EE2331 Data Structures and Algorithms

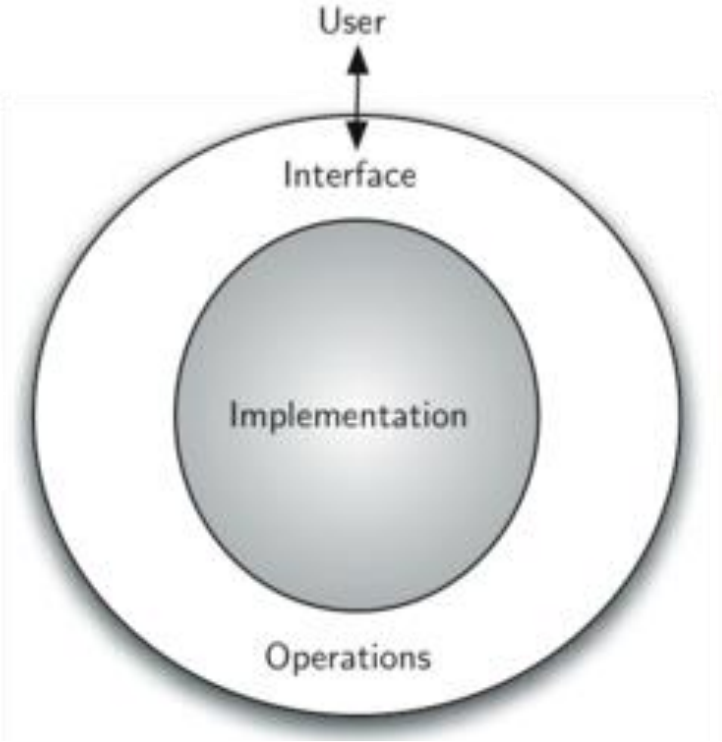
Stacks and Queues

Abstract Data Type (ADT)

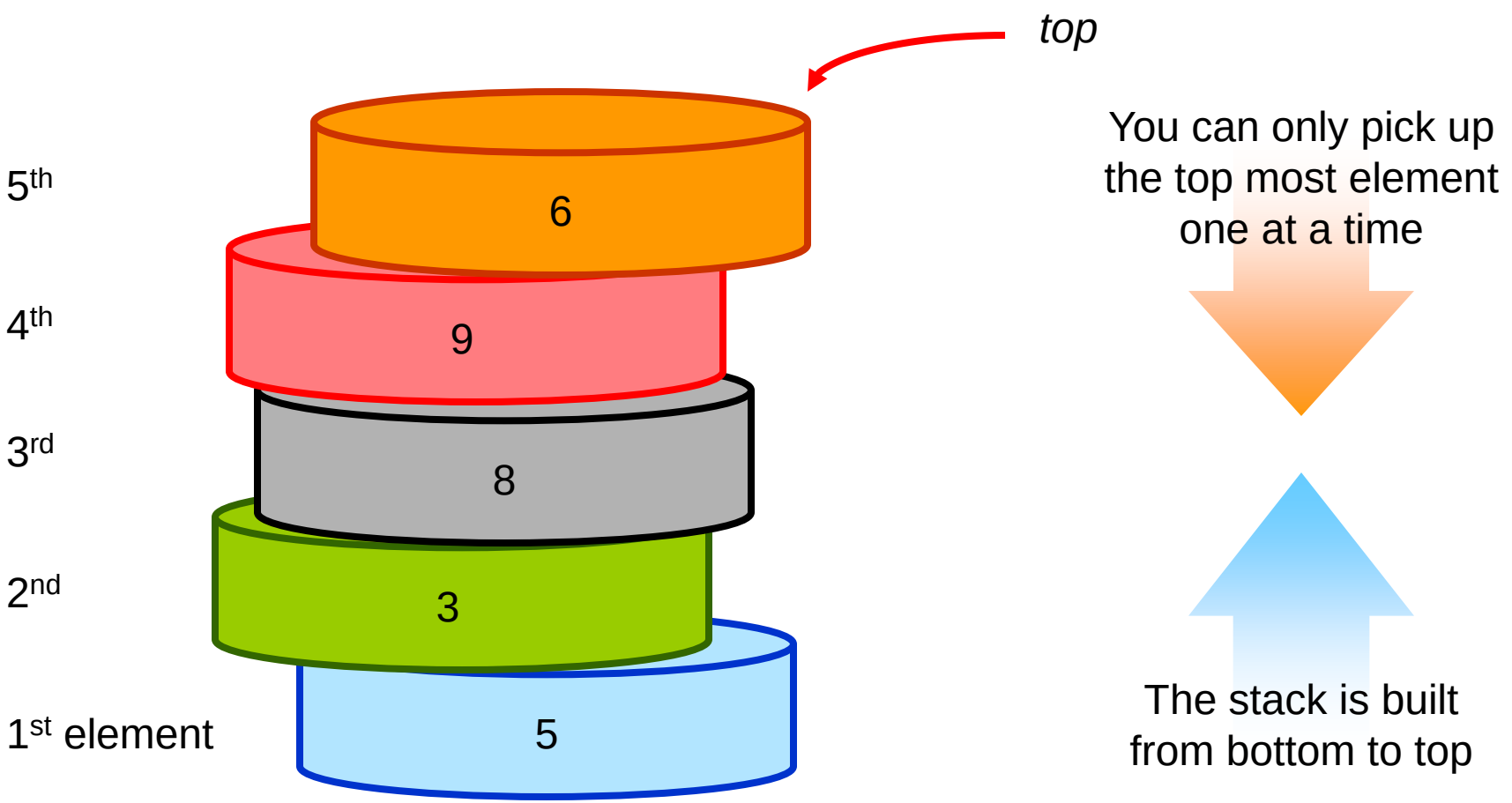
- To manage the complexity of problems and the problem-solving process, computer scientists use abstractions to allow them to focus on the “big picture” without getting lost in the details.
- **Abstract Data Type** is a logical description of how we view the data and the operations that are allowed without 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.



