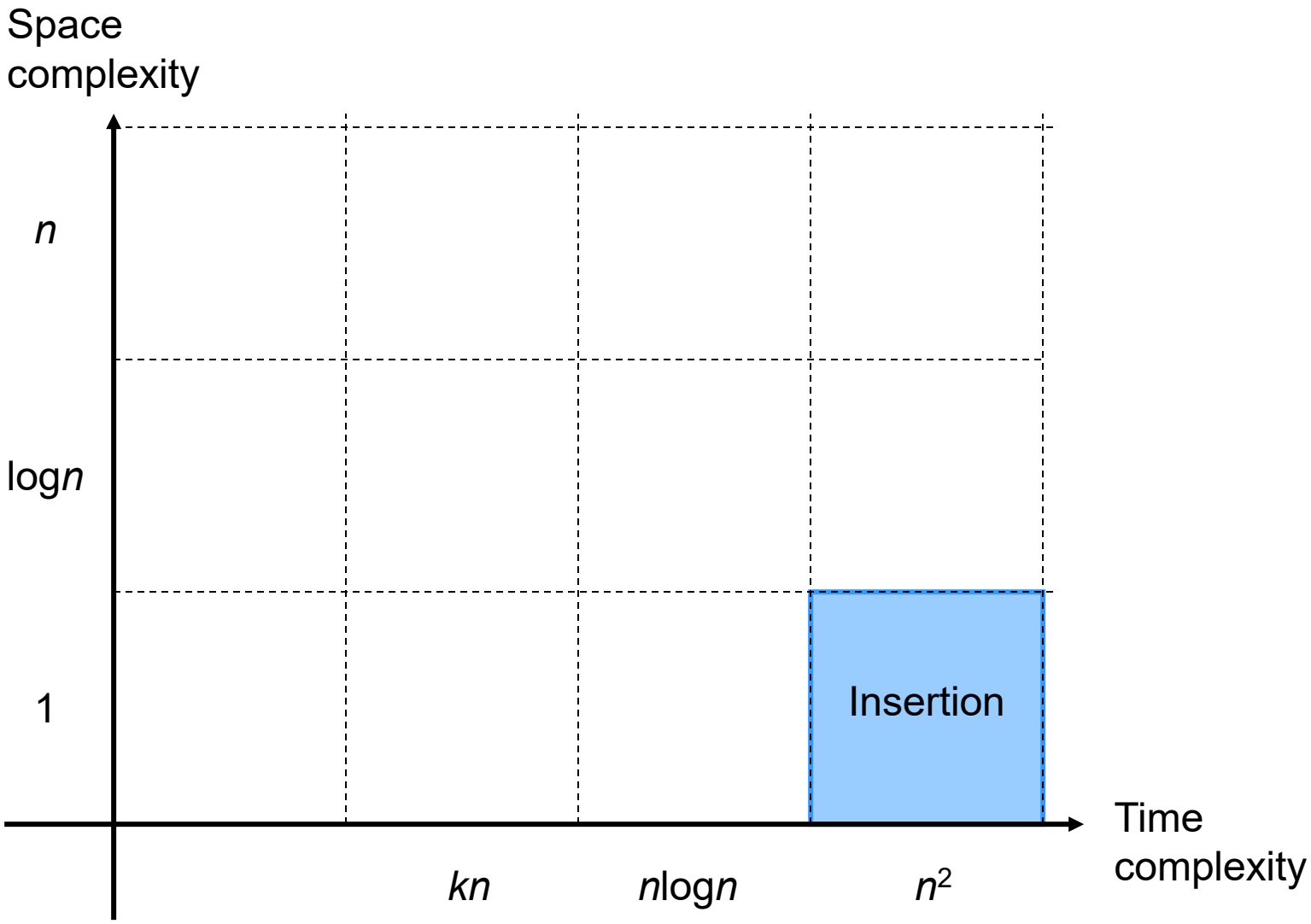
Big O notation will be introduced shortly. At this stage, think of big O as the fastest growing item in the running time equation (the dominate term/bottle neck)

Complexity Analysis

- The best case: O(n)
  - The list is already sorted; scan it once!
- The worst case: O(n^2)
  - *n-1* items to be inserted
  - At most *i* comparisons at *i-th* insertion
  - The total no. of comparisons = $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$
- The average case: O(n^2)
  - Half the number of comparisons

- Because of the simplicity of insertion sort, it is the fastest sorting method when the number of elements *N* is small, e.g. *N* < 10.

Summary


Space
complexity
n
logn
1
Insertion
Time
complexity
kn
nlogn
n2

Analysis of Algorithms

- Often several different algorithms are available to solve the same problem. These algorithms may not run with same efficiency.
  - May be impractical for large input size
  - May run extremely slow for particular inputs
- We want to know the efficiency and complexity of algorithms so as to compare them and make a wise choice.
- The complexity growth rate is far more important than the actual execution time during analysis.

Running time analysis

- Worst-case and average-case analysis
  - 1. The longest running time for any input of size n: the worst case

    E.g., 5 3 2 1 0 for insertion sort
  - 2. The upper bound on the running time for any input
  - 3. The worst case happens often

    E.g., database search: fail to find a match
  - 4. The average case is often roughly as bad as the worst case

    E.g., insertion sort, roughly $\begin{cases} half\ elements \leq key \\ half\ elements > key \end{cases}$, $t_j = \frac{j}{2}$

Running time analysis

- Simplifications/ approximations

n	$\frac{3}{2}n^2$	$\frac{3}{2}n^2 + \frac{7}{2}n - 4$	% difference
10	150	181	17% ($\frac{181-150}{180}$)
50	3,750	3,921	4.4%
100	15,000	15,436	2.3%
500	375,000	376,746	0.5%

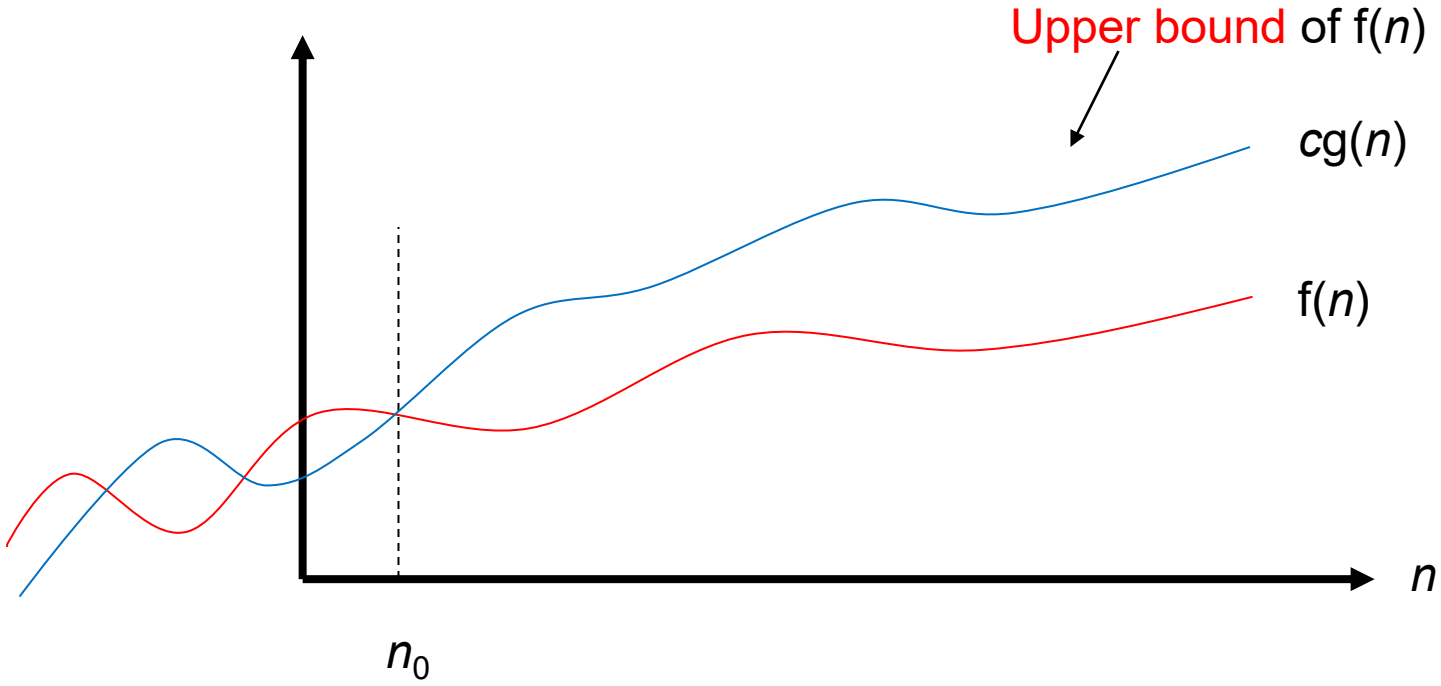
Highest-order term finally dominates the output

Asymptotic Complexity

- Asymptotic complexity is a way of expressing the main component of the cost of an algorithm.
- For example, when analyzing some algorithm, one might find that the time (or the number of steps) it takes to complete a problem of size n is given by
  $T(n) = 4n^2 - 2n + 2$
- If we ignore constants (which makes sense because those depend on the particular hardware the program is run on) and slower growing terms (i.e. 2n), we could say T(n) grows at the order of n^2 and write:
  $T(n) = O(n^2)$
- The letter O is used because the rate of growth of a function is also called its *Order*. Basically, it tells you how fast a function grows or declines.

Asymptotic Notation O

- Big-O notation defines an upper bound of an algorithm's running time.
- We say that a function f(n) is of the order of g(n), iff there exists constant $c > 0$ and n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$ (this definition is not required)
- In other words, f(n) is at most a constant times of g(n) for sufficiently large of values of n
- Using Big-O notation: $f(n) = O(g(n))$



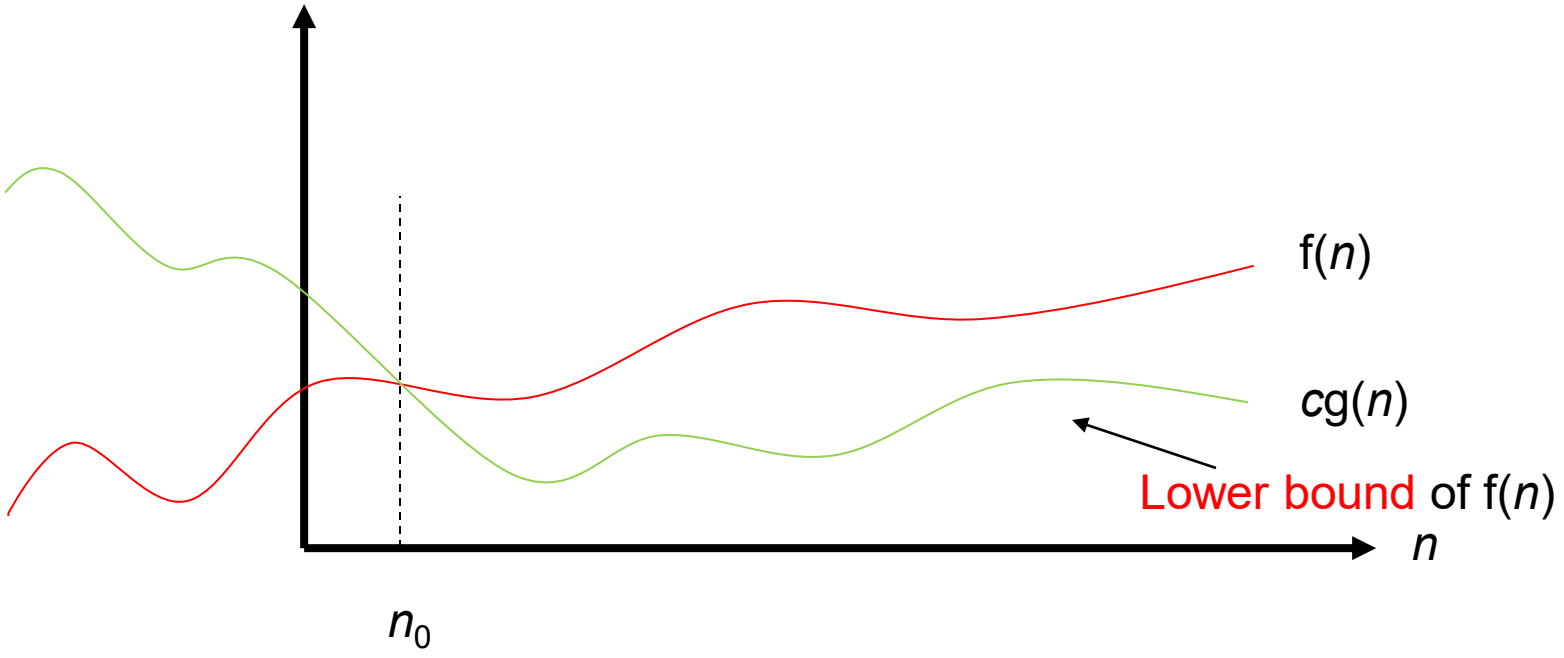
e.g. The time complexity of insertion sort is $O(n^2)$.

If $f(n)=5n^2+3$, we can say $f(n)=O(n^2)$

If $f(n)=100n+5$,
then $f(n)=O(n)$, $f(n)=O(100n)$, $f(n)=O(n^2)$, etc.

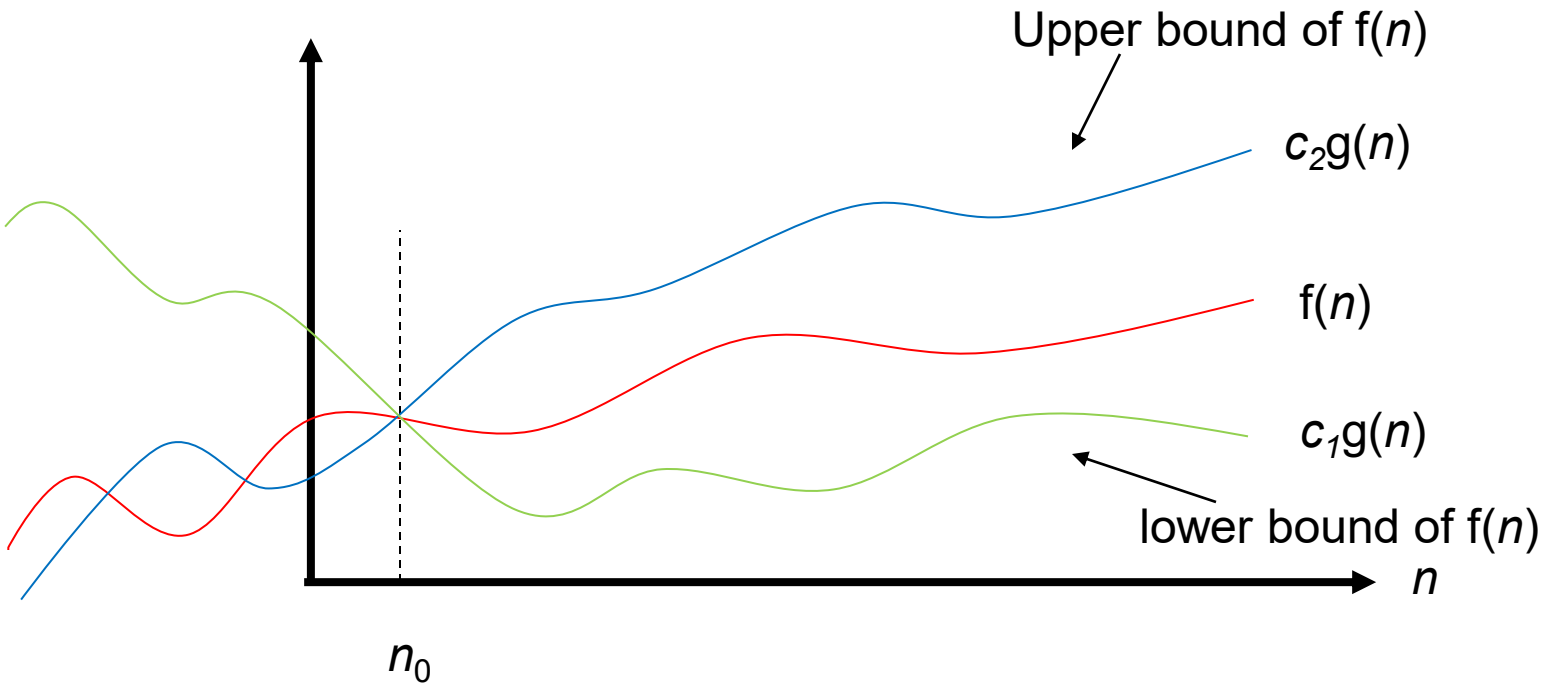
Asymptotic Notation Ω

- Big-Omega notation defines a lower bound of an algorithm's running time.
- f(n) = Ω(g(n)) iff there exists constant $c > 0$ and n_0 such that f(n) ≥ cg(n) for all n ≥ n_0 (I won't test you how to prove this)



Asymptotic Notation Θ

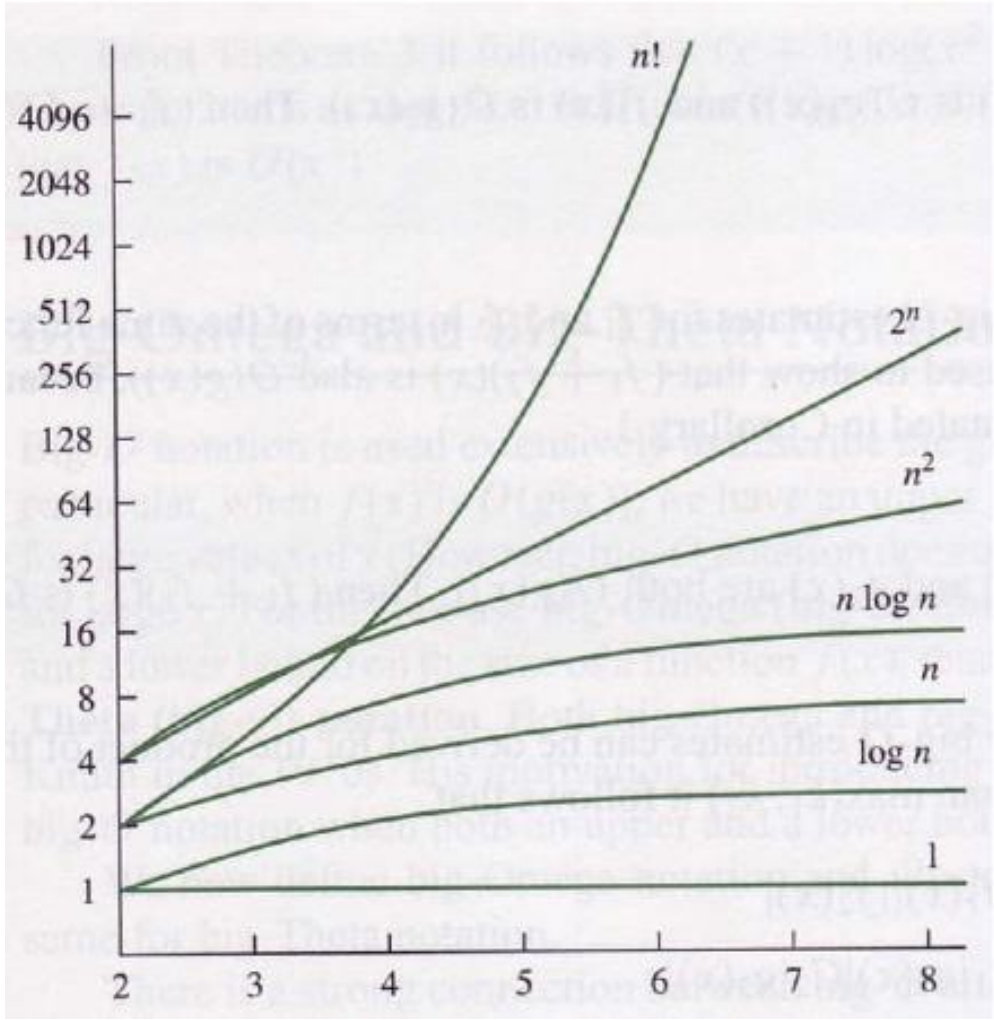
- Big-Theta notation defines an exact bound of an algorithm's running time.
- f(n) = Θ(g(n)) iff there exists constant $c_1 > 0$, $c_2 > 0$ and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n, $n \geq n_0$ (the proof is not required)



Important Complexity Classes

- $O(1)$: Constant time
- $O(\log_2 n)$: Logarithmic time
- $O(n)$: Linear time
- $O(n \log_2 n)$: Log-linear time
- $O(n^2)$: Quadratic time
- $O(n^3)$: Cubic time
- $O(n^k)$: Polynomial time
- $O(2^n)$: Exponential time

Important Complexity Classes



Factorial time

Exponential time

Quadratic time

Log-linear time

Linear time

Logarithmic time

Constant time

Increasing complexity

Becomes not feasible when *n* grows larger

In-class exercise: write the time complexity of method 1 and method 2 using big O notation

Method 1:

```
int sumOfSeries(int m) {
    int i, sum = 0;
    for (i = 1; i < m; i++)
        sum += i^2;
    return sum;
}
```

Method 2:

```
int sum(int n) {
    return (1 + n) * n / 2;
}
```

EE2331 Data Structures and Algorithms

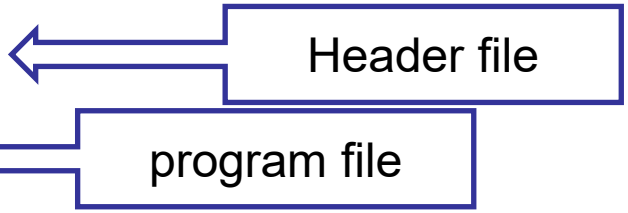
C++ Programming Review

Outline

In the lecture, we won't cover each page but these are good reference for you. The covered parts have gray background in the title.

- Standard Libraries
- Basic Data Types
- Arithmetic, Bitwise, Logical Operators
- Control Structures
- Pointers
- Arrays
- Composite Structures
- Parameter Passing in Functions
- Standard I/O
- Pseudo Code
- Suggestion for Good Programming Practice

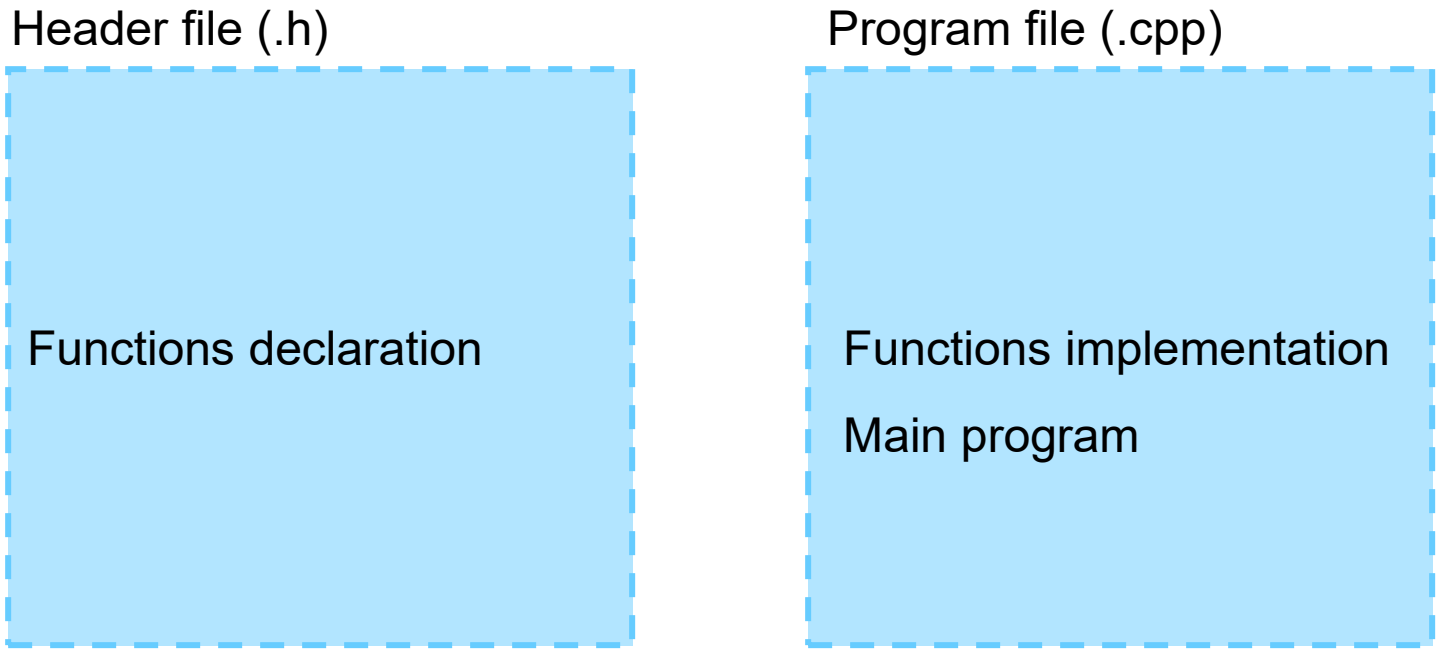
Let us start from an example (1)




```
#include <cstdio>

int main()
{
    int A, B, C;                                //local variable declaration

    printf("Enter the numbers A, B and C: "); //output function
    scanf("%d %d %d", &A, &B, &C);              //take inputs from standard input

    if (A >= B && A >= C)                       //logic operator
        printf("%d is the largest number.", A);

    if (B >= A && B >= C)
        printf("%d is the largest number.", B);

    if (C >= A && C >= B)
        printf("%d is the largest number.", C);

    return 0;                                   //return to OS (0=successful completion)
}
```

What is the purpose of this program?

C++ File Structure



A header file commonly contains forward declarations of subroutines. Programmers who wish to declare functions in more than one source file can place such declaration in a single header file, which other code can then include whenever the header contents are required.

Example 2 (self-defined header file)

```cpp
#include <cstdio>

/*
 * Demonstrate forward declaration of a function.
 */

int main(int argc, char** argv) {

    sayHello();
    return(0);
}

void sayHello() {
  printf("Welcome to EE2331!\n");
}
```

Can the above code compile successfully? If not, how to fix?

```
hello.cpp:11:5: error: use of undeclared identifier 'sayHello'
    sayHello();
    ^
1 error generated.
```

```cpp
#include <cstdio>
#include "hello.h"

/*
 * Demonstrate forward declaration of a function.
 */

int main(int argc, char** argv) {

    sayHello();
    return(0);
}

void sayHello() {
  printf("Welcome to EE2331!\n");
}
```

hello.cpp (END)

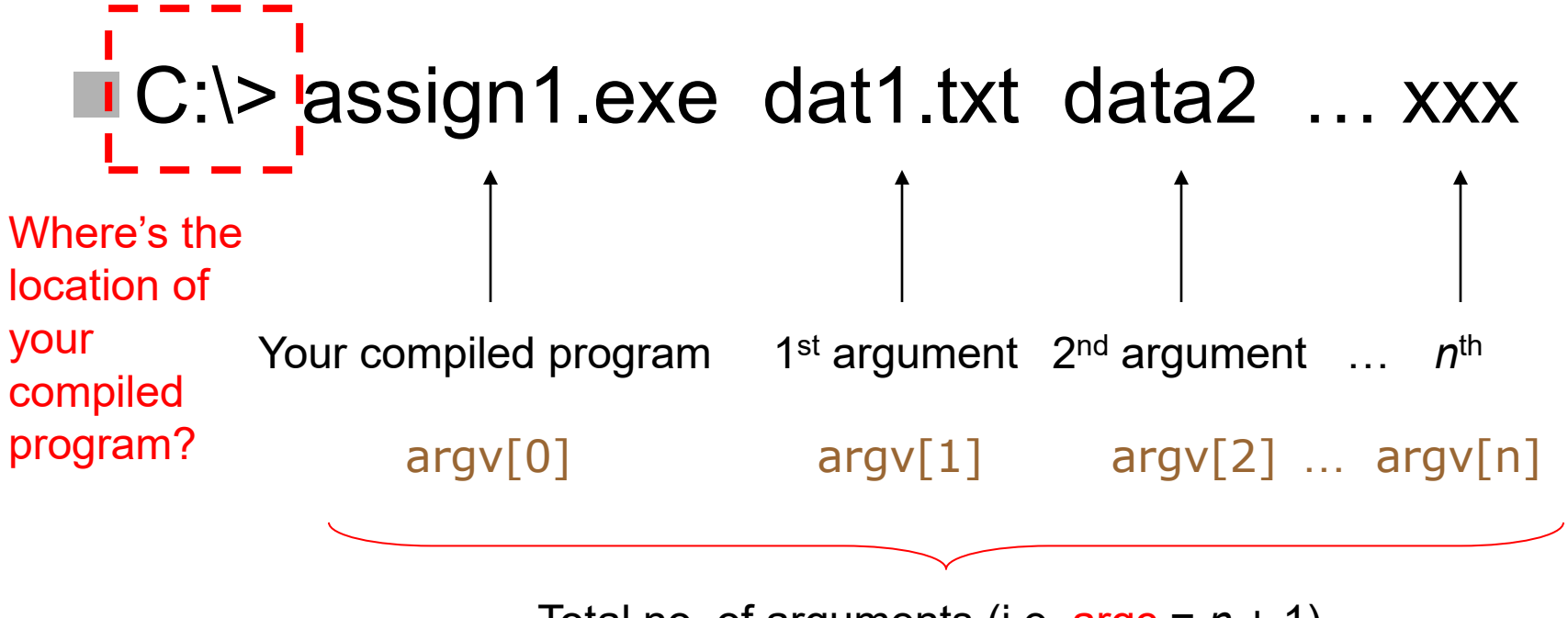
```cpp
// forward declaration
void sayHello();
```

hello.h (END)

Main Function

- There are two declarations of main that must be allowed:
  - int main() // without arguments
  - int main(int argc, char** argv) // with arguments
- The return type of main must be int.
  - Return zero to indicate success and non-zero to indicate failure.
  - You are not required to explicitly write a return statement in main(). If you let main() return without an explicit return statement, it's the same as if you had written return 0;.
    - int main() { } // equivalent to the next line
    - int main() { return 0; }
  - There are two macros, **EXIT_SUCCESS** and **EXIT_FAILURE**, defined in <cstdlib> that can also be returned from main() to indicate success and failure, respectively.

Command Line Arguments




```
int main(int argc, char *argv[]) {
    ...
}
```

argc: count

argv: value

Example 3

```
#include <cstdio>

int main(int argc, char* argv[])
{
  if(argc!=3)
    {
      printf("Enter two integers\n!");
      return 0;
    }

  for (int i = 0; i < argc; i++)
    printf("argv[%d] = %s\n", i, argv[i]);

  return 0;
}
```

A useful method to remind your users of the input format!

After I compile this program, I got the executable **program ex3.**
Show the output of the following three commands:

```
./ex3
```

Enter two integers

```
./ex3 4 9 10
```

Enter two integers

```
./ex3 4 9
```

```
argv[0] = ./ex3
argv[1] = 4
argv[2] = 9
```

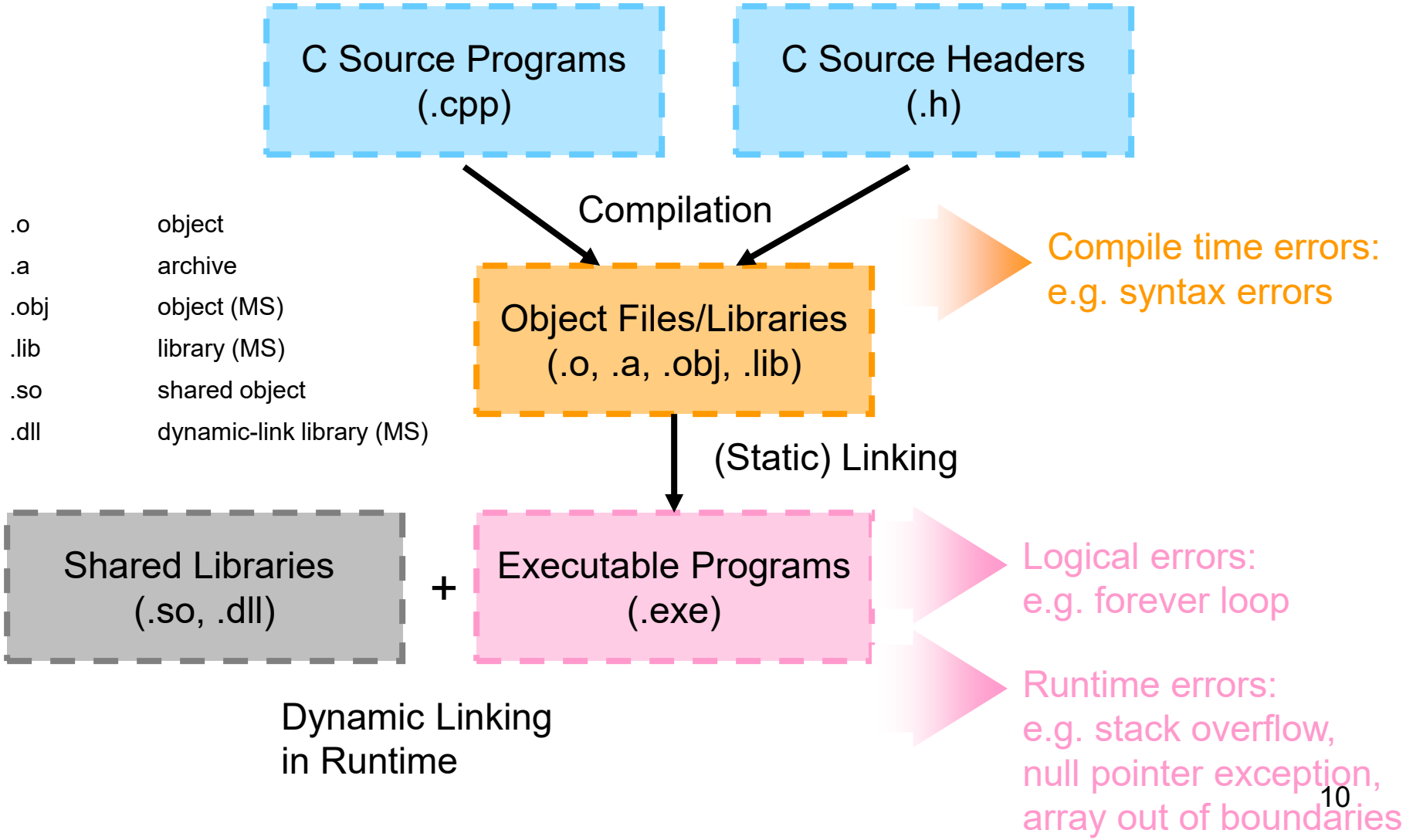
Command Line Arguments

```
int main(int argc, char *argv[]) {
    printf("argc = %d\n", argc);
    for (int i = 0; i < argc; i++)
        printf("argv[%d] = %s\n", i, argv[i]);}
```

```
>ex1_2.exe  123  abc
```

```
argc = 3
argv[0] = ex1_2.exe          //name of the program
argv[1] = 123                //string, not integer
argv[2] = abc
```

The Building Process



Common Standard Library Header

- <cstdio>
  - Standard I/O facilities: printf(), scanf(), getchar(), fopen(), fclose(), etc
- <cstdlib>
  - Standard utility functions: malloc(), free(), rand(), etc
- <cstring>
  - String functions: strcpy(), strcmp(), memset(), etc
- <iostream>
  - Perform both input and output operations with the stream objects: **cin** and **cout**

Comments

```
/* Block comment 1 */


/*
 * Block comment 2
 */
```

```
// Line comment
```

Primitive Data Types in C++

Data type	Size (byte)	Interpretation/representation	Range of values
bool	1	Boolean (not available in C)	false or true
char	1	signed number (2's complement)	-128 to 127
unsigned char		unsigned number	0 to 255
int	4	signed number (2's complement)	-2^{31} to $2^{31}-1$
unsigned int		unsigned number	0 to $2^{32}-1$
short	2	signed number (2's complement)	-2^{15} to $2^{15}-1$
unsigned short		unsigned number	0 to $2^{16}-1$
long	4	signed number (2's complement)	-2^{31} to $2^{31}-1$
unsigned long		unsigned number	0 to $2^{32}-1$
long long	8	signed number (2's complement)	-2^{63} to $2^{63}-1$
unsigned long long		unsigned number	0 to $2^{64}-1$
float	4	IEEE 32-bit floating point number	$\pm 1.4\times10^{-45}$ to $\pm 3.4\times10^{38}$
double	8	IEEE 64-bit floating point number	$\pm 5\times10^{-324}$ to $\pm 1.798\times10^{308}$
pointer	**4**	**memory address**	**0 to $2^{32}-1$**

pointer's size is 4 bytes for 32-bit machine; 8 bytes for 64-bit machine

Operators in C++

Operator	Symbol	Description
Assignment	=	
Arithmetic	+, -, *, /, %	
Increment, decrement	++, --	
Unary minus	-	
Comparison	==, !=, <, <=, >, >=	
Logical	!, &&, ||	
Bitwise	~, &, |, ^, <<, >>	
insertion, extraction	cout << s cin >> i	insertion to an output stream extraction from an input stream
Member and pointer	x[i]	subscript (x is an array or a pointer)
	*x	indirection, dereference (x is a pointer)
	&x	reference (address of x)
	x->y	structure dereference (x is a pointer to object/struct; y is a member of the object/struct pointed to by x)
	x.y	structure reference (x is an object or struct; y is a member of x)

Use of Variables

- Declaration
  - Given an identifier (variable name), you specify the data type of it and hence implicitly reserve the required memory space.
- Initialization
  - Variables should be initialized before being used.

```
int a;
cout << a;       // prints dummy value
```

Arithmetic Operators

- Addition

```
int a, b, c;
a = 1;
b = 2;
c = a + b;
printf("%d\n", c);
```

- Mind the overflow problem

```
int a, b, c;
a = b = 2147483647;  //the largest value of signed int
c = a + b;
printf("%d\n", c);
```

Arithmetic Operators

- Subtraction

```
int a, b, c;
a = 1;
b = 2;
c = a - b;
printf("%d\n", c);
```

- Mind the underflow problem

```
int a, b, c;
a = -2147483648;  //the smallest value of signed int
b = 2147483647;   //the largest value of signed int
c = a – b;
printf("%d\n", c);
```

Arithmetic Operators

- Division

```
int a, b, c;
a = 5;
b = 2;
c = a / b;
printf("%d\n", c);          // output is 2
```

  - Integer truncation occurs

Arithmetic Operators

- Remainder (Modulus Operator)

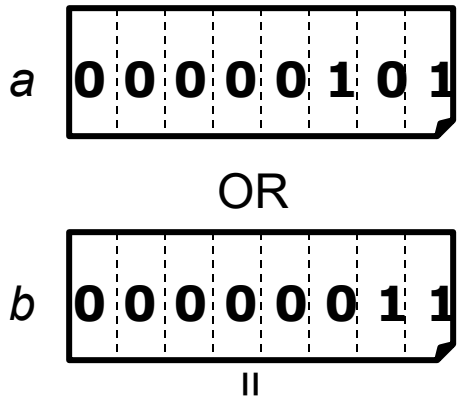
```
int a, b, c;
a = 5;
b = 2;
c = a % b;
printf("%d\n", c);
```

  - When to use it?
    - Generate periodic values
    - To wrap around the array index (in Queue)
    - To determine the hash key (in Hash table)

Bitwise & Logical Operators

- Bitwise OR

```
int a, b, c;
a = 5;
b = 3;
c = a | b;
printf("%d\n", c);
```



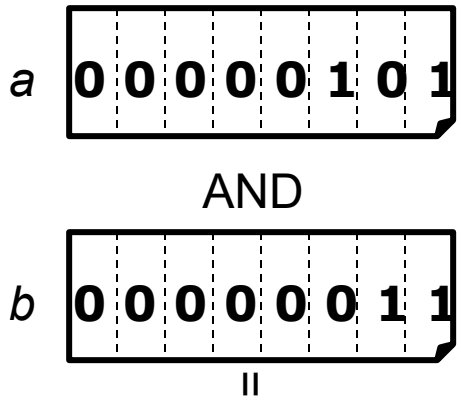
- Logical OR

```
int a, b, c;
a = 5;
b = 3;
c = a || b;
printf("%d\n", c);
```

Bitwise & Logical Operators

- Bitwise AND

```
int a, b, c;
a = 5;
b = 3;
c = a & b;
printf("%d\n", c);
```



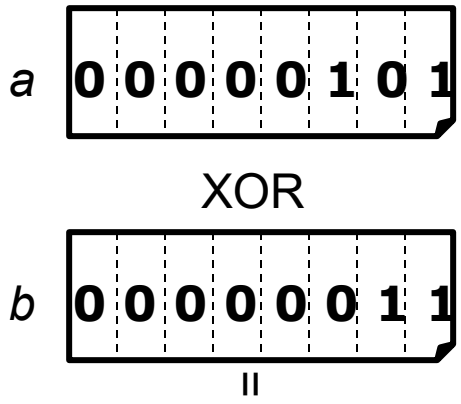
- Logical AND

```
int a, b, c;
a = 5;
b = 3;
c = a && b;
printf("%d\n", c);
```

Bitwise Operators

- Exclusive OR

```
int a, b, c;
a = 5;
b = 3;
c = a ^ b;
printf("%d\n", c);
```

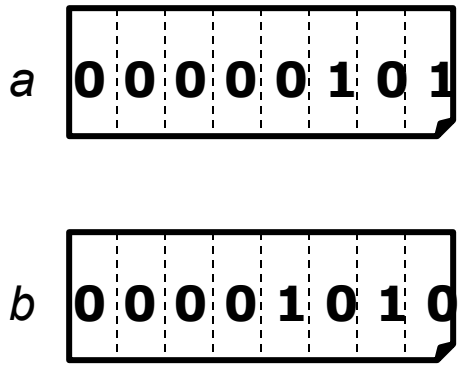


- When to use it?
  - Interchange two variables

Bitwise Operators

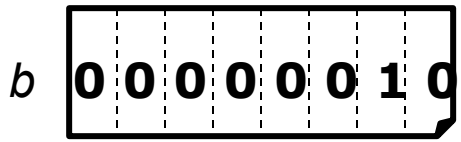
- Left Shift (x2)

```
int a, b;
a = 5;
b = a << 1;
printf("%d\n", b);
```



- Right Shift (/2)

```
int a, b;
a = 5;
b = a >> 1;
printf("%d\n", b);
```



Variable Assignments

- Example 1

```
int a, b, c;
a = b = c = 5;
printf("%d\n", a);
```

- Example 2

```
int a = 5, b = 5, c = 5;
a = b == c;
printf("%d\n", a);
```

What are the outputs of the two examples?