Stack

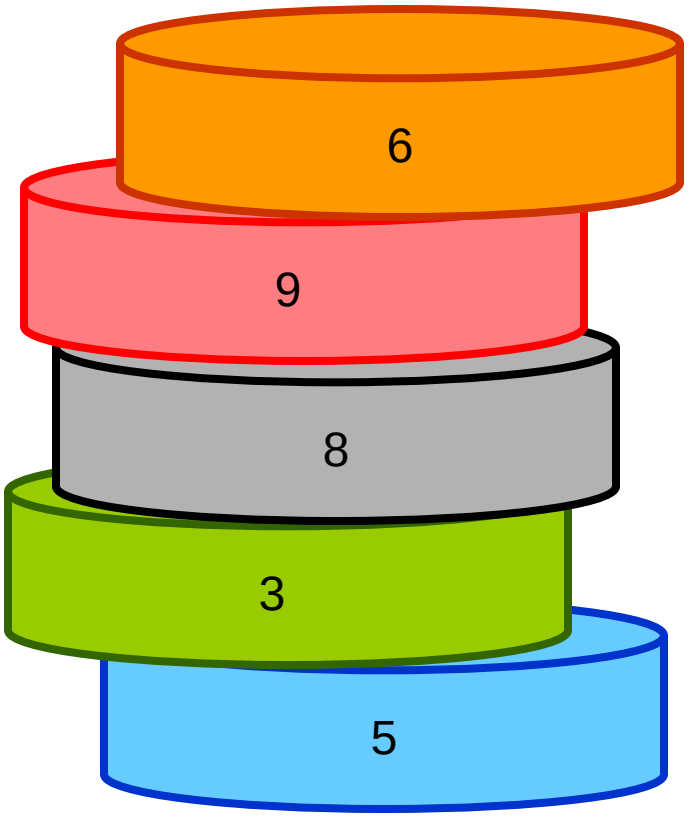

top
5th
4th
3rd
2nd
1st element
6
9
8
3
5
You can only pick up the top most element one at a time
The stack is built from bottom to top

Input/Output Order

- Last In First Out (LIFO)

Input sequence

5th
4th
3rd
2nd
1st



1st
2nd
3rd
4th
5th

Output sequence

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).

Stack ADT (Abstract Class)

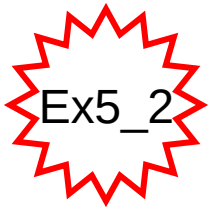

```
template<class Type>
class stackADT {
public:
   virtual void initialize() = 0;     //pure virtual function
   //Note that the initialize() function is not part of the C++ STL. The
   //initialization can be taken care of by the constructor.

   virtual int size() const = 0;
   virtual bool empty() const = 0;
   virtual Type& top() const = 0;     //return reference of the top element
   virtual void push(const Type& item) = 0;    //constant reference
   virtual void pop() = 0;
   //Note that in the C++ STL, the pop function does not
   //return the (old) top element that is removed.

   //Remark: in Java, the pop method of the class Stack
   //        will return the removed element.
}
```

Remarks of Stack

- In the textbook, the author includes a function full() in the stackADT. The function full() returns true if the stack is full.
- However, full() is not part of the stack class in C++ STL. It is not required if the stack is implemented using linked list (i.e. no size limit).
- Basically in the high level applications using stack, usually we only need to check if the stack is empty. If we call the top() function on an empty stack, an underflow exception occurs.
- The top() function in the STL returns the top element by reference. It is different from the example given in the textbook, where the top() function returns the top element by value.

- We shall first discuss some examples on the uses of stack in algorithm design, and then come back to discuss the internal implementation (*see* ***stack.h***) of the stack ADT.
- One common use of stack is for the simulation of recursion, i.e. converting a recursive algorithm to an equivalent non-recursive algorithm using a stack. This will be discussed in the topic of tree data structure.

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 $ * / −

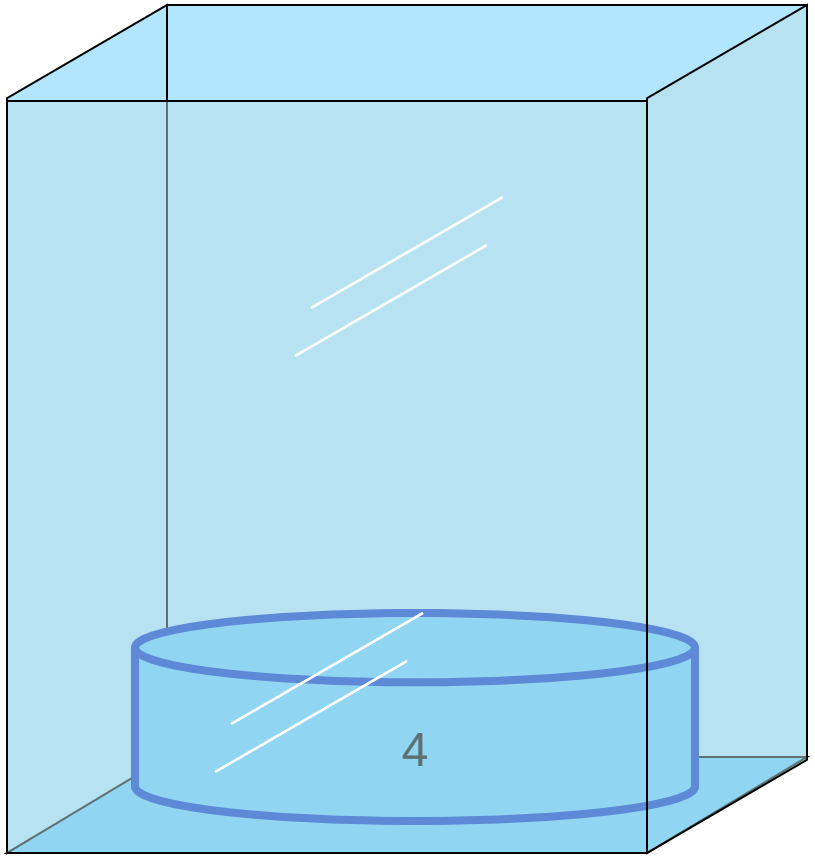
- Read from postfix
  - If input is an operand, push on stack
  - If input is an arithmetic operator
    - pop from stack twice (the two nearest operands)
    - compute their result
    - push the result onto stack