Typecasting

- Example 1 - Implicit

```
int a;
float b = 10.5;
a = b;                  // precision loss with warning
printf("%d %f\n", a, b);    // 10 10.5
```

- Example 2 - Explicit

```
int a;
float b = 10.5;
a = (int) b;            // still precision loss but NO warning
printf("%d %f\n", a, b);    // 10 10.5
```

Typecasting

- Example 3

```
int a = 3;
int b = 2;
int c = 4;
cout  << a / b * c << endl;  // output is 4 !!
cout  << a * c / b << endl;  // output is 6
```

- The resultant type of an arithmetic operation will be promoted to the type of the operators with larger precision.
  - int / int → int
  - float / int → float

Control Structures

If-then-else

- ? : (ternary operator)
  - equivalent to if-then-else
  - *expression* ? *true instruction* : *false instruction*;

```
if (a < b)
    min = a;
else
    min = b;
```

```
min = a < b ? a : b;
```

For-Loop and While-Loop

- for-loop and while-loop are interchangeable

```
for (initialization; loop_test; loop_counting) {
        //loop-body
}
```

```
initialization;
while (loop_test) {
        //loop-body
        loop_counting;
}
```

Loop head is executed one more time than loop body. But we won't care about this slight difference.

Jump Statements

- Jump statements allow the early termination of loops

- These cause unconditional branches
  - goto is bad practice and will not be dealt with
  - break will exit the inner most loop
  - continue will force the next iteration
  - return will return to the calling function
  - exit will quit the program

Breaking Out Loops Early

```
for (i = 0; i < n; i++) {
    …
    if (…) break;          //to break out the for-loop
    …
}
```

```
while (…) {
    …
    if (…) continue;       //to skip the rest part of current iteration,
                           //and continue for next iteration
    …
}
```

Bad Styles of Loop

```
// DON'T use != (Not equal) to test the end of a range
for (i = 1; i != n; i++) {
   //loop body
}
// How does the loop behave if n happens to be zero or negative?
```

```
// DON'T modify the value of the loop-counter inside the loop body of a for-loop.
for (i = 1; i <= n; i++) {
   //main body of the loop
   if (testCondition)
      i = i + displacement;

   //i++ is executed before going back to top of the loop
}
```

Breaking Out Functions Early

```
void func(…) {
    …
    if (…) return;          //to break out the function
    …
}
```

```
int func(…) {
    …
    if (…) return 0;        //to break out the function, and
                            //return a value to the calling function
    …
}
```

Breaking Out Programs Early

```
void func(…) {
    …
    if (…) exit(0);          //to terminate the program, and
                             //return normal exit value 0 to operating
    …                        //system!
}
```

```
int main(…) {
    …
    if (…) exit(1);          //to terminate the program, and
                             //return abnormal exit value 1 to
    …                        //operating system!
    return 0;                //normal completion of the program
}
```

Loop Design

- Find the maximum value in an array of integers.
- In-class exercise: any mistake in this program?

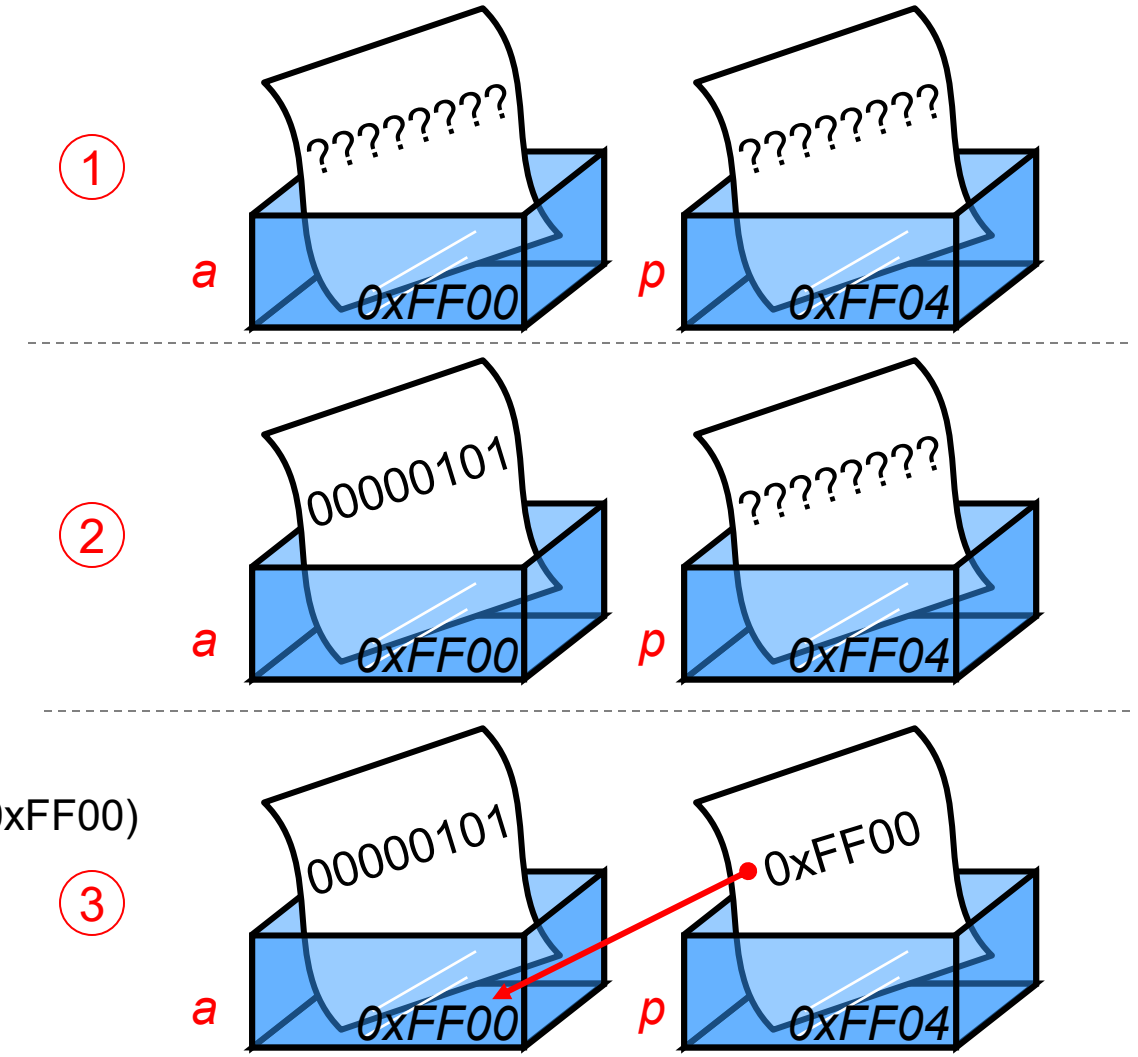
```
int max(int a[], int n) {          //n = no. of elements in a[]
   int m = 0;                      // variable to store the max value
   for (int i = 0; i < n; i++)
      if (a[i] > m)
         m = a[i];
   return m;
}
```

Pointers and Arrays

Pointers

```
1 int a, *p;
2 a = 5;
3 p = &a;
```

Note: The actual size of integers and pointers are 4-byte long



a: value of *a* (i.e. 5)

&a: address of *a* (i.e. 0xFF00)

*a: ?

p: value of *p* == address of *a* (i.e. 0xFF00)

&p: address of *p* (i.e. 0xFF04)

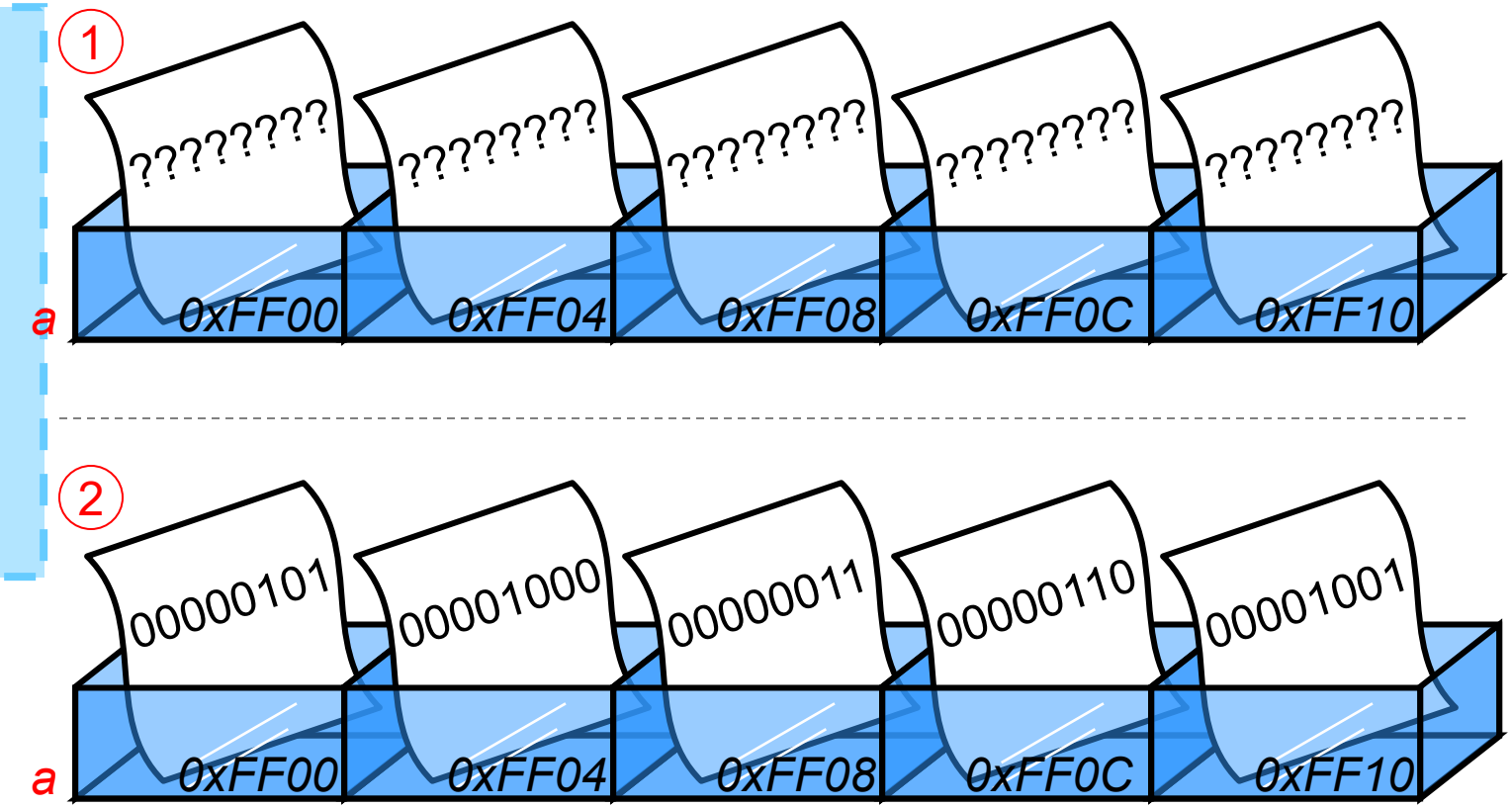
*p: value pointed by p (i.e. 5)

Pointers Example

```
int x = 1, y = 2;
int *a, *b, *c;

a = &x;
b = &y;
printf("%d %d %d %d\n", x, y, *a, *b);

c = a;              // swap a with b
a = b;
b = c;
printf("%d %d %d %d\n", x, y, *a, *b);
```

Creation of Array

```
1  int a[5];
   a[0] = 5;
   a[1] = 8;
   a[2] = 3;
   a[3] = 6;
2  a[4] = 9;
```



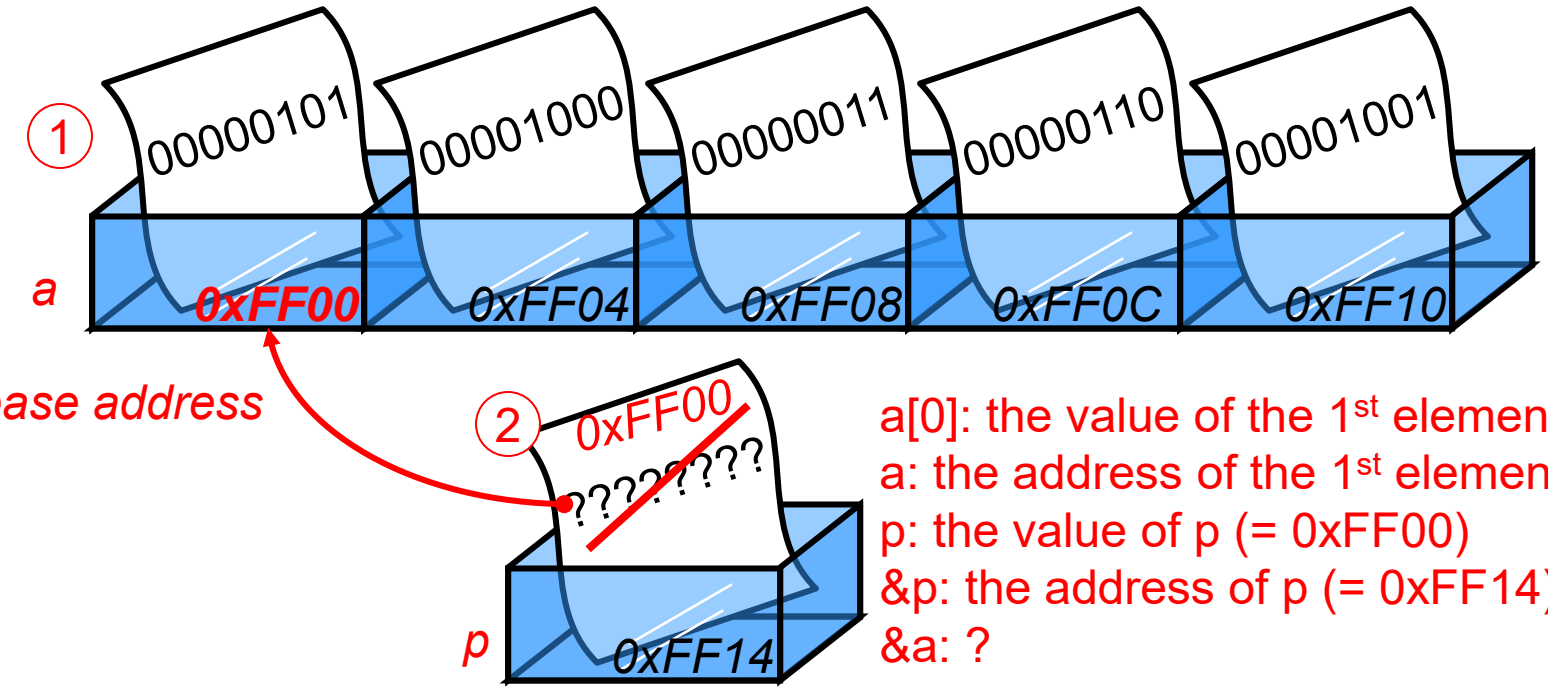
Note: the elements of integer array should be 4-byte long.

Base Address of Arrays

Initialization, set size implicitly

```
1  int a[] = {5, 8, 3, 6, 9};
   int *p;
2  p = a;  //why not p = &a; ??
```

The array variable 'a' is interpreted as a pointer pointing to the first element (base address) of the array.

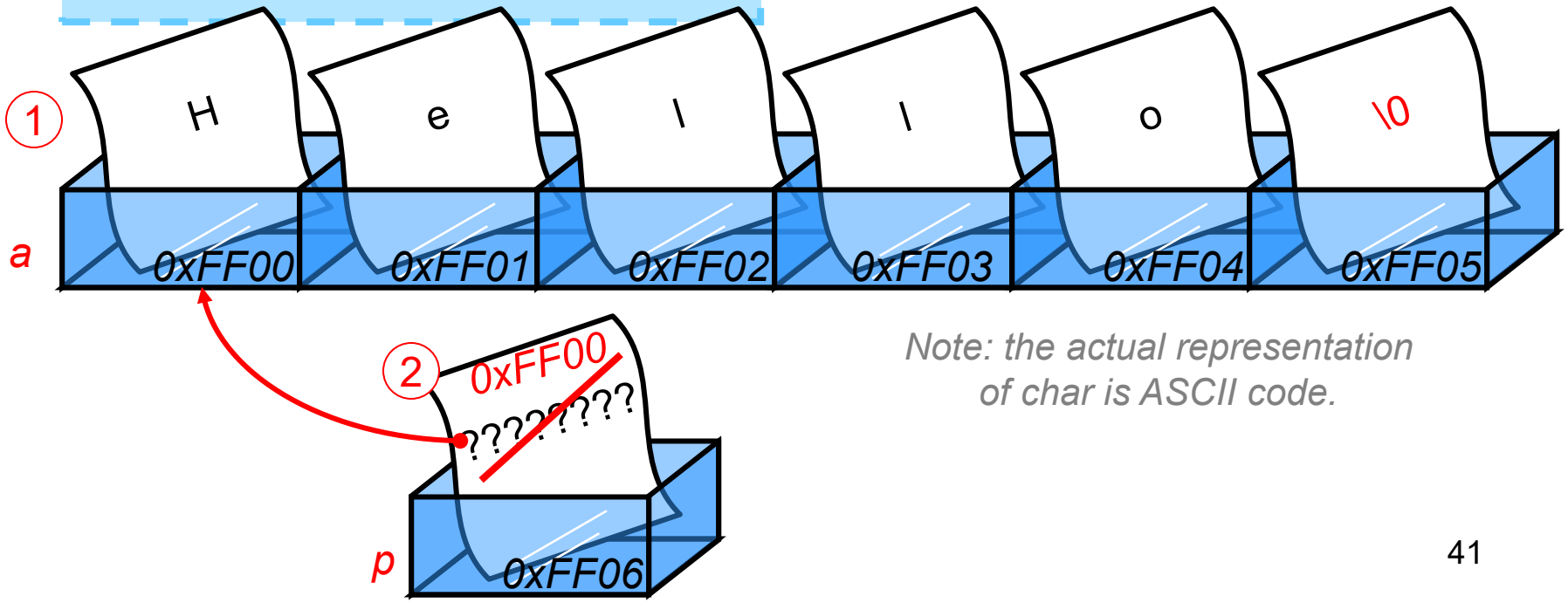


a[0]: the value of the 1st element (= 5)
a: the address of the 1st element (= 0xFF00)
p: the value of p (= 0xFF00)
&p: the address of p (= 0xFF14)
&a: ?