Example

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

Postfix: 4 5 + 7 2 - *

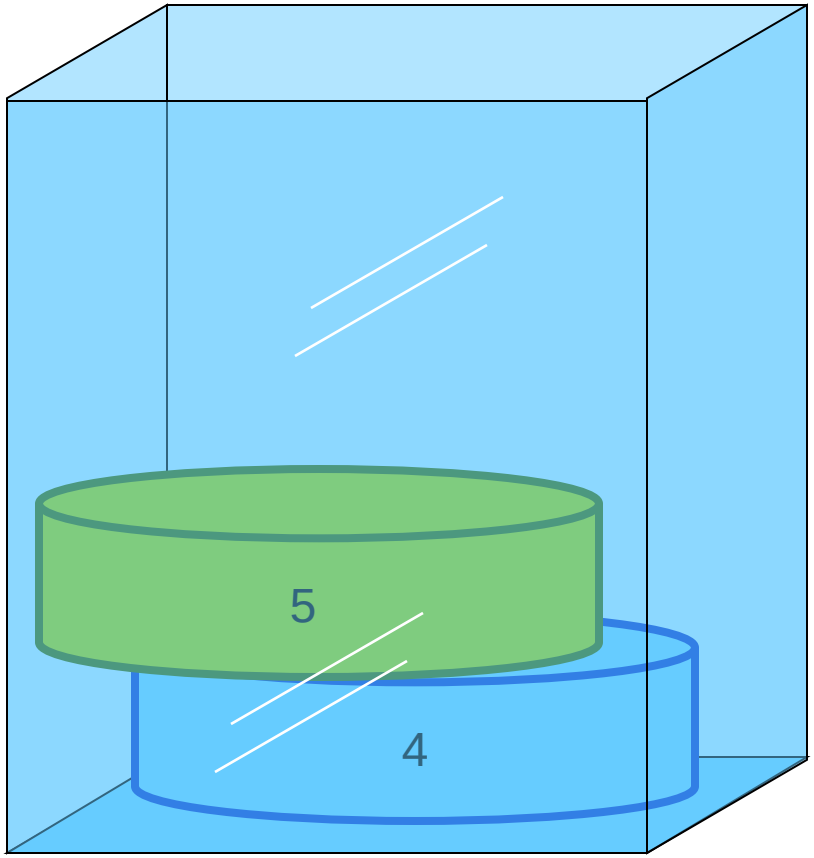
4 5 + 7 2 - *



Step 1: push(4)

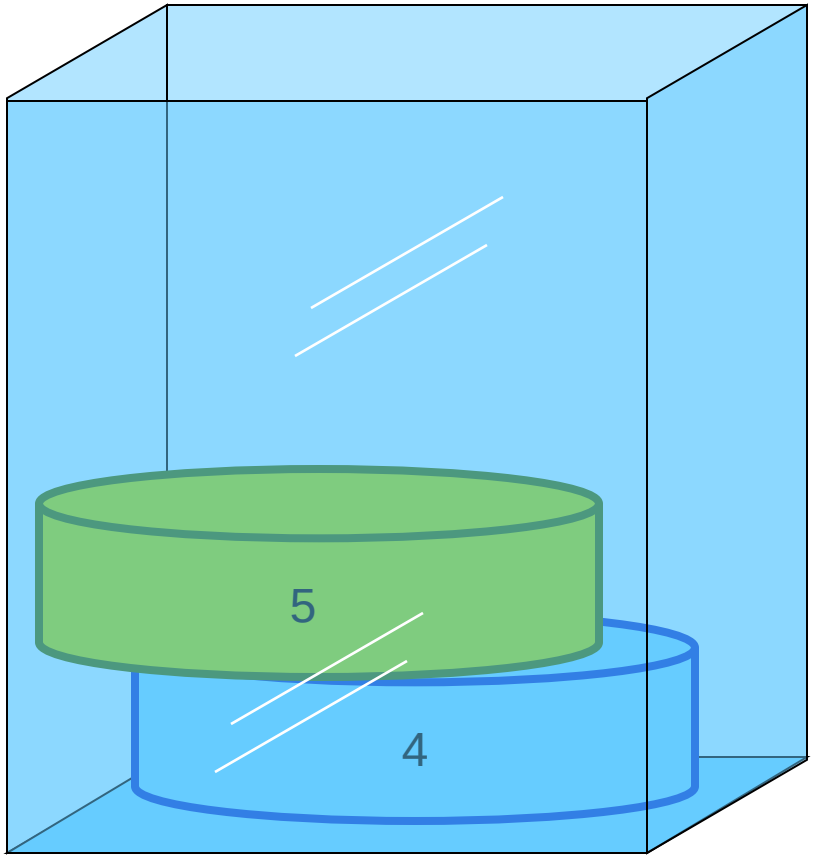
Example

4 5 + 7 2 - *



Step 2: push(5)

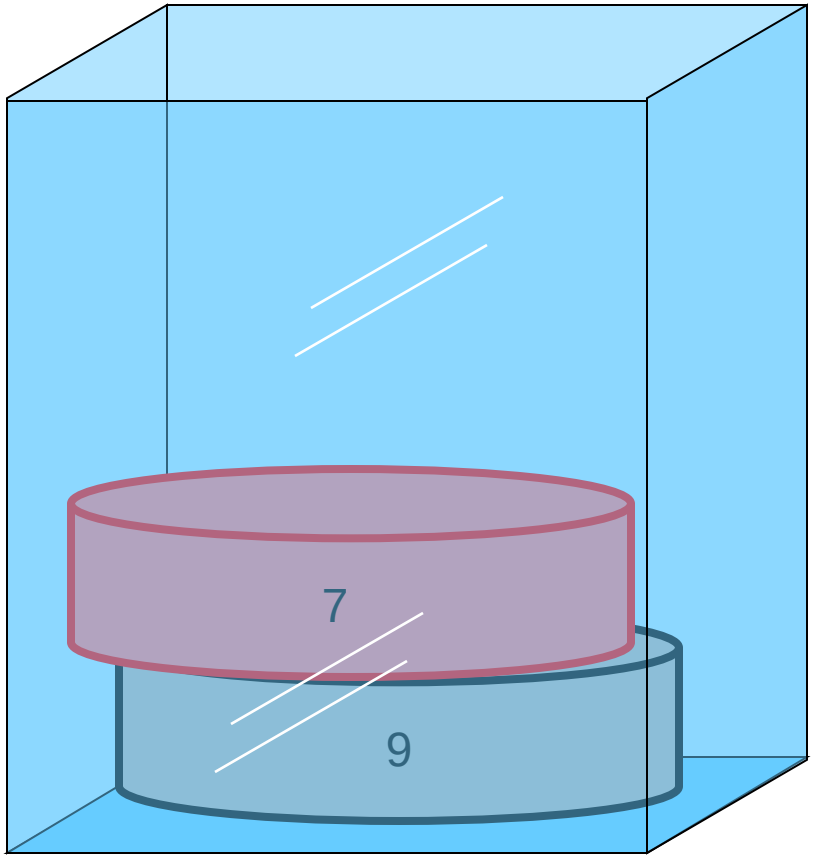
4 5 + 7 2 - *



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

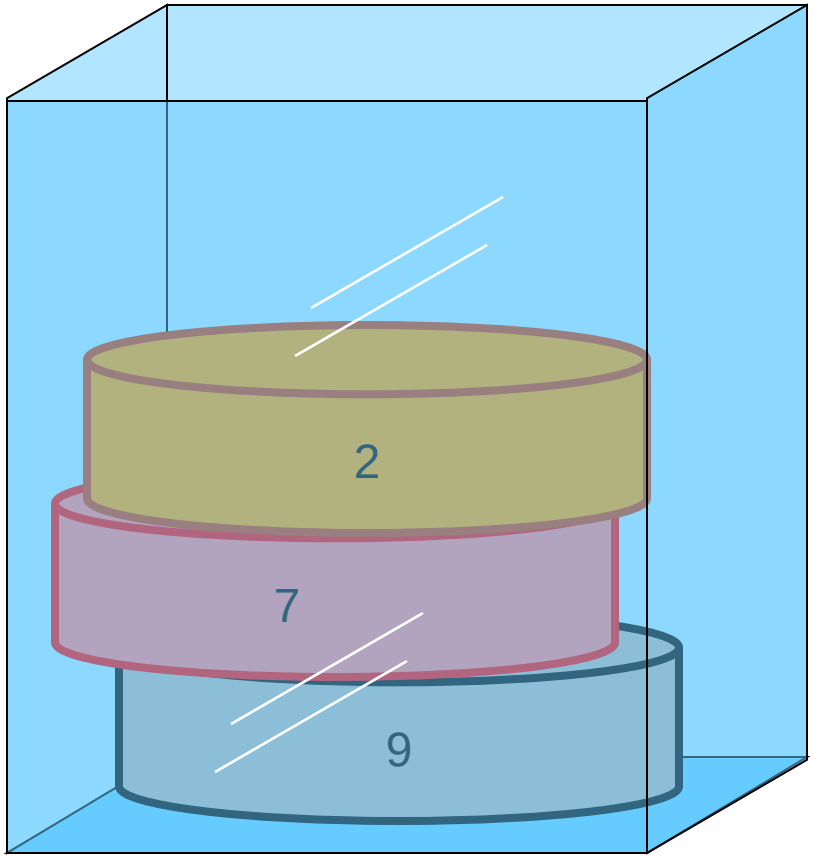
Example

4 5 + 7 2 - *



Step 4: push(7)

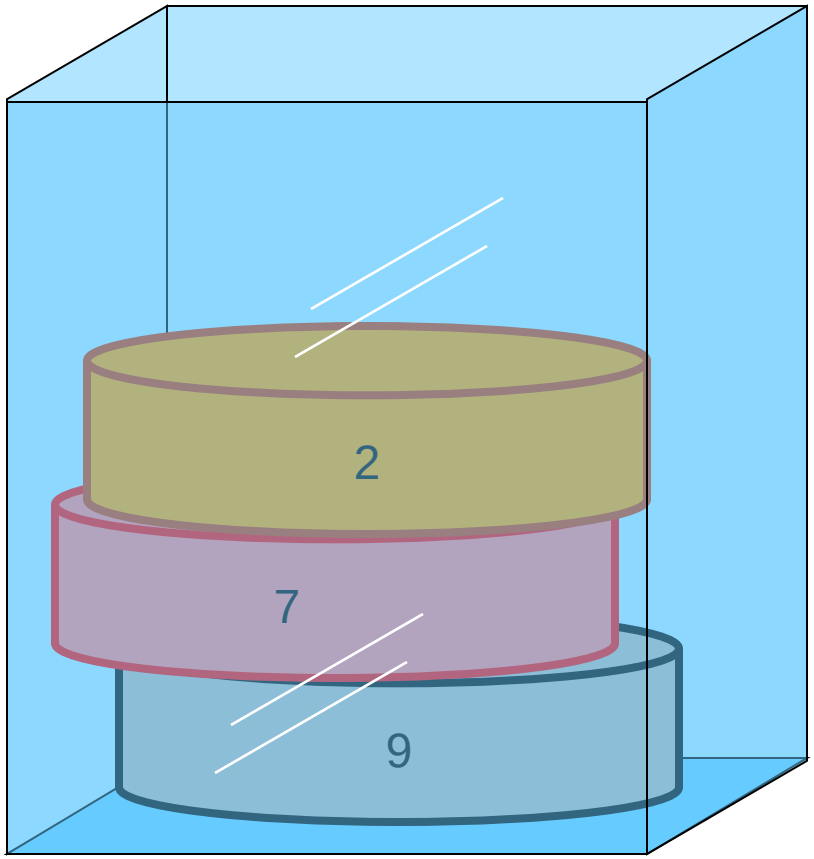
4 5 + 7 2 - *



Step 5: push(2)