C-String (Character Array)

```
1  char a[] = "Hello";
   char *p;
2  p = a;
   printf("%d\n", sizeof(a));
```

Null character ('\0') filled at the end of character array (string)

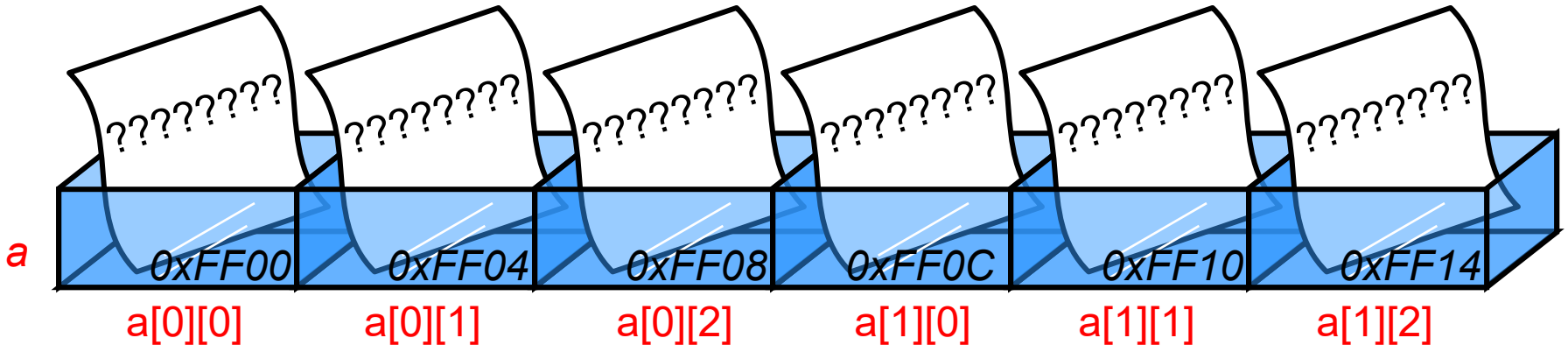


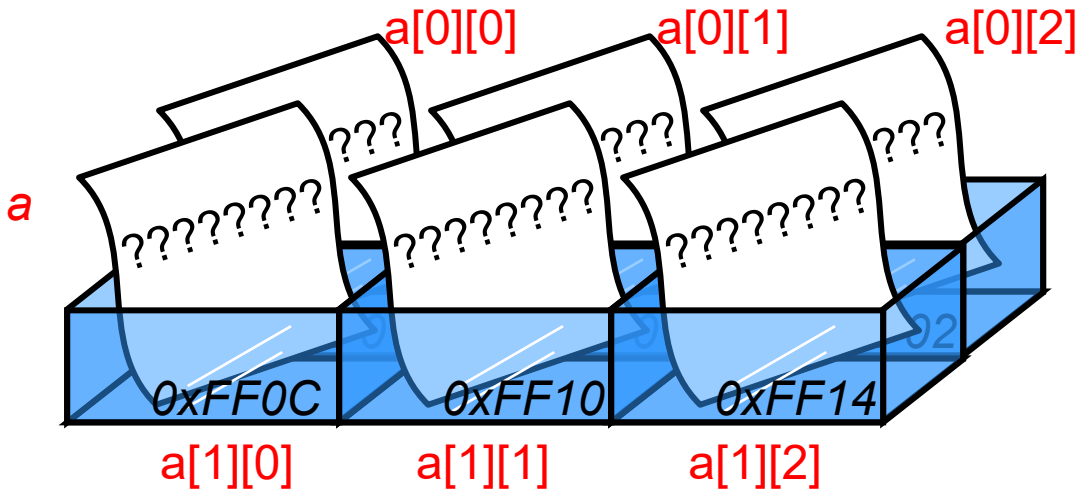
Note: the actual representation of char is ASCII code.

2D Arrays

```
int a[2][3]; //2 rows, 3 columns
```

Multi-dimensional arrays are mapped to the linear address space of the computer system.





In C/C++, elements of a multi-dimensional array are arranged in row-major order.

Size of Array

- The size of array is fixed and predetermined
- Cannot declare an array with variable size

```
#define n 10     //n is a macro
int i, a[n];     //ok, n is substituted by 10 during compilation
for (i = 0; i < n; i++)
    a[i] = i;
```

```
int n=100;       // n is a variable
int i, a[n];     // compilation error, a commonly seen mistake
for (i = 0; i < n; i++)
    a[i] = i;
```

Boundaries of Array

- C/C++ will not check the boundaries of array

```
int a[10];
a[11] = 0;        //allow to run (dangerous!)
                  //but result is unpredictable!
```

- It is the responsibility of programmers to ensure not going out the boundaries

```
int a[10];
int i = 11;
if (i >= 0 && i < 10) a[i] = …;  //boundaries checking
```

Composite Structures

Typedef

- To rename a type to a new name

```
int func(int x) {
    return x*x;
}
int main(…) {
    int a, b;
    a = 1;
    b = func(a);
    …
}
```

equivalent

```
typedef int NUM;
NUM func(NUM x) {
    return x*x;
}
int main(…) {
    NUM a, b;
    a = 1;
    b = func(a);
    …
}
```

Structures

- To define a composite structure

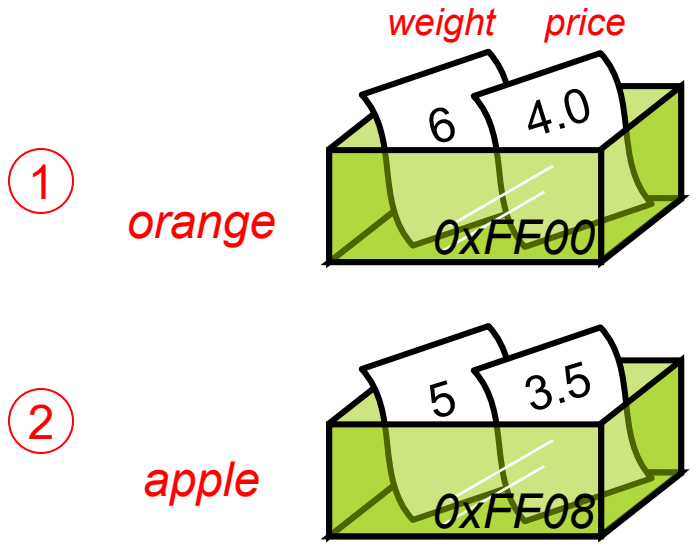
```
struct name{
    data_type1 member1;
    data_type2 member2;
    …
};
```

- To refer to this structure, use

```
struct name          // C
name                 // C++
```

Structure

```
struct Product{
    int weight;
    float price;
};
```




```
int main(…) {
1   Product orange = {6, 4.0};
    Product apple;
    apple.weight = 5;
2   apple.price = 3.5;
    printf("%d\n", apple.weight);
    …
}
```

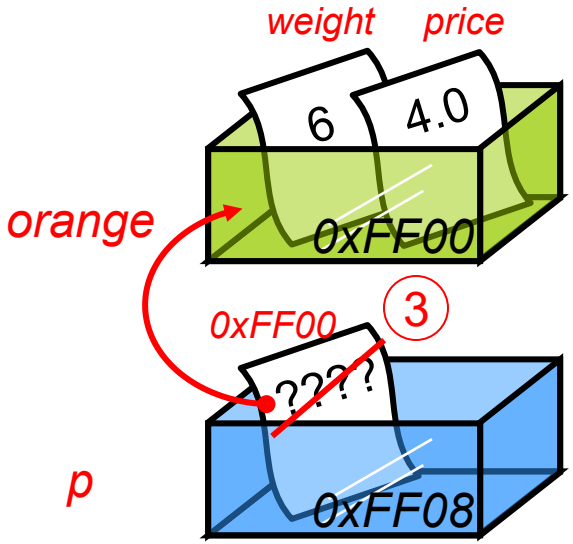
A structure can be initialized by using {}

Or use the . (dot) operator to access the member of a structure

Pointer to Structure

```
struct Product{
    int weight;
    float price;
};
```

```
int main(…) {
1 Product orange = {6, 4.0};
2 Product *p;
3 p = &orange;
    printf("%d\n", p->weight);
    printf("%d\n", (*p).weight);
    …
}
```



Use the arrow -> operator to access the member of pointer-to-structure

Parameter Passing in Functions

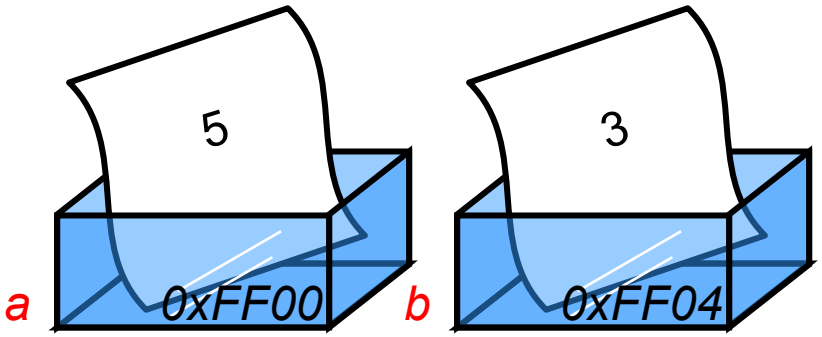
Parameter Passing in Functions

- Pass by value
  - Involve copying the value of parameters
- Pass by pointer
  - Just pass the address of the parameters, without copying the value of them
  - Usually used in passing large-size data structures, e.g. arrays, structures, objects, lists, etc
- Pass by reference
  - C++ reference is similar to pass-by-pointer but without the hassles of pointers' (&)reference/ (*)dereference syntax
  - You can specify a formal parameter in the function signature as a reference parameter

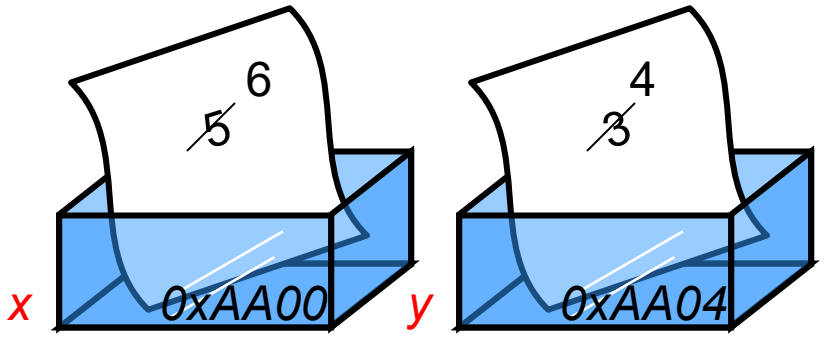
Pass by Value

```
void plus_one(int x, int y) {
    x++;    y++;
}
```

```
int a = 5, b = 3;
plus_one(a, b);
```



The values of *a*, *b* have not been modified



A new set of variables is **duplicated** in function *plus_one*

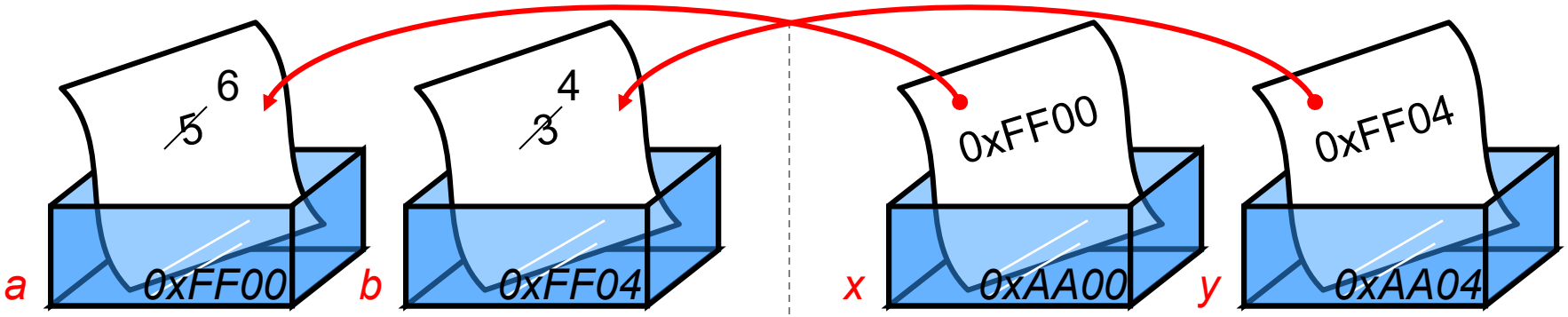
Pass by Pointer

```
void plus_one(int *x, int *y) {
    (*x)++;    (*y)++;
}
```

pointers

```
int a = 5, b = 3;
plus_one(&a, &b);
```

addresses



The values of *a*, *b* have been modified!

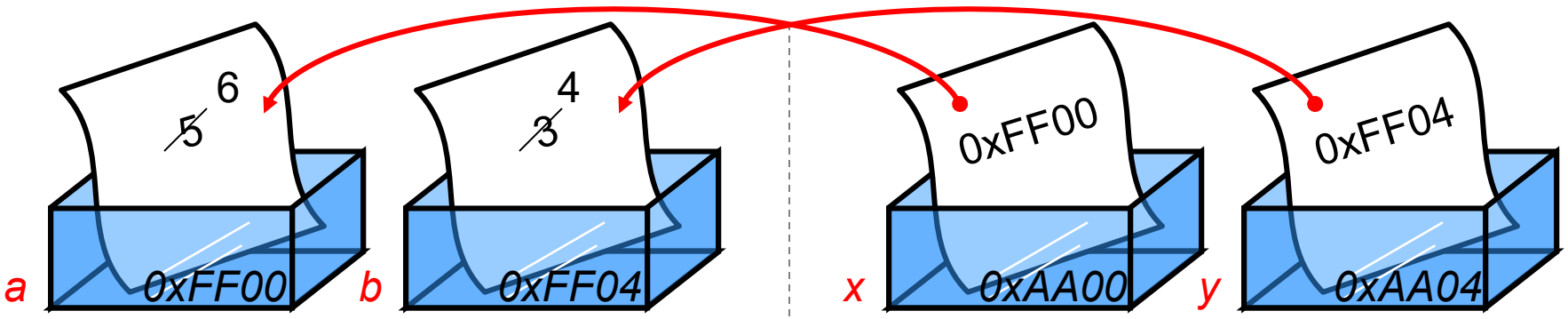
The new set of variables is actually **pointing** to *a*, *b*

Pass by Reference

```
void plus_one(int &x, int &y) {
   x++;   y++;
}
```

reference parameters

```
int a = 5, b = 3;
plus_one(a, b);
```



The values of *a*, *b* have been modified!

The new set of variables is actually **referencing** to *a*, *b*

C++ Reference Example

```
int i = 2;
//an initial value must be provided in the declaration of r
int &r = i;         //r is a reference to an integer
int *p = &i;        //p is a pointer to an integer

printf("%d %d %d %d\n", i, r, p, *p);
// output: 2 2 001AF9C0 2

r = 4;
printf("%d %d\n", i, r);
// output: 4 4
```

Reference vs. Pointer

1. Pointers can point nowhere (NULL), whereas reference always refers to an object.
2. References must be initialized as soon as they are created.
3. A pointer can be re-assigned any number of times while a reference cannot be re-seated after binding.
4. You cannot take the address of a reference like what you can do with pointers. Any occurrence of its name refers directly to the object it references.
5. There is no reference arithmetic but you can take the address of an object pointed by a reference and do pointer arithmetic on it (because of #4).

Pseudo Code

- We need a language to express program development
  - English is too verbose and imprecise.
  - The target language, e.g. C/C++, requires too much details.
- Pseudo code resembles the target language in that
  - it is a sequence of steps (each step is precise and unambiguous)
  - it has similar control structure of C/C++
- Pseudo code is a kind of structured English for describing algorithms. It allows the designer to focus on the logic of the algorithm without being distracted by details of language syntax.

```
x = max{a, b, c}
```

Pseudo code

```
x = a;
if (b > x)  x = b;
if (c > x)  x = c;
```

C++ code

Pseudo Code Example

- An $m \times n$ matrix is said to have a saddle point if some entry A[i][j] is the smallest value on row *i* and the largest value in column *j*.

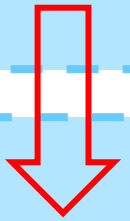
An 6×8 matrix with a saddle point

11 33 55 16 77 99 10 40
29 87 65 20 45 60 90 76
50 53 78 **44** 60 88 77 81
46 72 71 23 88 26 15 21
65 83 23 36 49 57 32 14
70 22 34 19 54 37 26 93

- Problem:
- Given an $m \times n$ matrix, determine if there exists one or more saddle points.

Pseudo Code Solutions

```
// high-level pseudo code solution
for each row {
   j = index of the smallest element on row i;
   if (A[i][j]) is the largest element in column j)
      A[i][j] is a saddle point;
}
```



```
// refined pseudo code
for (i = 0; i < m; i++) {                  //for each row
   j = index of the smallest element on row i;
   for (k = 0; k < m; k++)                 //for each element in column j
      if there does not exist A[k][j] > A[i][j]
         A[i][j] is a saddle point;
}
```

Suggestions for Good Style

- Use informative and meaningful variable names
- Insert useful comments (i.e. assertions) in the source program
- Format the source file with proper indentation of statements and align the braces so that the control structures can be read easily
- Do not use goto statement, especially backward jump
- Use single-entry single-exit control blocks, or at most one break statement inside a loop
- Avoid ambiguous statements e.g. x[i] = i++;
- Minimize direct accesses to global variables, especially you should avoid modifying the values of global variables in a function
- Always make a planning of the program organization and data structures before start writing program codes
- Should avoid using the trial-and-error approach without proper understanding of the problem to be solved

Standard Input / Output

cin & cout

- Default input/output stream objects
- A stream is a sequence of bytes (characters) that can be read from or written to
  - cin is a stream on the keyboard input
  - cout is a stream on the screen output

- The extractor (>>) / insertor (<<) is used to read/write from/to the input/output stream

Standard Output

```
#include <cstdio>
#include <iostream>
using namespace std;
…
int x = 1;
float y = 2.5;
char z = ‘a’;
char w[80] = “xxxxxx”;

printf("%d %f %c %s\n", x, y, z, w);
std::cout << x;
cout << endl;
cout << y << " " << z << " " << w;
```

How to output the values to standard output (screen)?

Use printf() in <cstdio>:

- integer: %d
  float: %f
  character: %c
  string: %s

Use cout in <iostream>:

- *cout* is defined in the *std* namespace
- Use insertion operator to insert values to output stream.
- Multiple insertions can be chained.
- Use *endl* to set a new line.

Standard Input

```
#include <cstdio>
#include <iostream>
using namespace std;
…
int x;
float y;
char z;
char w[80];

cin >> x;
scanf("%f", &y);
cin >> z;
scanf("%s", w);
```

How to read the values from standard input (console)?

Use scanf() in <cstdio>:

- integer: %d
  float: %f
  character: %c
  string: %s

Use cin in <iostream>:

- *cin* is defined in the *std* namespace
- Use extraction operator to extract values from input stream.

scanf()

- *scanf* can only read a “word”, but not a sentence. It stops reading if meets whitespace characters.
- What are whitespace characters?
  - Blank space: ‘ ’
  - Newline: ‘\r’ ‘\n’
  - Tab: ‘\t’
- Visual Studio compiler will tell you the function *scanf* is not safe.
  - Add this code to the beginning of your program to suppress this MS secure warning

```
#ifdef _MSC_VER
#define _CRT_SECURE_NO_WARNINGS
#endif
```

scanf() Examples

scanf() will stop reading when it meets enter, space or tab (whitespace)

```
scanf("%s", w);
printf("##%s##\n", w);
```

```
abc<enter>
##abc##
```

The newline character has been ignored by scanf()

Space

```
scanf("%s", w);
printf("##%s##\n", w);
```

```
abc def<enter>
##abc##
```

The space and following characters have been ignored by scanf()

More on Input

- When looking for the input value in the stream, the >> operator skips any leading whitespace characters and stops reading at the first character that is inappropriate for the data type (whitespace or otherwise).
- You can use the **get()** function to input the very next character in the input stream without skipping any whitespace characters:

```
char someChar;
cin.get(someChar);
```

- The **ignore()** function is used to skip characters in the input stream:

```
cin.ignore(200, '\n');
```

- The first parameter is an int expression; the second, a char value. This skips the next 200 characters or until a newline character is read, whichever comes first

Output Manipulators

- Manipulators change the output format of your data. To use them, you will need to include this header in your C++ source code.

```
#include <iomanip>
```

- setw() sets the width of the field to be printed to the screen
  - `cout << 5 << setw(4) << 6 << 7;  // output:5   67`
- setprecision() sets the *decimal precision* to be used to format floating-point values:
  - `cout << setprecision(5) << 3.14159;  // 3.1416`
  - `cout << setprecision(1) << 3.14159;  // 3`
- To specify the number of digits after the decimal point:
  - `cout << setiosflags(ios::fixed);  // not use scientific notation`
  - `cout << setprecision(2) << 12.1234; // 12.12`
- Other floating point output flags:
  - `setiosflags(ios::scientific);  // use scientific notation`
  - `resetiosflags(ios::floatfield)  // restores default (use fixed // or scientific notation based // on the precision)`

File Input/Output

- In a similar way C++ provides streams which can manipulate files
- C++ provides 2 file streams

  ifstream input file stream

  ofstream output file stream

  Must `#include <fstream>` to use them

- Example:

```
#include <fstream>

int number;
ifstream  in("in.dat");
ofstream out("out.dat");
in >> number;
out << number;
```

Input File Streams (ifstream)

- Allows data to be read from a file
- An input file stream can be defined as follows:

```
ifstream stream_var(filename);
```

Example:

```
ifstream inFile("test.dat");
```

  - If stream opened successfully, inFile evaluates to positive and the stream becomes attached to the file test.data
  - If stream open failed (e.g. file does not exist) inFile evaluates to zero

- **Important:** Effects of reading data from file which has failed to open is undefined

Input File Streams (ifstream)

- When file opened successfully, data can be read using normal extractor functions

```
int n;
char c;
ifstream inFile("test.dat");
inFile >> n;
inFile.get(c);
inFile.close();
```

- **Note:** When a file stream goes out of scope it will automatically close the file it is attached to

File Input Failure/End

- To check if the file has been opened or not, you can use:

```
if (inFile) // testing if the file opened successfully
{ ... }
```

- To test for end of file, you can use:

```
while (!inFile.eof())
{ ... }
```

For instance:

```
int number;
inFile >> number;  // reading number from a file
while (!inFile.eof())
{
     cout << number; // print number on screen
     inFile >> number;
}
```

Output File Stream (ofstream)

- Allows data to be written to a file

  An output file stream can be defined as follows:

  ```
  ofstream stream_var(filename);
  ```

  Example:

  ```
  ofstream outFile("temp.data");
  ```

  - If stream opened successfully, outFile evaluates to positive and the stream becomes attached to the file temp.data
  - If stream open failed (e.g. no disk space) outFile evaluates to zero

- **Note:**
  - If the file already exists its contents will be deleted
  - If the file does not exist, a file with the same name is created
  - Data can be appended to a file by using constructor with two arguments

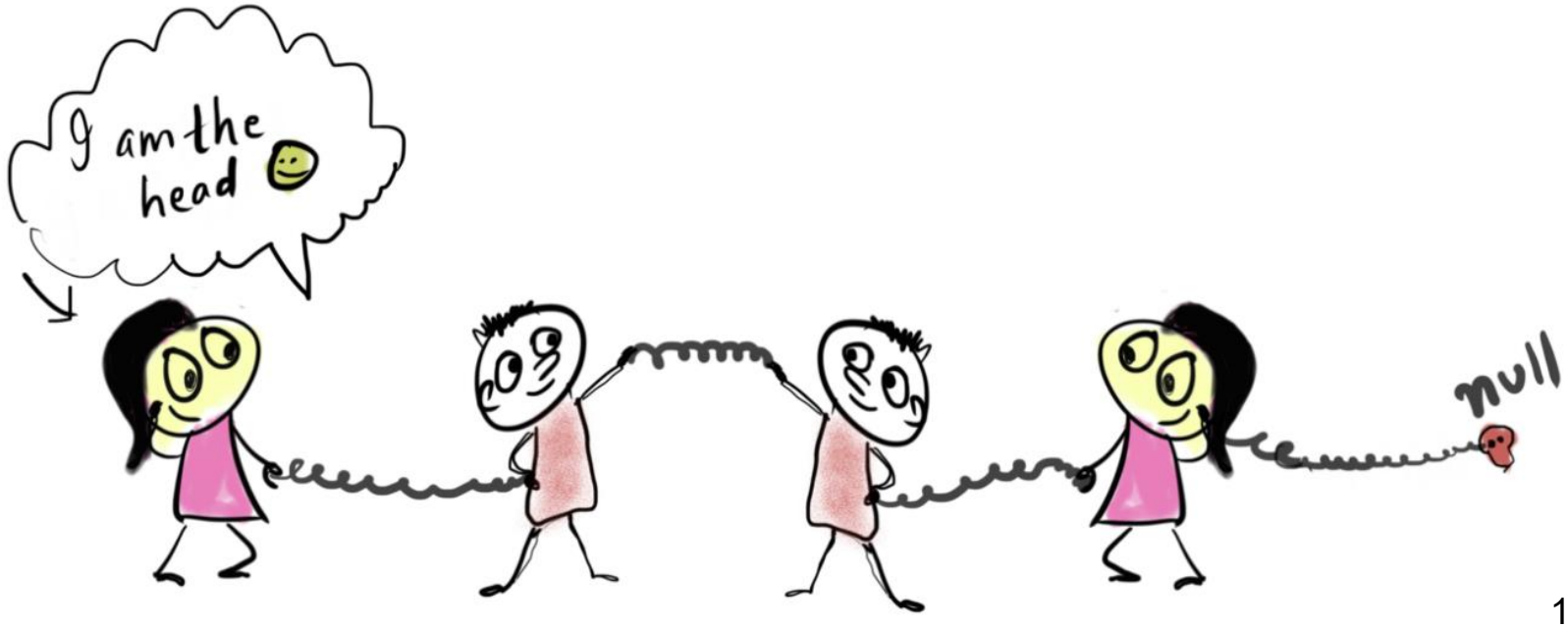
    ```
    ofstream outFile("temp.data", ios::app);
    ```

Example on How to Write to a File

```
#include <iostream>
#include <fstream>
#include <iomanip>

using namespace std;

int main ()
{
   float first, second, sum;                // Declaring variables
   ofstream outFile("out.dat");             // Opening file for output

   cout << "Enter two numbers" << endl;
   cin >> first >> second;                  // Reading in the two numbers
   sum = first + second;
   outFile << setiosflags(ios::fixed);      // Formatting the output
   outFile << setprecision(2);
   outFile << sum << endl;                  // Writing into the file

   return 0;
}
```

EE2331 Data Structures and Algorithms

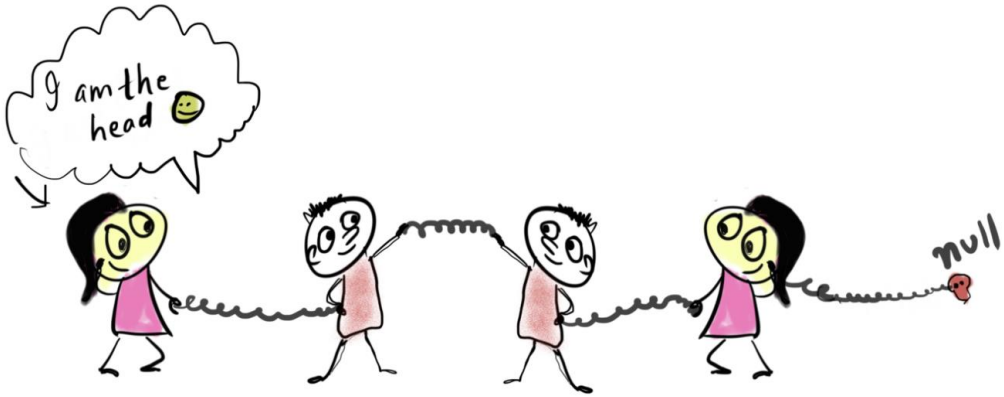
Linked List, Stack, Queue: non-random access linear data structure



Compared to array:

No random access

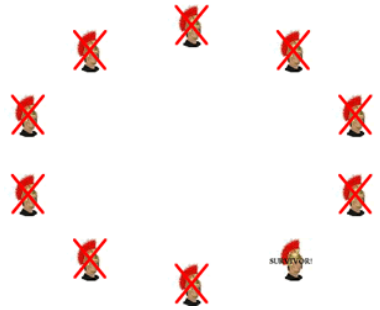
- Array A[5]={1,2,3,4,0}
  - A[0]=1, A[2]=3, etc.
- Linked list, stack, queue
  - Lack direct access to the elements by their positions
  - has to iterate from a known position (like the beginning or the end) to that position, which takes linear time in the distance between these



Linear List

- Each element in the list has a unique predecessor (previous) and successor (next).
- Unordered/Random list
  - There is no ordering of the data.
- Ordered list
  - The data are arranged according to a key. A key is one or more fields within a structure that is used to identify the data or otherwise control its use.
- General list
  - Data can be inserted and deleted anywhere and there are no restrictions on the operations that can be used to process the list.
- Restricted list
  - Insertion, deletion and processing of data are restricted to specific locations, e.g. the two ends of the list. Stack and Queue are examples of restricted list.

Josephus Problem



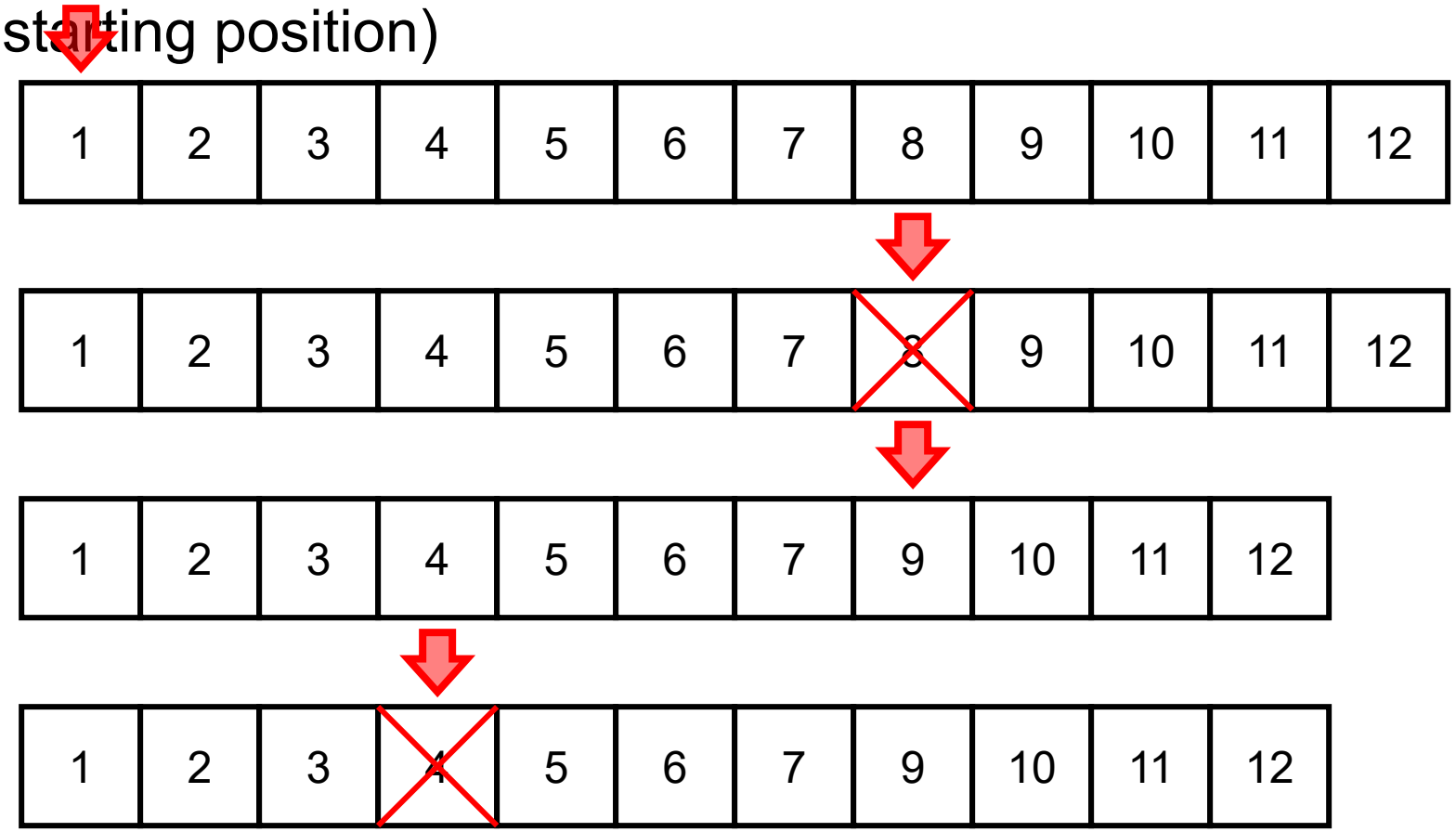
- People are standing in a circle waiting to be executed. Counting begins at a specified point in the circle and proceeds around the circle in a specified direction. After a specified number of people are skipped, the next person is **executed**. The procedure is repeated with the remaining people, starting with the next person, going in the same direction and skipping the same number of people, until only one person remains, and is **freed**.
- The problem — given the number of people (***n***), starting point, direction, and number to be skipped (***k***) — is to choose the position in the initial circle to avoid execution (i.e. **guessing who is the survivor**).

History (from wiki)

- The problem is named after Flavius Josephus, a Jewish historian living in the 1st century. According to Josephus' account of the siege of Yodfat, he and his 40 soldiers were trapped in a cave by Roman soldiers. They chose suicide over capture, and settled on a serial method of committing suicide by drawing lots.

The Josephus Problem

- If $k = 7$, $n = 12$ (skip 7 positions including the starting position)



Array Implementation

- A simple approach is by writing a program to simulate the counting-out game. But what data structure should be used?
- With a list using array implementation
  - Array has the advantage of random access (i.e. direct access to any position)
  - However, the insert and delete operation may involve substantial data movement
  - Another disadvantage of representing a list using an array is that the maximum length of the list needs to be determined a priori

Linked List & Node Structure

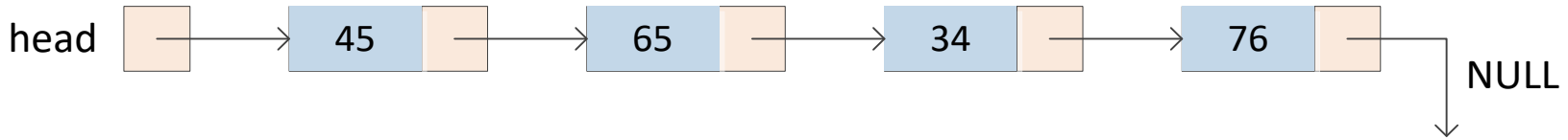
- A sequence of nodes (elements), each containing arbitrary data and links (pointers) pointing to the next and/or previous nodes

- It contains pointer(s) of the same type
  - Recursive data structure (self-referential datatype)
- A list is formed by linking nodes together in a sequential manner
- In typical C++ implementations, we shall define the node using struct, and the linked list is defined as a class.

```
struct node {
   int info;       // data
   node* link;
};

node* head;        // head pointer pointing to first node
```

Linked List Example

- In the C++ terminology, a linked list is classified as a container.
- The address of the first node in the list is stored in a separate pointer variable usually called *head*, *first*, or *list*.
- The null pointer (NULL, physical value 0 in C/C++) is used to denote the end of the list, or not a valid address.
- Example: a linked list of 4 integers.



Common Operations on Linked List

https://yongdanielliang.github.io/animation/web/LinkedList.html

Find Length & Find Node

- To find the length of the linked list

```
int len = 0;
node *cur = head;          //traverse the list using cur

while (cur != NULL) {      // cur points to a valid node
   len++;
   cur = cur->link;        //move to the next node
}
```

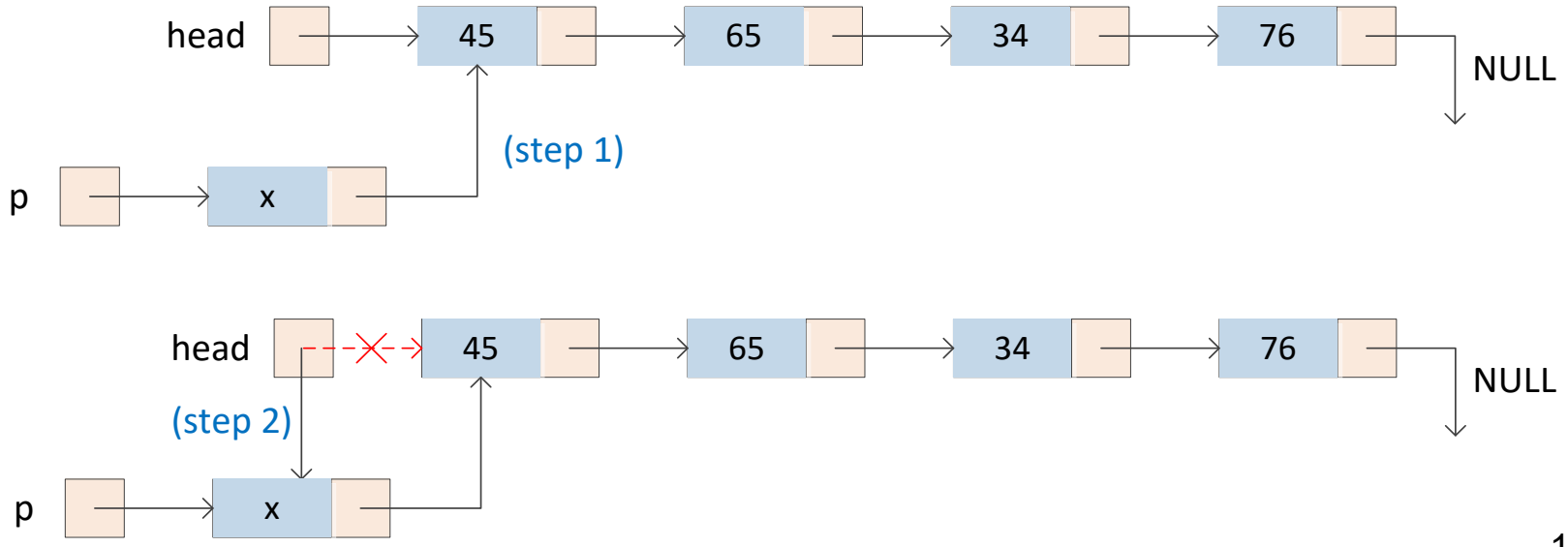
- To search an element x from the beginning of the list

```
node *cur = head;

while (cur != NULL && cur->info != x) {          // search x
   cur = cur->link;
}
// cur now points to target x or is NULL (x not exist)
```

Insert Node (at front)