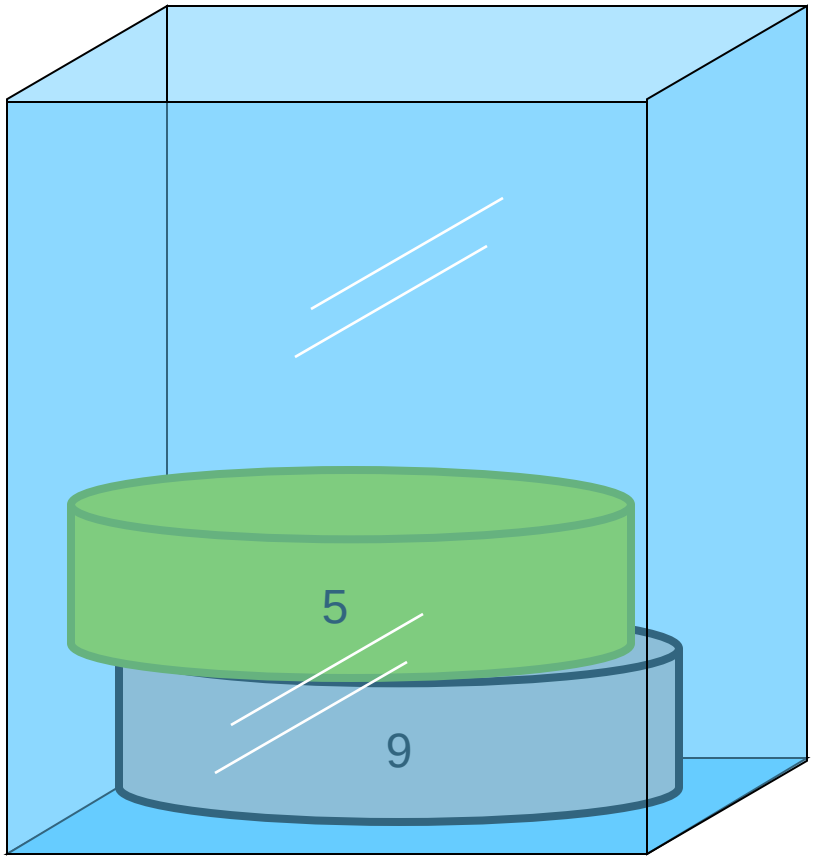
Example

4 5 + 7 2 - *



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

4 5 + 7 2 - *

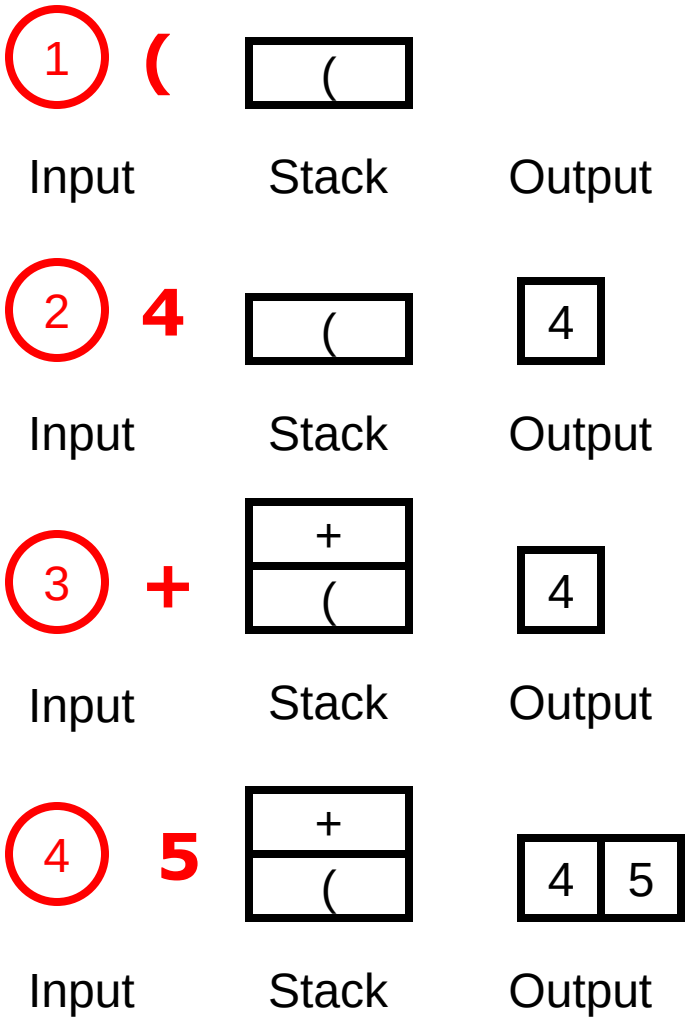


Step 7: pop() twice and compute the result

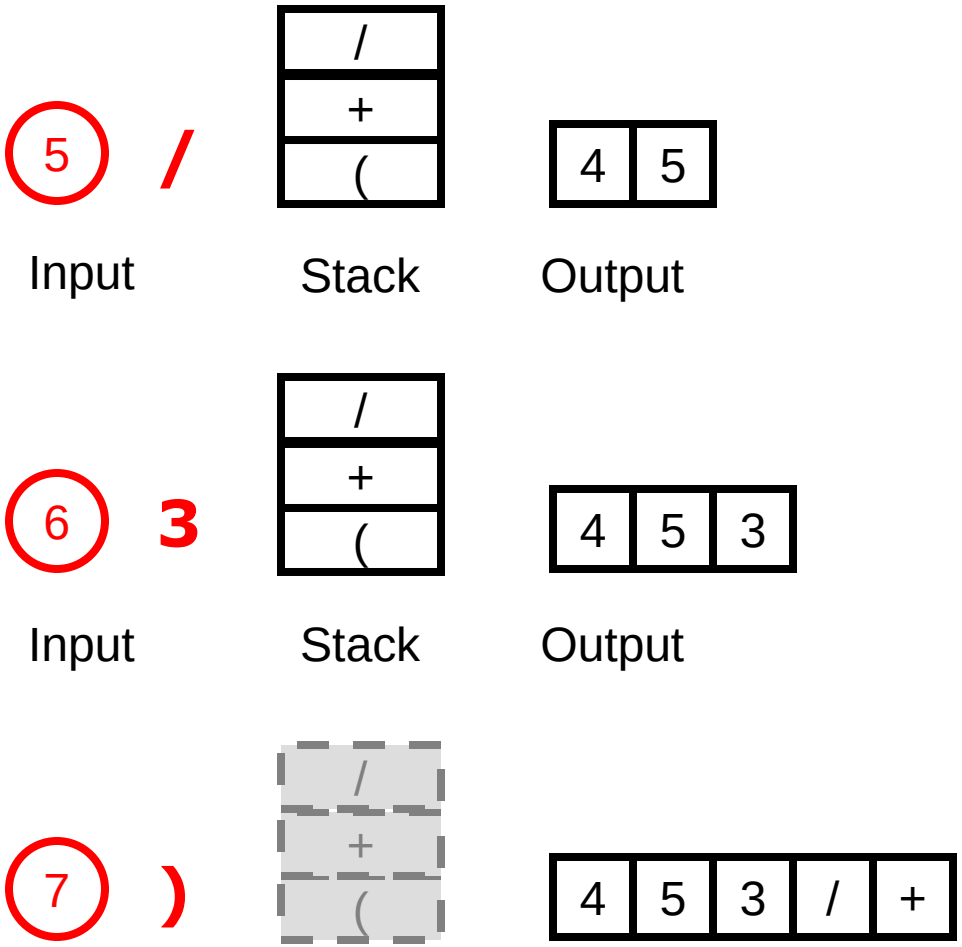
Convert Infix to Postfix

- Read from infix
  - If input is an operand, output it
  - If input is operator (, push it onto stack
  - If input is operator), pop all operators from stack and output them orderly until popping the nearest operator ((but the open-parentheses is not output)
  - If input is an operator (e.g. +, -, *, /)
    - If input has higher precedence than the top element of the stack, then push the input onto stack
    - Otherwise pop to output the top element repeatedly until the input has higher precedence than the top element of the stack, then push the input onto stack
- Pop and output the rest from the stack until the stack is empty