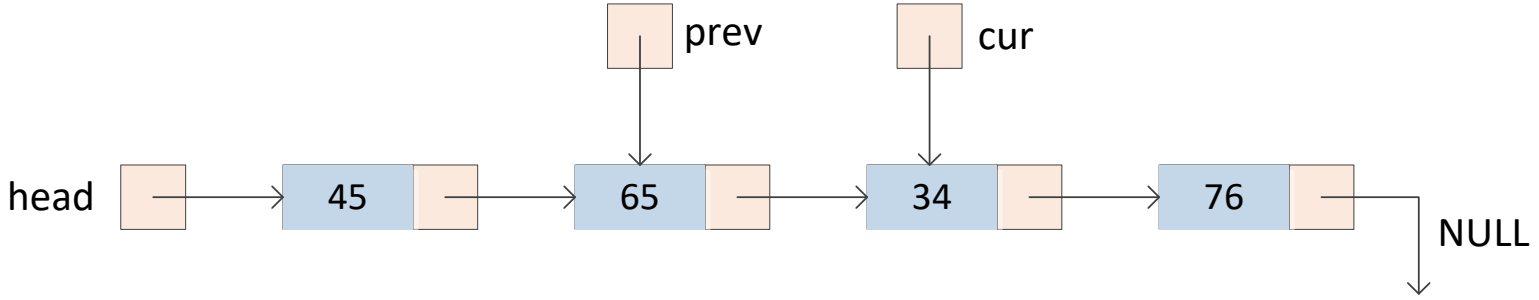
- Insert a new element x at the front of the list

```
node *p = new node;       // create the node dynamically for storing x
p->info = x;

p->link = head;           // step 1
head = p;                 // step 2
```

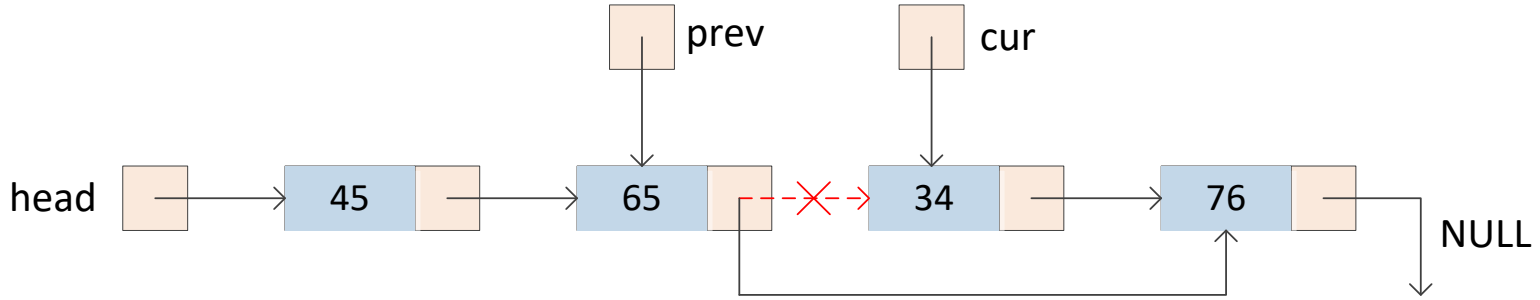


Remove Node

1. Locate the node storing the value x, e.g. x = 34

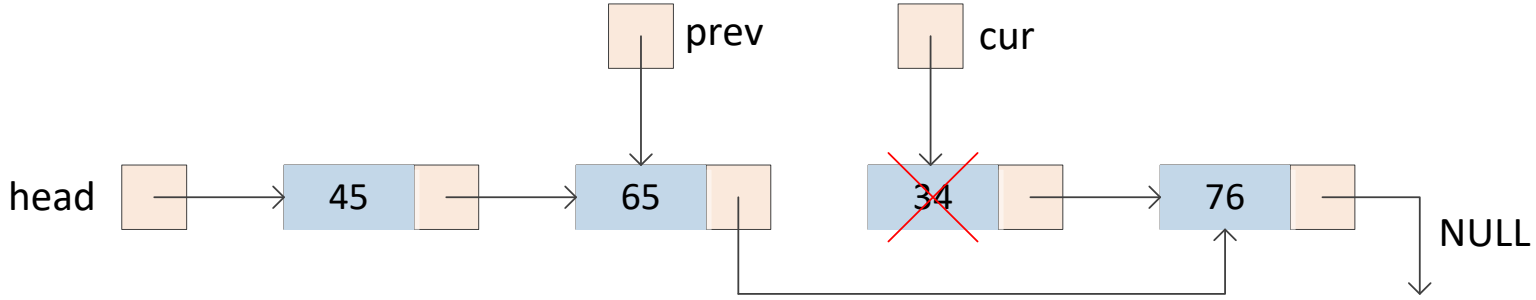


2. Update the links

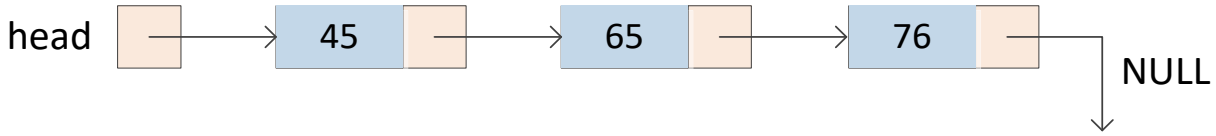


Remove Node (cont.)

3. Physically delete the node



4. Structure of the list after removing the element 34



Remove Node (cont.)

- Remove the element x (1st instance) from the linked list
- To remove a node from the linked list, we need to know the reference to its predecessor.

```
node *cur = head;
node *prev = NULL;          // prev points to the predecessor of cur

while (cur != NULL && cur->info != x) {                 // search x
   prev = cur;
   cur = cur->link;
}

// if cur == NULL, x is not found in the linked list
if (cur != NULL) {          // cur->info == x
   if (prev != NULL)        // why checking this?
      prev->link = cur->link;         // skip cur node
   else                     // cur is the first node in the list
      head = cur->link;     // x is the first node
   delete cur;              //free the storage of the removed node
}
```

Insert Node

- Insert a new element x into an ordered list

```
node *p = new node;
p->info = x;

if (head == NULL || x <= head->info) {
   p->link = head;          //insert at front
   head = p;
} else {                    //head != NULL && x > head->info
   node *prev = head;       // x to be inserted between prev and cur
   node *cur = head->link;  // i.e. prev->info < x <= cur->info

   while (cur != NULL && x > cur->info) {     // search position
      prev = cur;
      cur = cur->link;
   }

   // end-of-list OR x <= cur->info, so insert node p after node prev
   p->link = prev->link;
   prev->link = p;
}
```

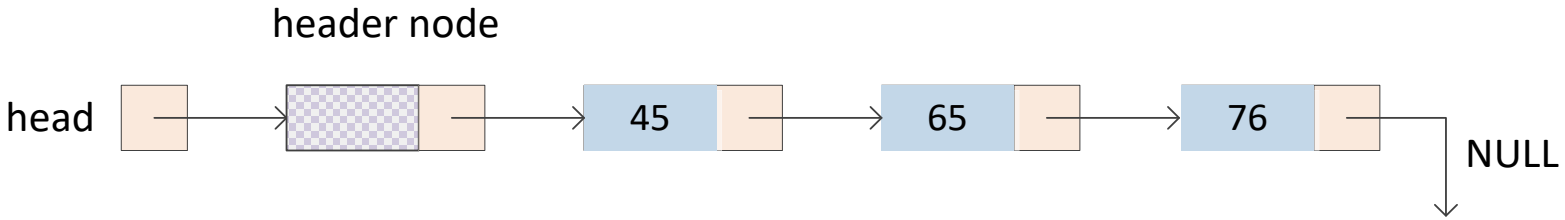
Remark

- Because you can't go backward in a singly linked list, for removing/inserting node, you usually need to use a pair of pointers – *predecessor* and *current*, to keep track of the previous node and update its link.
- For a linked list of size n, you generally should test your algorithm against the two boundary cases: n=0 and n=1 in addition to the general case.
- **Common Problems**
  - Null-pointer exception is a common error in programs that manipulate linked list.
  - A pointer must be properly initialized or tested for not equal to NULL before you can use it to access a data member (dereferencing) or the next node.
  - Broken list due to deletion of nodes
  - Losing reference to some nodes (memory leak)

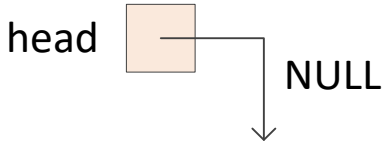
Other Variants of Linked List

Linked List with Header Node

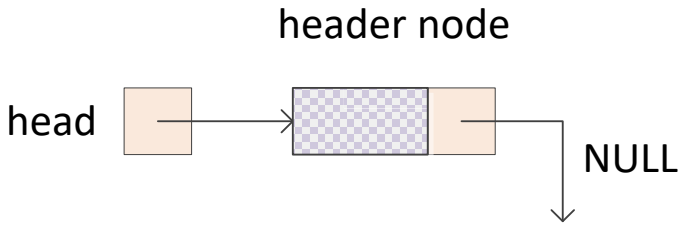
- The data field of the header node is NOT used to store valid data. Some metadata may be stored in the header node.



- List does not exist (or not yet created):



- List is empty:



Why Header Node?

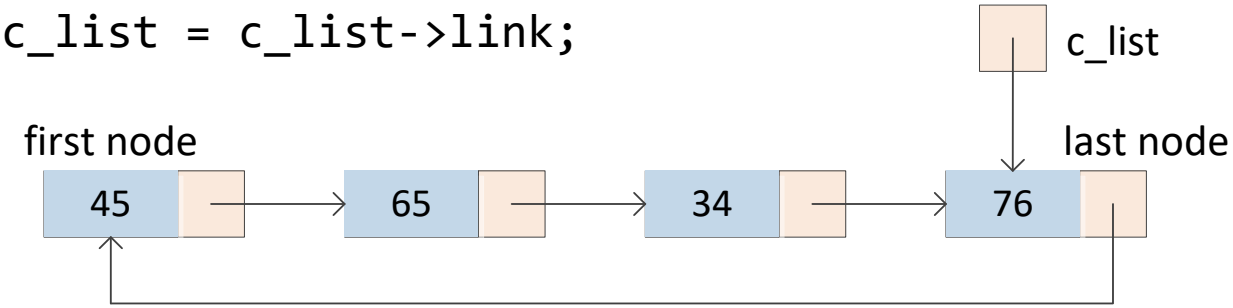
- Header node is guaranteed to exist at all times.
- Header node makes all list nodes intrinsically the same – having a predecessor.
- Having a sentinel zeroth node simplifies a lot of operations you might want to perform on a linked list - for example, a lot of operations no longer need to explicitly check for an empty list.

```
// Simplified version: Insert a new element x into an ordered list
node *p = new node;
p->info = x;

node *prev = head;          // point to the dummy header
node *cur = head->link;     // point to 1st node

while (cur != NULL && x > cur->info) {        // search position
   prev = cur;
   cur = cur->link;
}
p->link = prev->link;
prev->link = p;
```

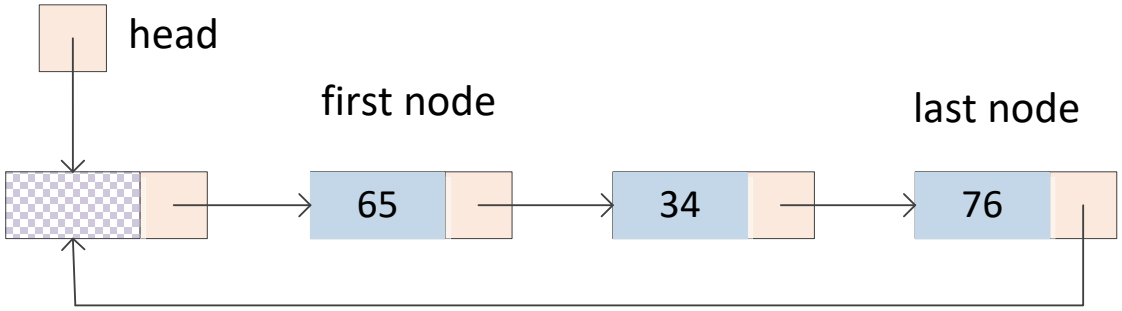
Circularly Linked List (not required)

- The link of the last node points back to the first node.
- When the pointer ***c_list*** reaches the last node in the list, it can re-visit the first node in one step:

```
c_list = c_list->link;
```



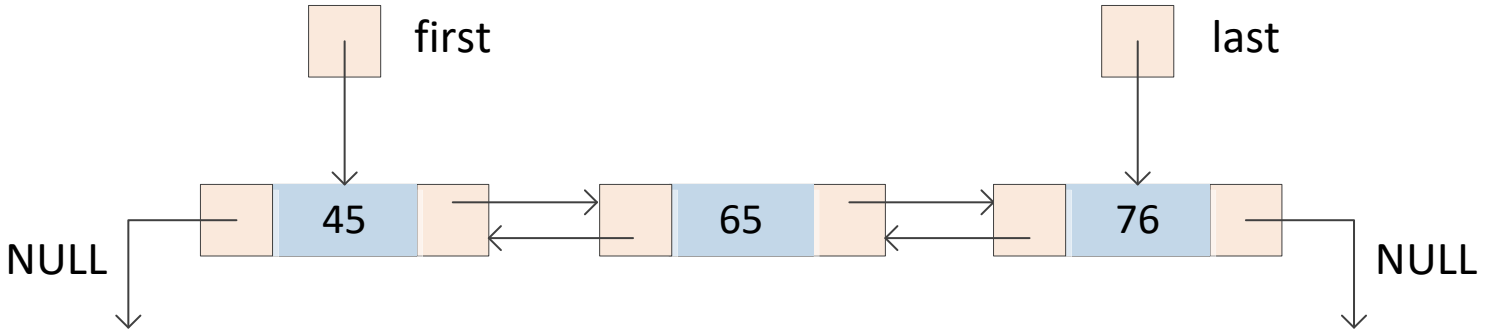
- Circular list with header



Doubly Linked List

- With a doubly-linked list, we can traverse the list in the forward or backward direction.

```
template<class Type>
struct nodeType {
   Type info;
   nodeType<Type> *next;          //points to successor node
   nodeType<Type> *back;          //points to predecessor node
}
```



Insert Node on Doubly Linked List (not required)

- Insert element x after the current node in a doubly-linked list

```
nodeType<Type> *p = new nodeType<Type>;
p->info = x;

// assume current is a pointer to current node
p->next = current->next;      //(1)
p->back = current;            //(2)

if (current->next != NULL)    //current has a successor
   current->next->back = p;   //(3)

current->next = p;            //(4)
```

Remove Node on Doubly Linked List (not required)

- Remove the node pointed by ***p*** in a doubly-linked list

```
if (p->next != NULL)
   p->next->back = p->back;          //(1)

if (p->back != NULL)
   p->back->next = p->next;          //(2)

delete p;
```

In-Class Exercise

- Return the value/info of the last node of a list
- Remove last node of a list

- The given singly linked list has a ***dummy header***
  - Input: a pointer to node structure, *list*, which points to the head of a linked list
  - Precondition: *list* is a valid linked list with header

Search the Last Node

```
// Output: a pointer p points to the last node of the list
node* searchLastNode(node *list) {

}
```

Remove the Last Node

```
// singly linked list with header hode
void removeLastNode(node *list) {

}
```

Linked List C++ Implementation

- Operations that we would perform on a linked list:
- Initialize the list.
- Clear the list.
- Determine if the list is empty.
- Print the list.
- Find the length of the list.
- Make a copy of the list, e.g. assignment operator= and the copy constructor.
- Search the list for a given item.
- Insert an item to the list.
  - The requirement of the insert operation depends on the representation invariant or the intended uses of the list.
  - For ordered list, we need to maintain the ordering of list elements.
  - If it is used as a queue, insertion is performed at the rear (end of list).
  - If it is used as a stack, insertion is performed at the front.

Linked List C++ Implementation

- Remove an item from the list. Similar to the case of insertion.
- Traverse the list (in the application program that uses the linked list object), i.e. retrieve the elements one by one (in some specific order) to carry out the required computation on each node.
  - To implement the traversal, we shall make use of an iterator.
  - A linked list is a container that holds together a collection of items.
  - An iterator is an object that produces each element of a container, one at a time.
  - The two basic operations on an iterator are the dereference operator *, and the pre-increment operator ++ (advance to the next element).
- There can be other operations on the linked list, e.g. reverse the list, merge two lists, etc.

- We want the linked list class to be generic such that it can be used to process different data types.

About using reference for a pointer

```
/* use reference when you need to change the passed parameters for a function

   function(type1& parameter1, type2 parameter2)
   {
       parameter1=.... //modify parameter1
   }

   main....
   {
     type1 x1;
     type2 x2;
     function(x1, x2);

     //x1 is modified accordingly

   }
*/
```

Reference of pointer: Example
(refer to the sample code for tutorial week 3)

```
void insert(ListNode*& head, int x) {
        ListNode* p = new ListNode;
        p->info=x;
        p->link=NULL;

        if (head == NULL || x <= head->info) {
                p->link = head;
                head = p;
        }
        else {
                ListNode* prev = head;
                ListNode* cur = head->link;

                while (cur != NULL && x > cur->info) {
                        prev = cur;
                        cur = cur->link;
                }

                p->link = prev->link;
                prev->link = p;
        }
}
```

```
int main() {
  ListNode* head = NULL;
  ifstream inFile("testData.txt");

  if (!inFile.is_open()) {
    cout << "Error: cannot open data file" << endl;
    exit(0); //terminate the program
  }
  while (!inFile.eof()) { //not end of file
    int i;
    inFile >> i;    //read in an integer

    if (!inFile.fail())
      insert(head, i); //insert into the linked list
    else
      break;
  }
  inFile.close();
}
```

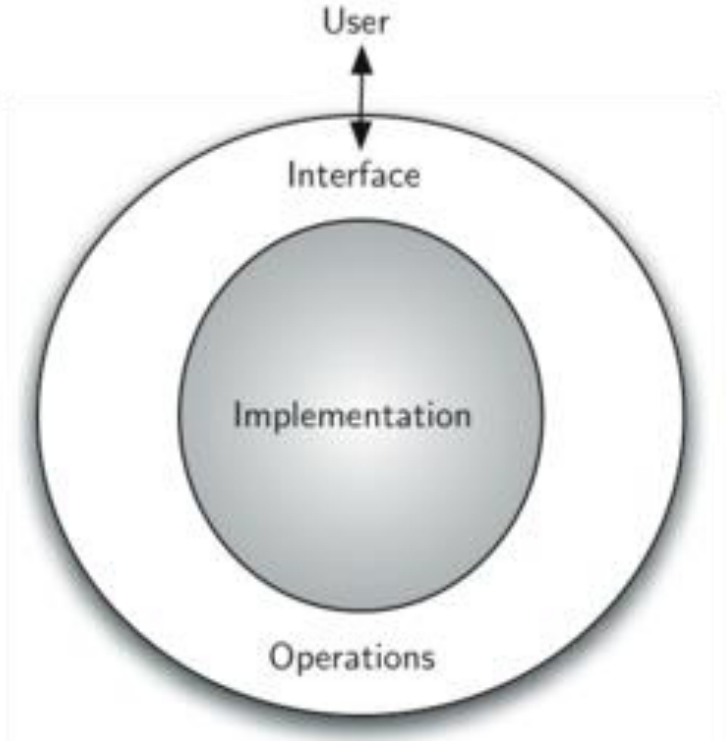
Use linked list in C++ STL

Abstract Data Type (ADT)

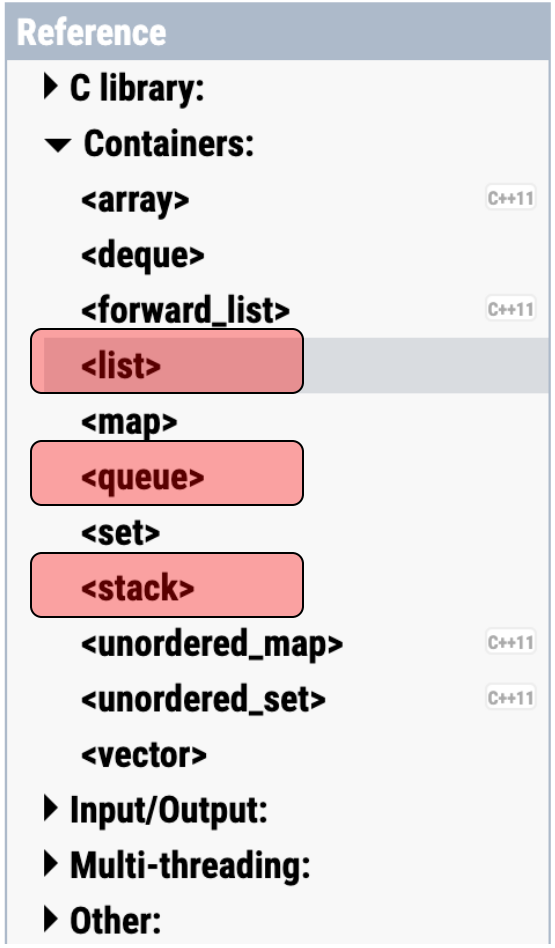
- To manage the complexity of problems and the problem-solving process, computer scientists use abstractions to allow them to focus on the “big picture” without getting lost in the details.
- **Abstract Data Type** is a logical description of how we view the data and the operations that are allowed without regard to how they will be implemented. This means that we are concerned only with what the data is representing and not with how it will eventually be constructed.
- For example, the standardized user interface of an Android phone is a logical property of the device, while the construction of the physical Android phone is the implementation details. From the point of view of the user, you only need to know the logical property (i.e. the user interface) of the device when you are using the phone, and you don’t need to know its internal implementation details.

Abstract Data Type (ADT)

- This provides an **implementation-independent** view of the data. Since there will usually be many different ways to implement an abstract data type, this implementation independence allows the programmer to switch the details of the implementation without changing the way the user of the data interacts with it. The user can remain focused on the problem-solving process.



https://cplusplus.com/reference/stl/



List containers are implemented as doubly-linked lists

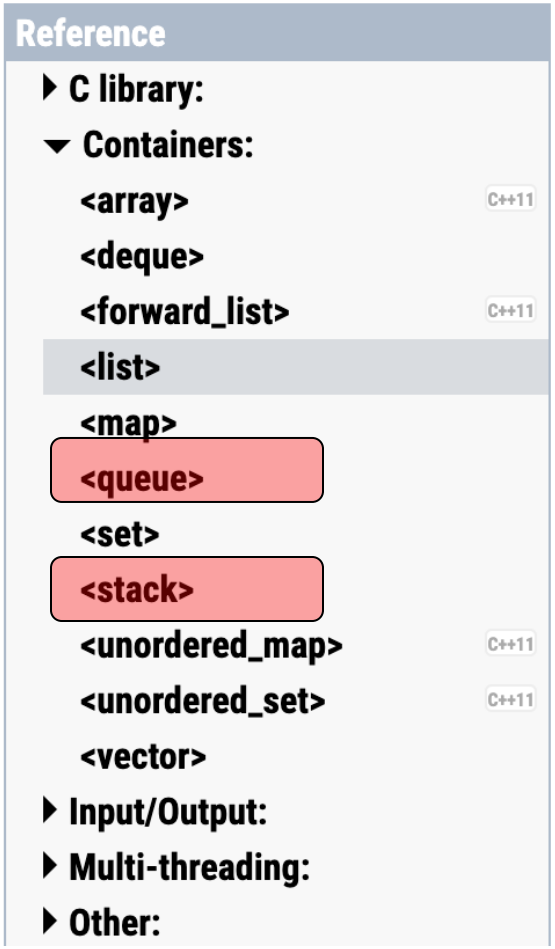
Compared to other base standard sequence containers (array, vector), lists perform generally better in inserting, extracting and moving elements in any position within the container for which an iterator has already been obtained, and therefore also in algorithms that make intensive use of these, like sorting algorithms.
they lack direct access to the elements by their position; For example, to access the sixth element in a list, one has to iterate from a known position (like the beginning or the end) to that position, which takes linear time in the distance between these. They also consume some extra memory to keep the linking information associated to each element (which may be an important factor for large lists of small-sized elements).

Example

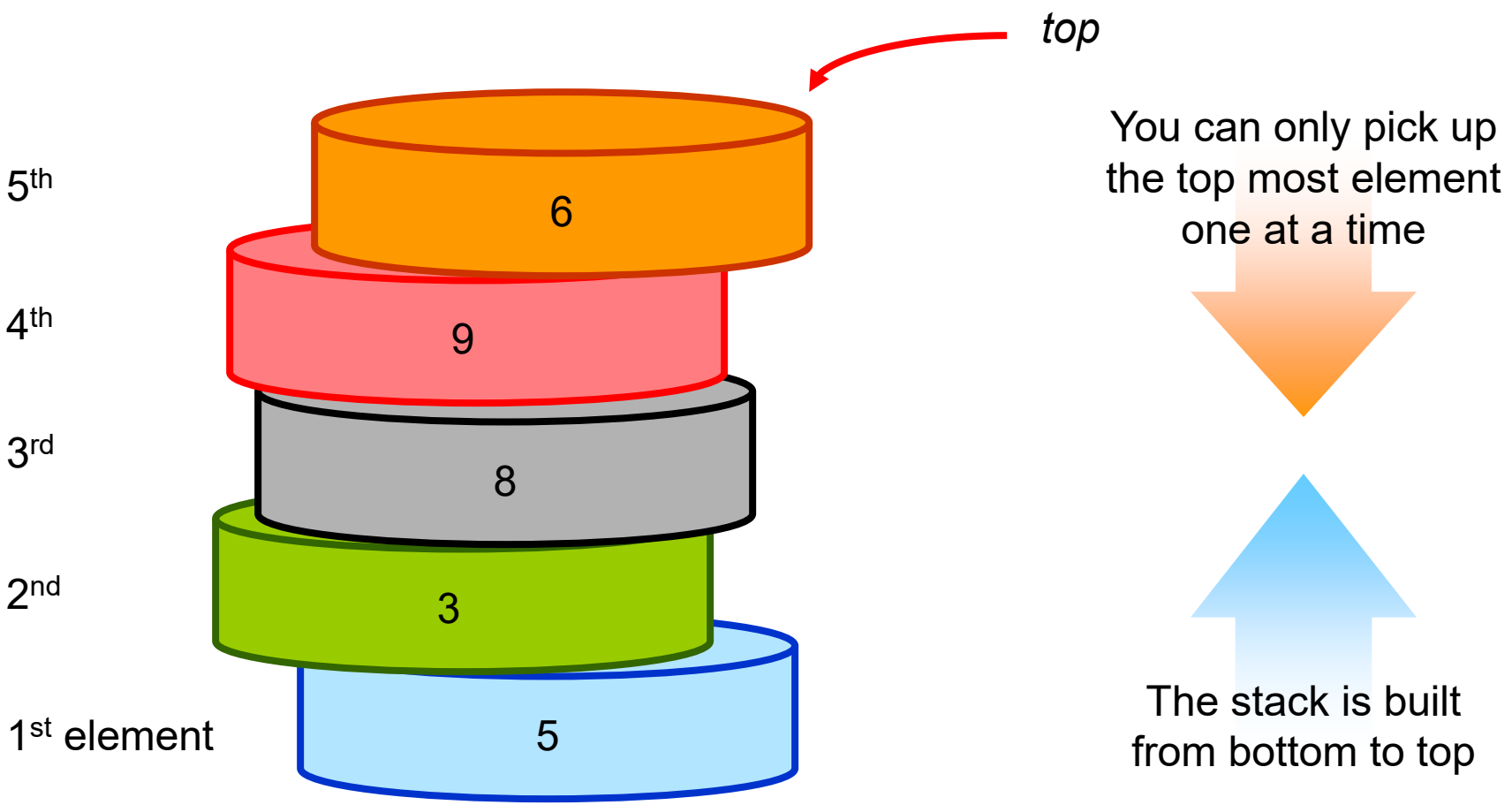
```
1  // constructing lists
2  #include <iostream>
3  #include <list>
4
5  int main ()
6  {
7    // constructors used in the same order as described above:
8    std::list<int> first;                                // empty list of ints
9    std::list<int> second (4,100);                       // four ints with value 100
10   std::list<int> third (second.begin(),second.end());  // iterating through second
11   std::list<int> fourth (third);                       // a copy of third
12
13   // the iterator constructor can also be used to construct from arrays:
14   int myints[] = {16,2,77,29};
15   std::list<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );
16
17   std::cout << "The contents of fifth are: ";
18   for (std::list<int>::iterator it = fifth.begin(); it != fifth.end(); it++)
19     std::cout << *it << ' ';
20
21   std::cout << '\n';
22
23   return 0;
24 }
```

Stack and queue

https://cplusplus.com/reference/stl/

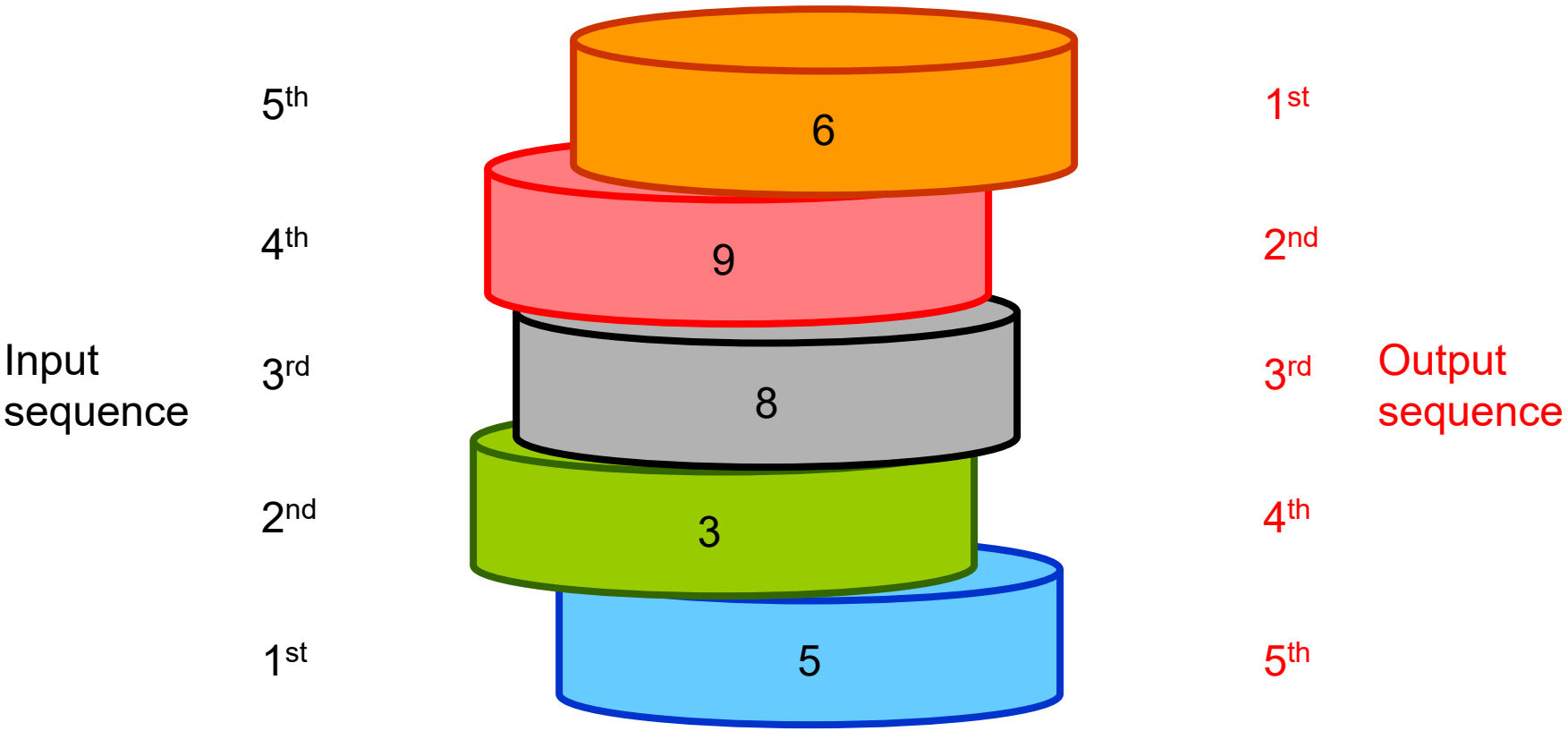


Stack


top
5th
4th
3rd
2nd
1st element
6
9
8
3
5
You can only pick up the top most element one at a time
The stack is built from bottom to top

Input/Output Order

- Last In First Out (LIFO)



Stack Operations

- A stack is a list of homogeneous elements in which the addition and deletion of elements occur only at one end, called the top of the stack.
- A stack is also called a Last In First Out (LIFO) data structure.
- Operations on a stack:
  - **initialize**: initialize the stack to an empty state
  - **size**: determine the number of elements in the stack
  - **empty**: determine if the stack is empty
  - **top**: retrieve the value of the top element
  - **push**: insert element at the top of stack
  - **pop**: remove top element

- In C++, we can define an ADT using an abstract class. In our discussion, I will try to follow the notations used in the C++ STL (Standard Template Library).

Using Stack to Reverse Order

- Use the class ***stack*** in C++ STL

```
#include <iostream>
#include <stack>

using namespace std;

int main() {
          stack<int> s;
          s.push(10);
          s.push(20);
          s.push(30);

          while(!s.empty()) {
                    cout << s.top() << " ";     // output: 30 20 10
                    s.pop();                    // remove the top item
          }
}
```

Using Stack to Evaluate Arithmetic Expression

- How does a computer evaluate this?
  - (4 + 5) * (7 - 2)

- In infix format, the binary operator is placed in between the 2 operands. The order of evaluation is determined by the precedence relation of the operators and parentheses, if any.
  - Order of precedence: ( ) > *, / > +, -
- Postfix notation is another way of writing arithmetic expressions, where the operator is written after the two operands:
  - e.g. 4 + 5 (infix) will be changed to 4 5 + (postfix)
  - The order of evaluation is the same as the order in which the operators appear in the postfix expression.
  - Precedence rules and parentheses are never needed!

Evaluate Postfix Expressions

- In the examples shown below, $ represents the exponentiation operator.

Infix	Postfix
A + B	AB +
A + B − C	AB + C −
(A + B) * (C − D)	AB + CD − *
A $ B * C − D + E / F / (G + H)	AB $ C * D − EF / GH + / +
((A + B) * C − (D − E)) $ (F + G)	AB + C * DE − − FG + $
A − B/(C * D $ E)	ABCDE $ * / −

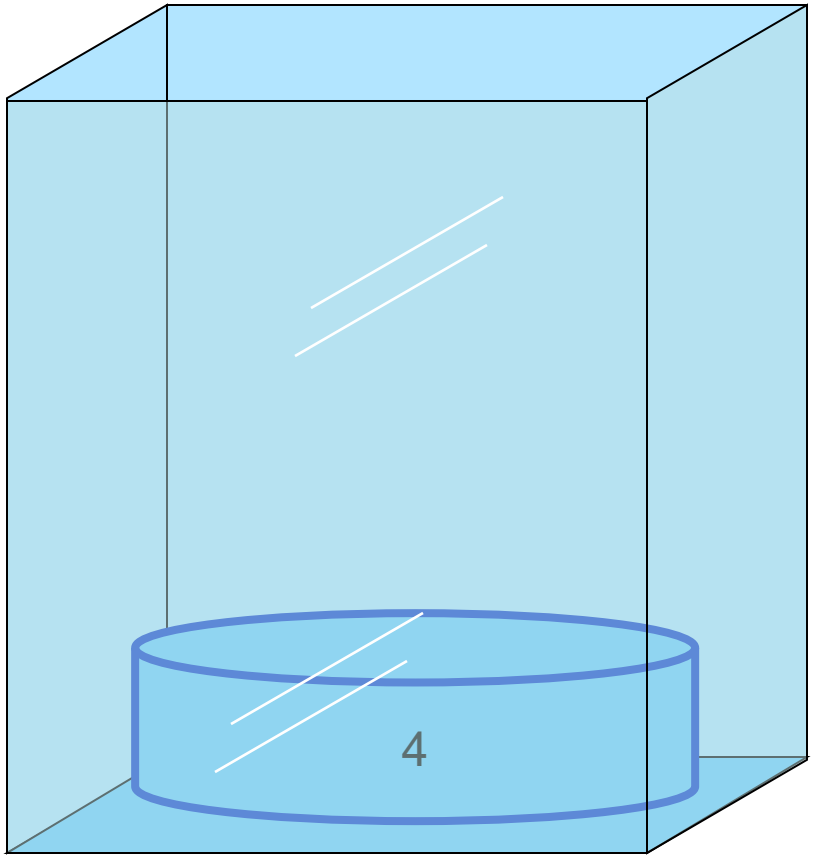
- Read from postfix
  - If input is an operand, push on stack
  - If input is an arithmetic operator
    - pop from stack twice (the two nearest operands)
    - compute their result
    - push the result onto stack

Example

Infix: (4 + 5) * (7 - 2)

Postfix: 4 5 + 7 2 - *

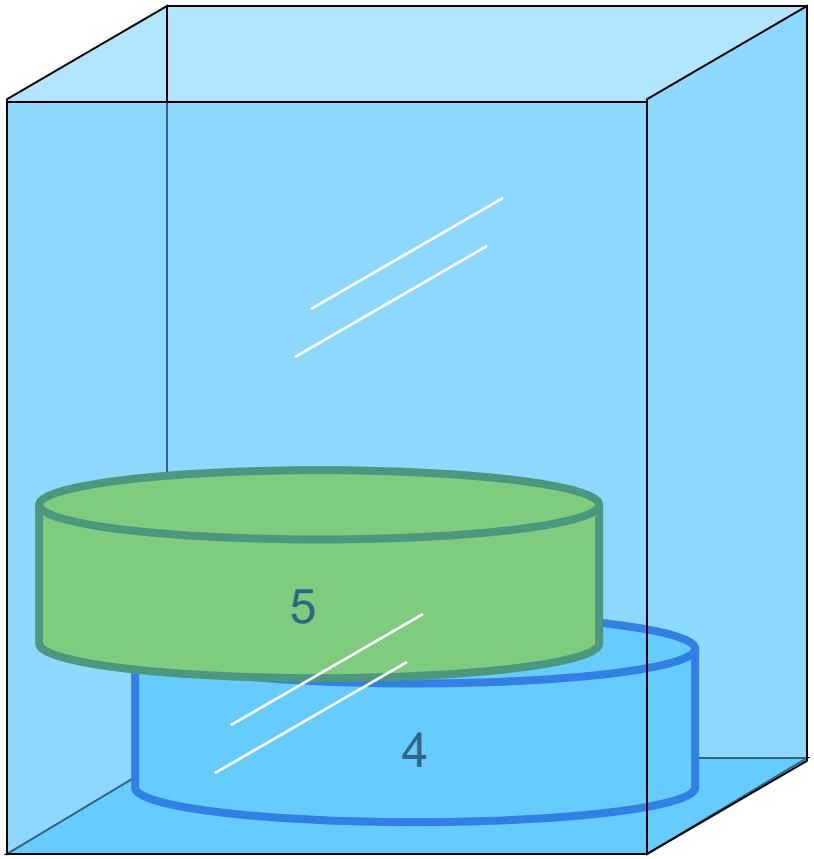
4 5 + 7 2 - *



Step 1: push(4)

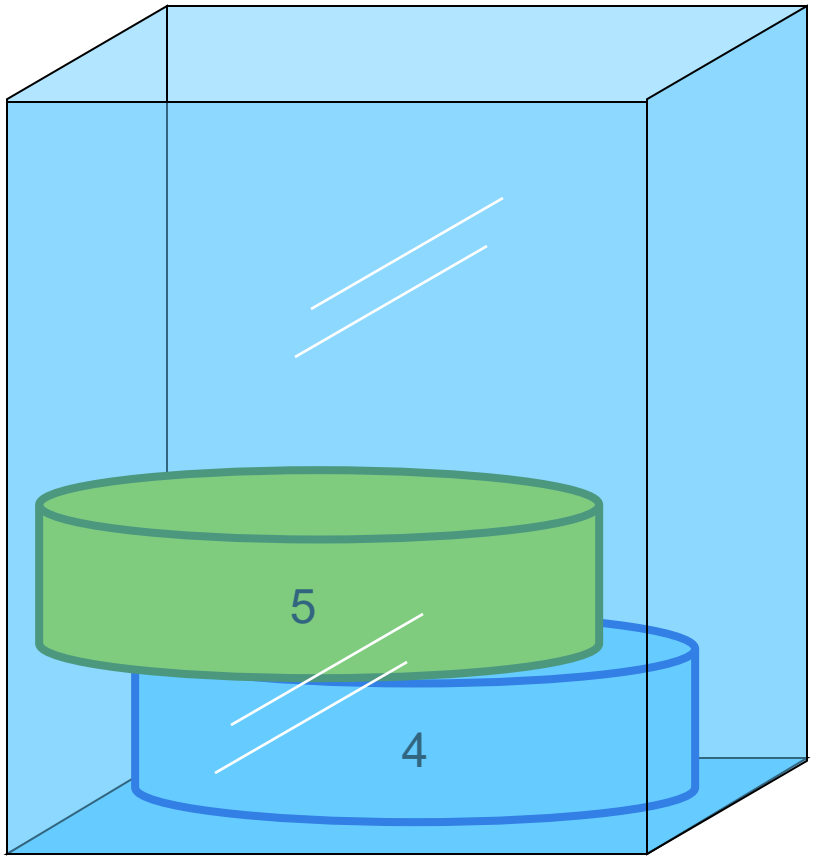
Example

4 5 + 7 2 - *



Step 2: push(5)