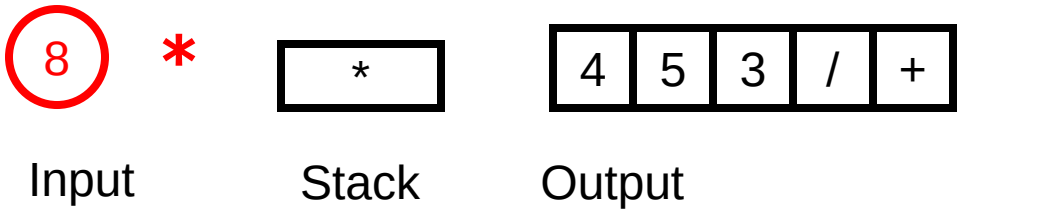
Example: (4+5/3)*(7*2–1)

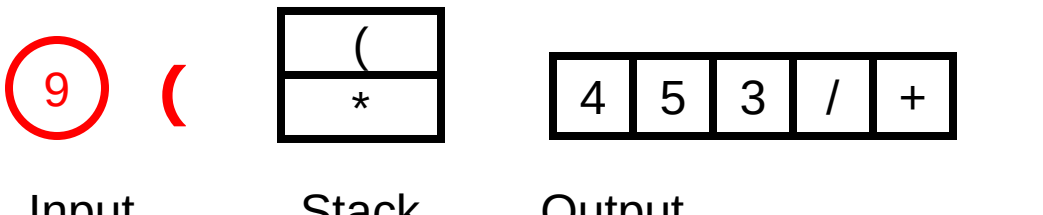


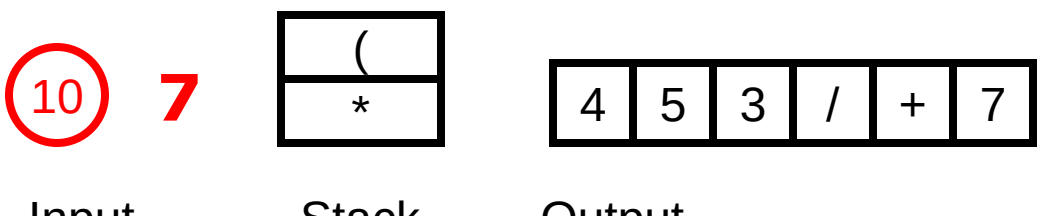
Example: (4+5/3)*(7*2–1)



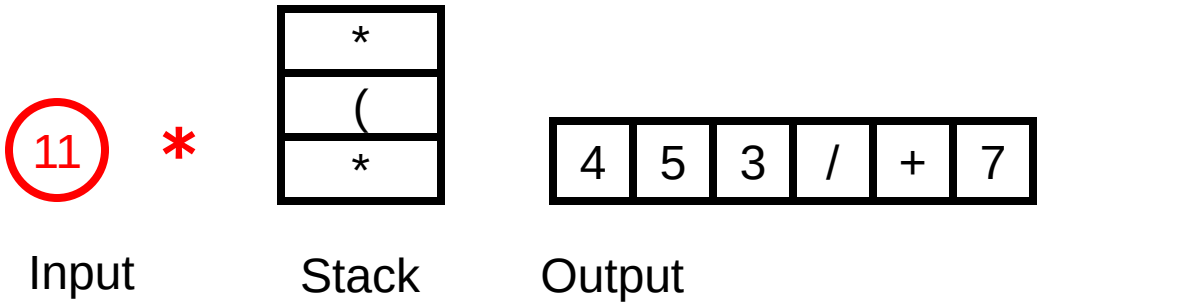
Example: (4+5/3)*(7*2–1)