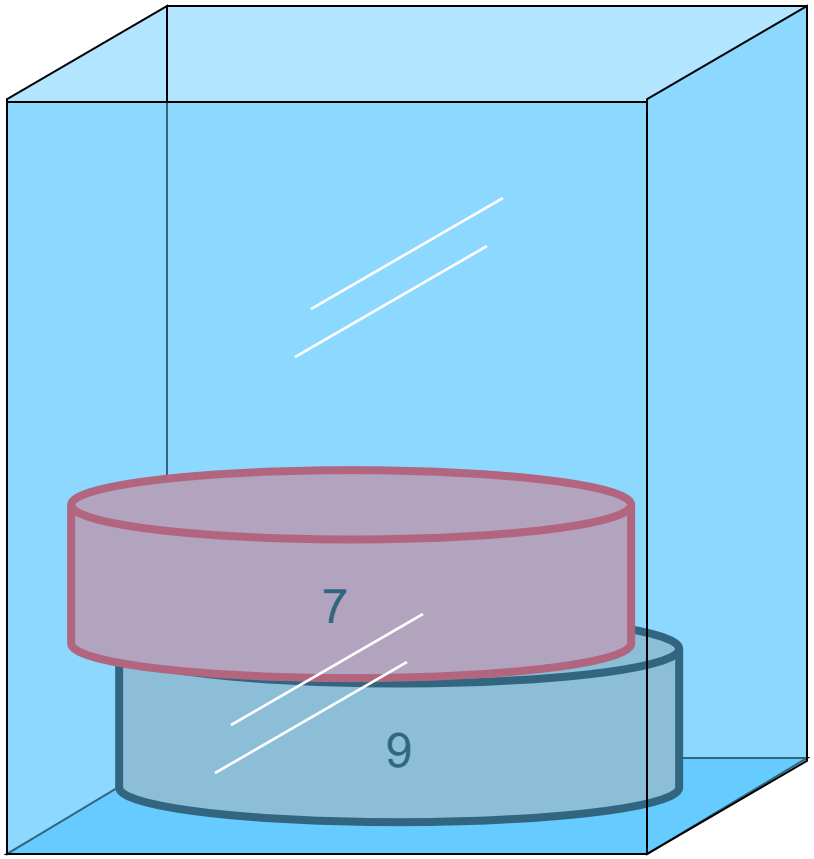
4 5 + 7 2 - *



Step 3: pop() twice and then push the result

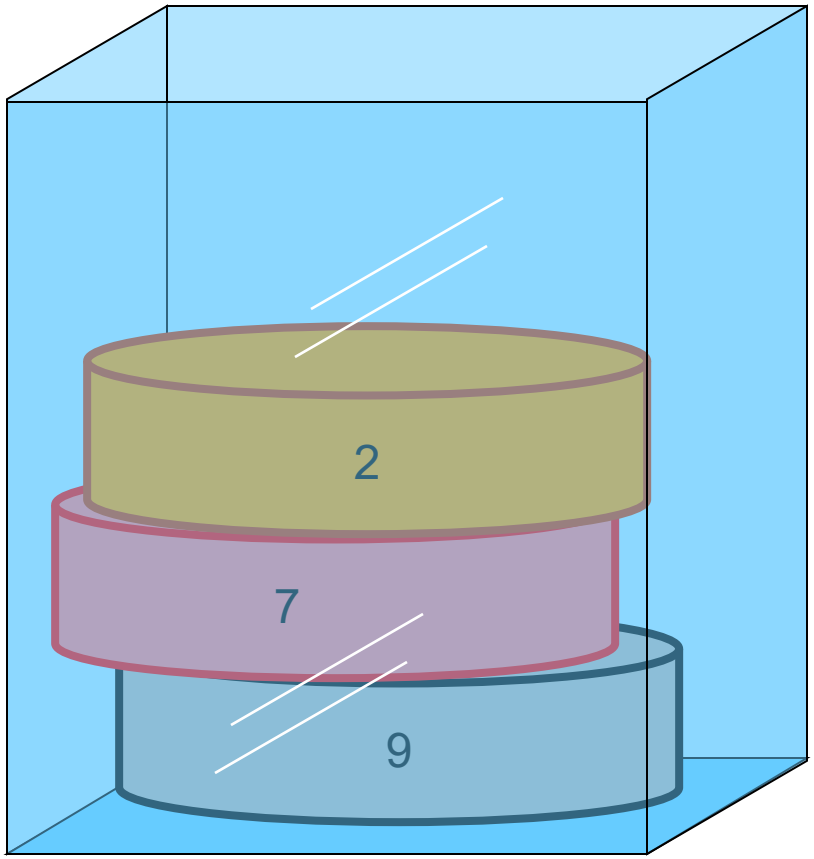
Example

4 5 + 7 2 - *



Step 4: push(7)

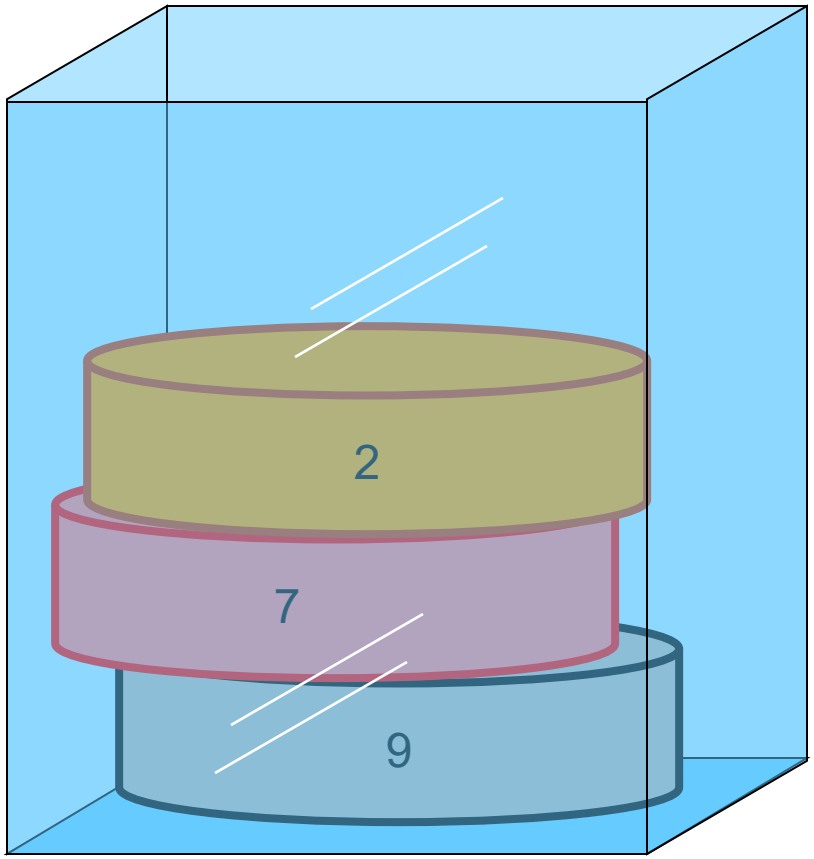
4 5 + 7 2 - *



Step 5: push(2)

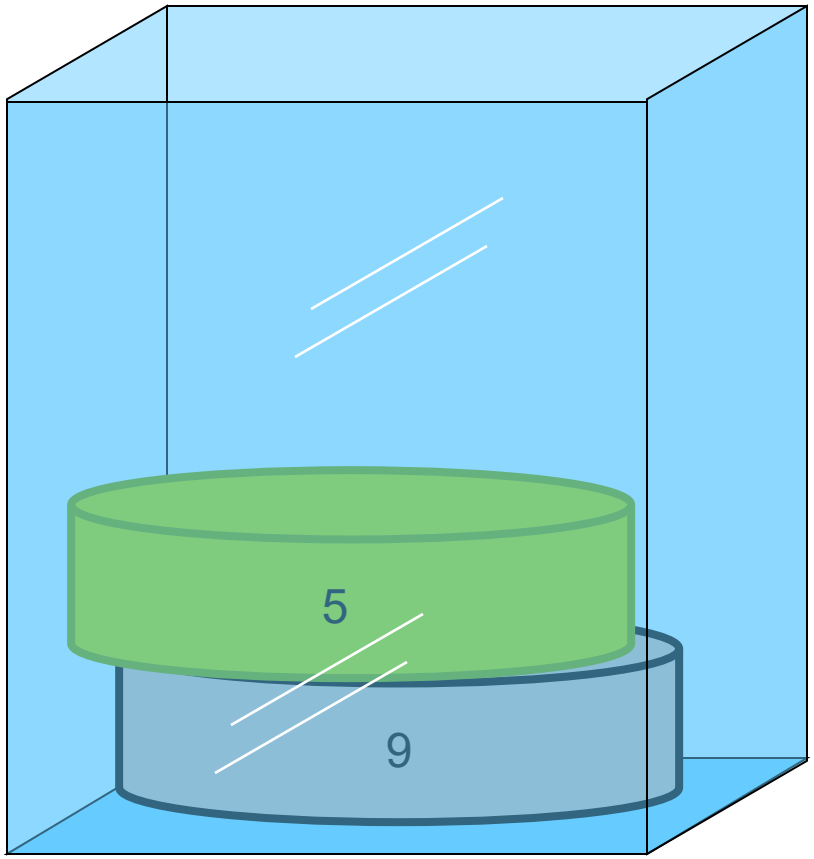
Example

4 5 + 7 2 - *



Step 6: pop() twice and then push the result

4 5 + 7 2 - *

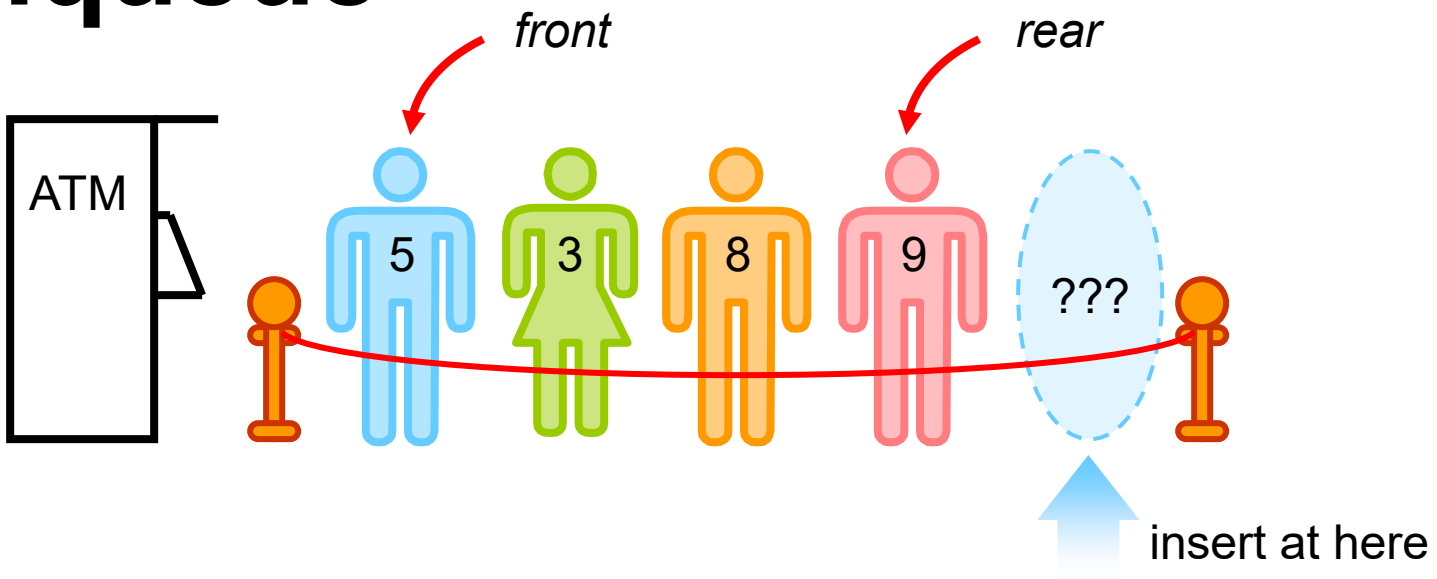


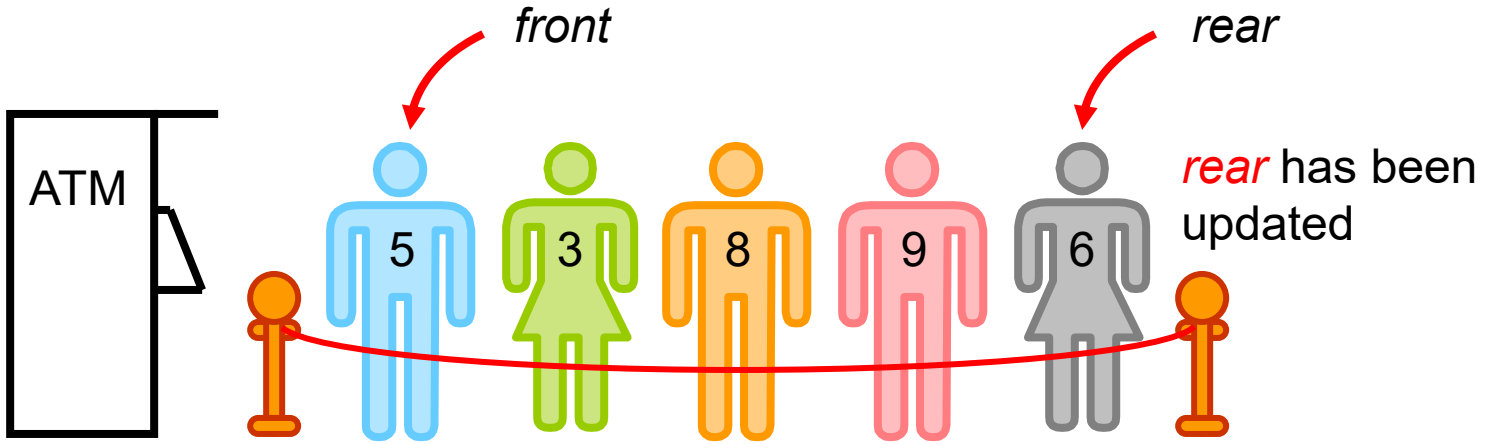
Step 7: pop() twice and compute the result

Queue

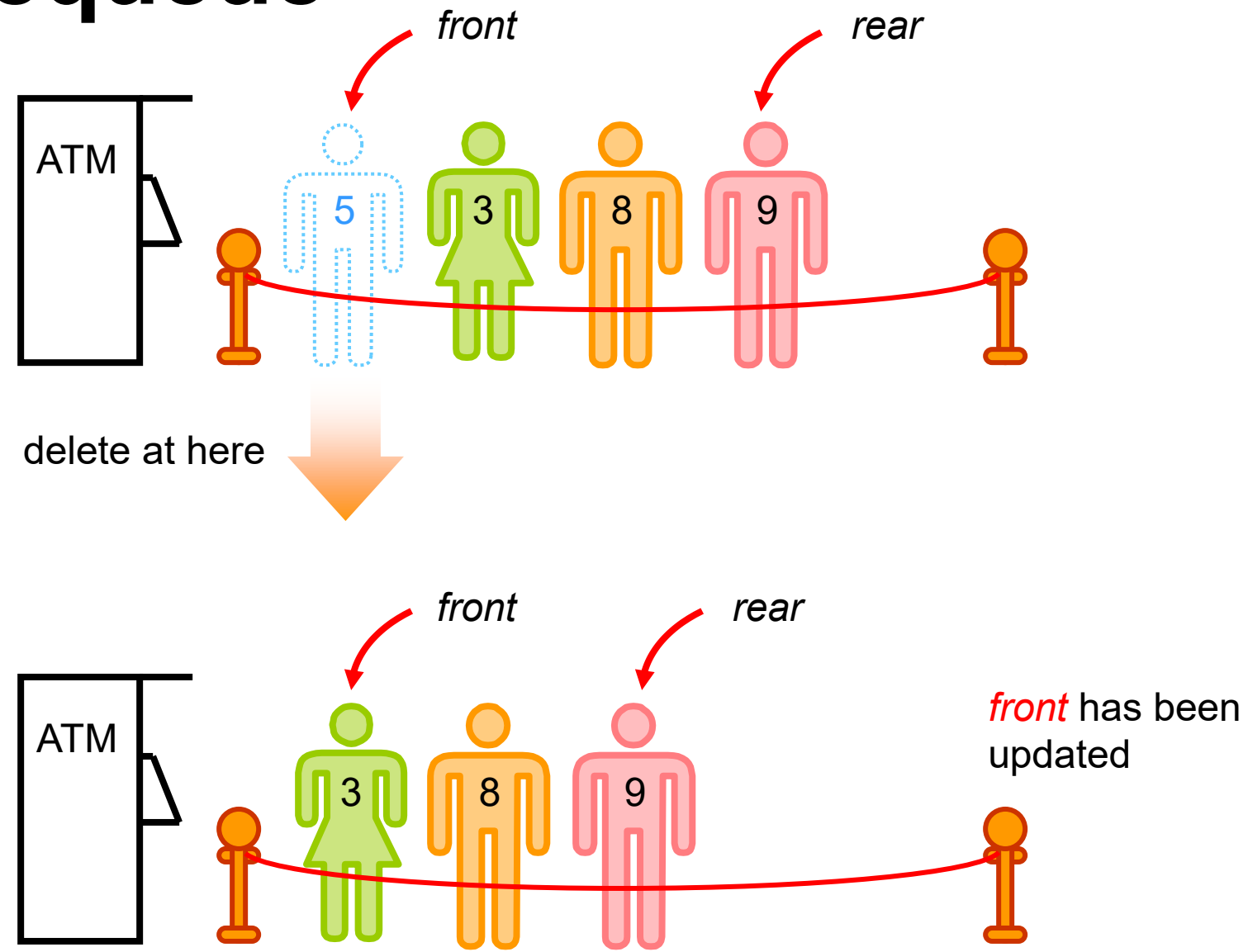
- A first-in-first-out (FIFO) queue is an ordered collection of items from which items may be deleted at one end (called the front) and into which items may be inserted at the other end (called the rear).
- Operations on a queue :
  - **initialize**: initialize the queue to an empty state
  - **size**: determine the number of elements in the queue
  - **empty**: determine if the queue is empty
  - **front**: retrieve the value of the front element
  - **back**: retrieve the value of the last element (this is not common in the applications of queue)
  - **push**: insert element at the rear of queue (in most textbooks, this operation is called enqueue)
  - **pop**: remove front element (in most textbooks, this operation is called dequeue)

Enqueue


front
rear
ATM
5
3
8
9
???
insert at here


front
rear
ATM
5
3
8
9
6
rear has been updated

Dequeue


front
rear
ATM
5
3
8
9
delete at here
front
rear
ATM
3
8
9
front has been updated

Example

```cpp
// queue::push/pop
#include <iostream>       // std::cin, std::cout
#include <queue>          // std::queue

int main ()
{
  std::queue<int> myqueue;
  int myint;

  std::cout << "Please enter some integers (enter 0 to end):\n";

  do {
    std::cin >> myint;
    myqueue.push (myint);
  } while (myint);

  std::cout << "myqueue contains: ";
  while (!myqueue.empty())
  {
    std::cout << ' ' << myqueue.front();
    myqueue.pop();
  }
  std::cout << '\n';

  return 0;
}
```

https://cplusplus.com/reference/queue/queue/push/