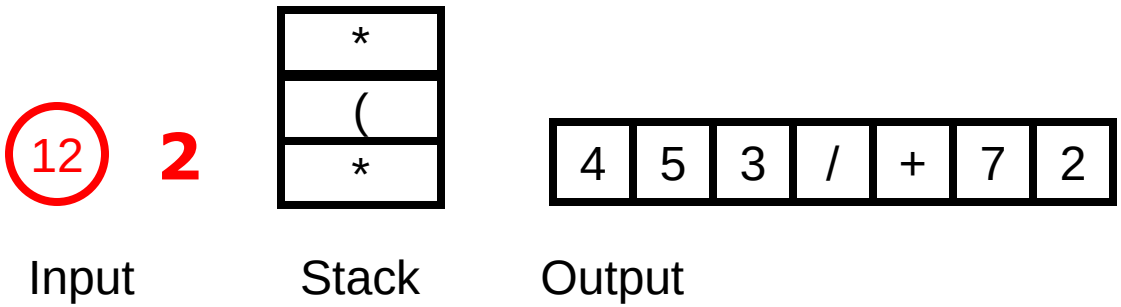


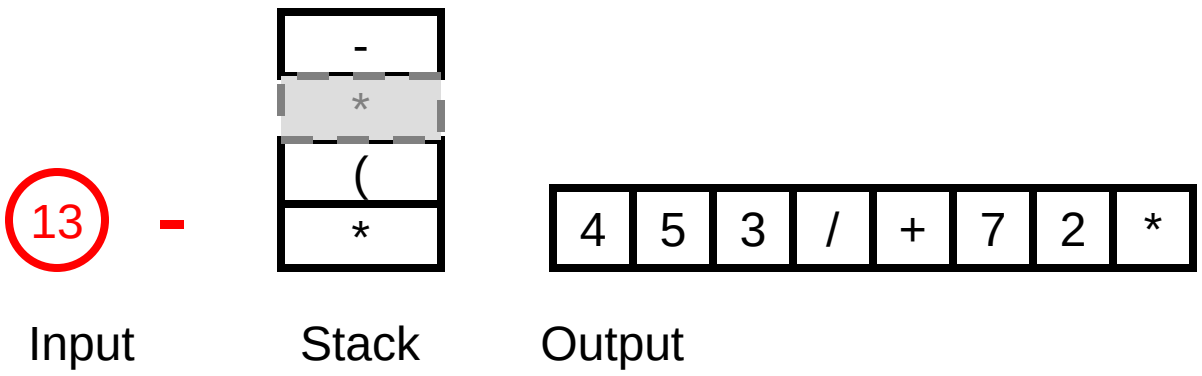




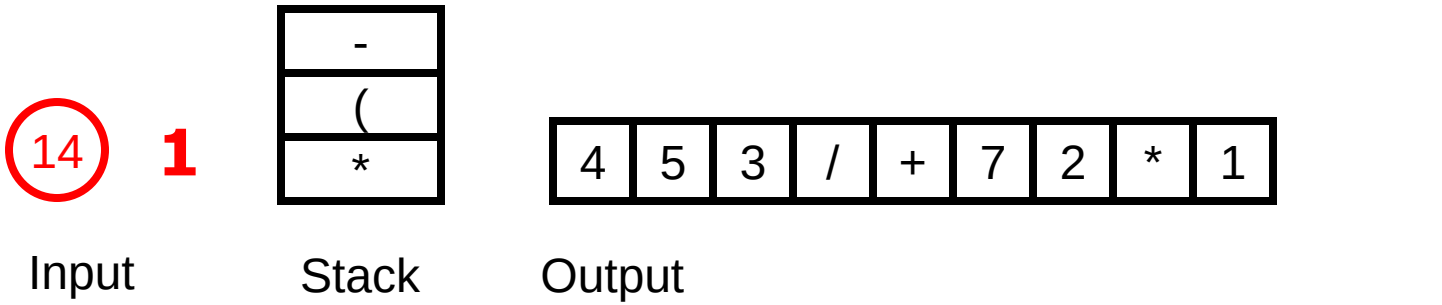
Example: (4+5/3)*(7*2–1)







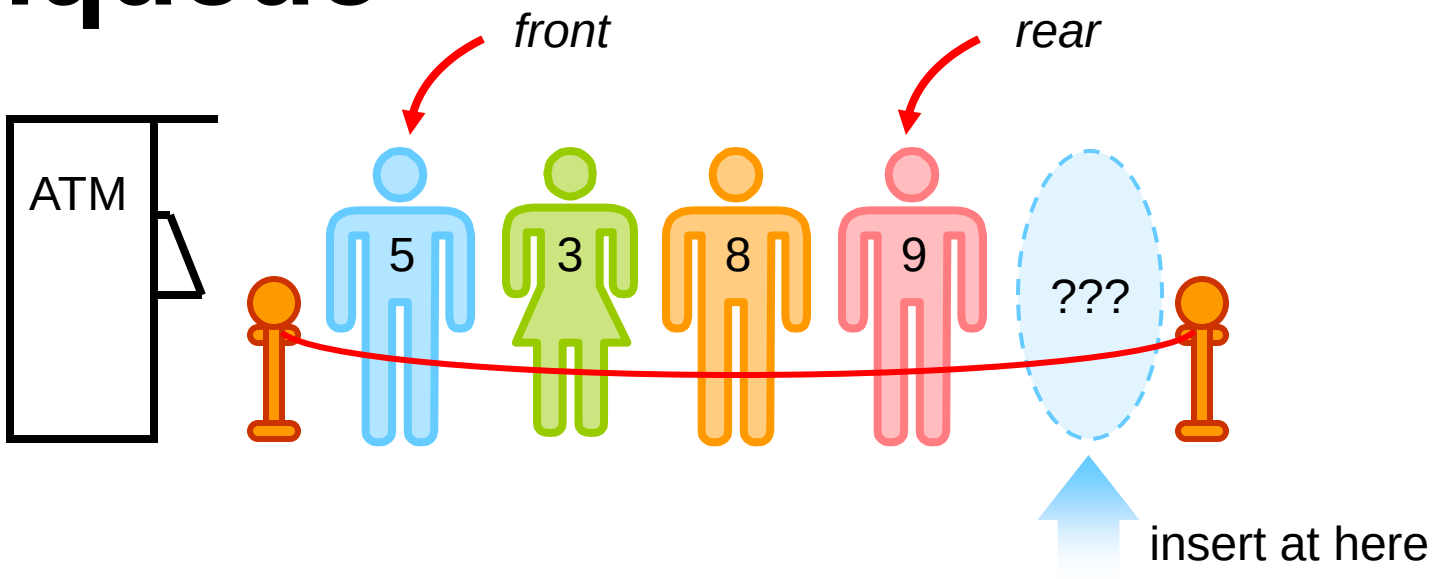
Example: (4+5/3)*(7*2–1)

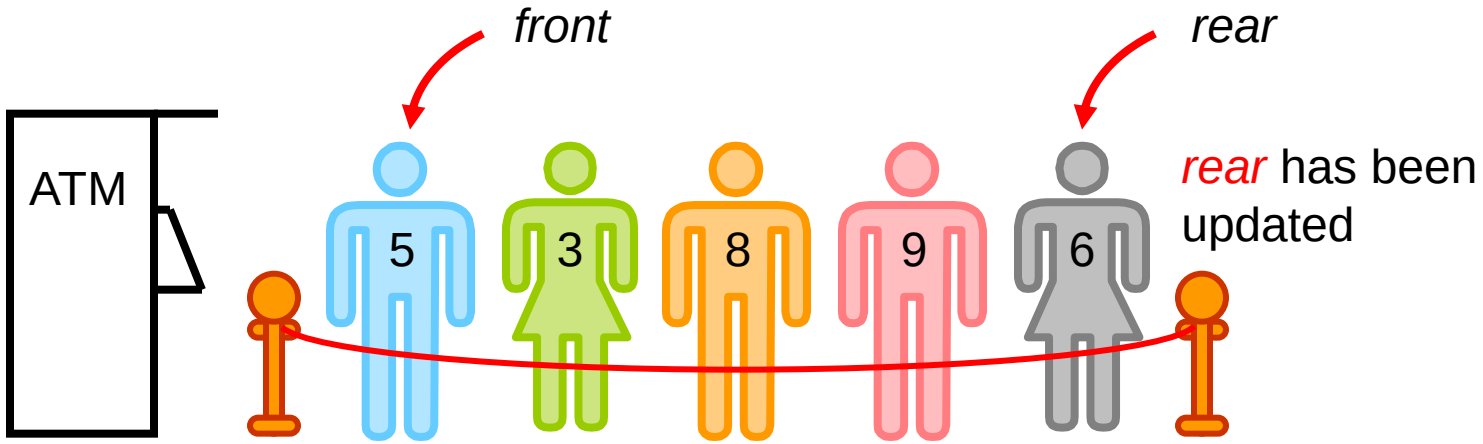


Queue

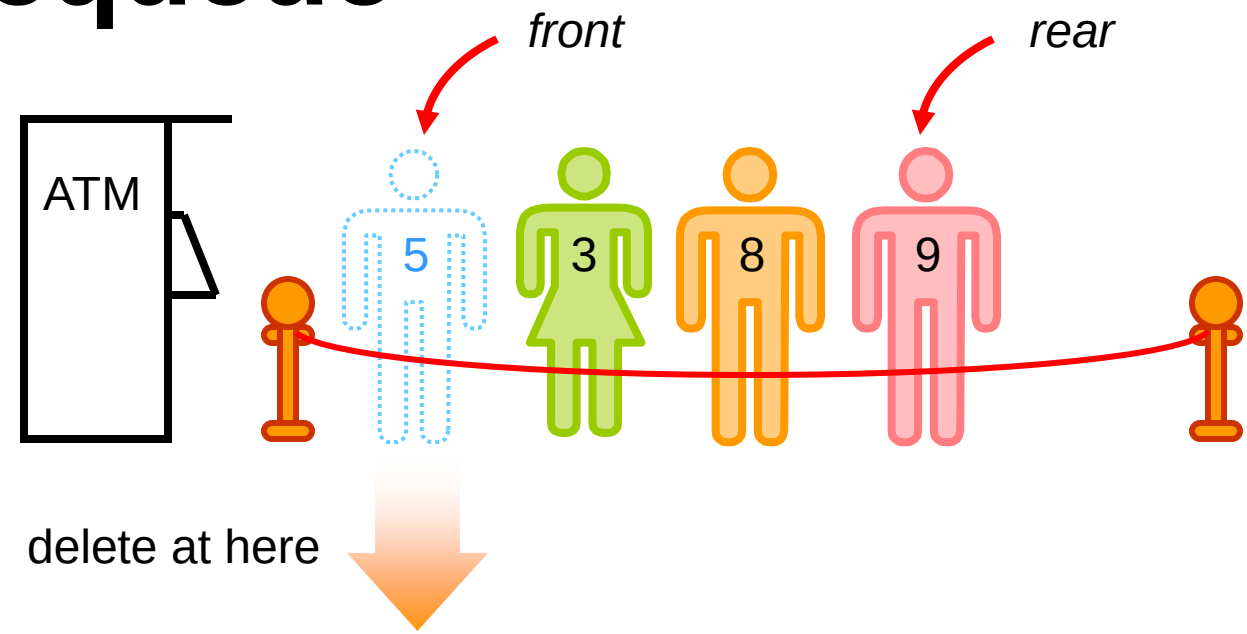
- A first-in-first-out (FIFO) queue is an ordered collection of items from which items may be deleted at one end (called the front) and into which items may be inserted at the other end (called the rear).
- Operations on a queue :
  - **initialize**: initialize the queue to an empty state
  - **size**: 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)

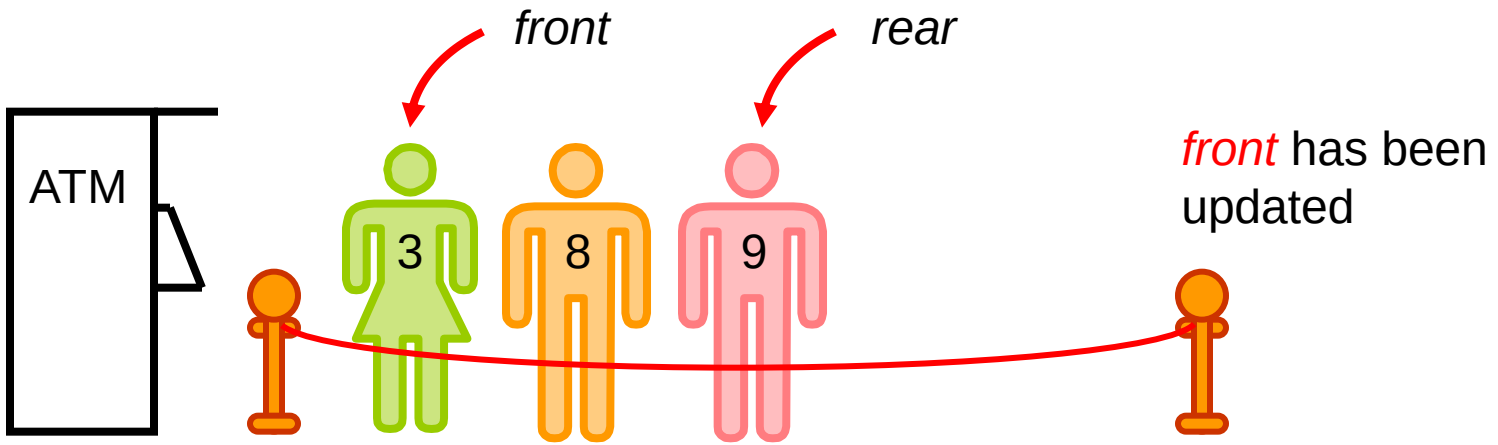
Enqueue


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

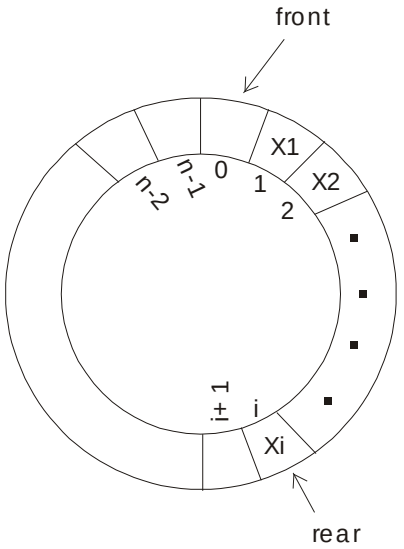

front
rear
ATM
3
8
9
front has been updated

Queue ADT (Abstract Class)


```
template<class Type>
class queueADT
{
public:
   virtual void initialize() = 0;
   //remark: the initialize() function is not part of the
   //        C++ STL. The initialization can be taken care
   //        of by the constructor.

   virtual int size() const = 0;

   virtual bool empty() const = 0;

   virtual Type& front() const = 0;

   virtual void push(const Type& item) = 0;

   virtual void pop() = 0;
}
```

Queue Implementation Using Circular Array

- The program maintains two indexes, front and rear
- If the queue is empty, front == rear.
- If the queue is not empty,
  - rear points to the last element, and
  - front points to the slot before the first element.
- An array of size *n* can hold up to *n*-1 elements.



*(see **queue.h**)*

- Example applications of queue:
  - message buffering in inter-process communications
  - task scheduling in operating system
  - breadth-first search of multi-dimensional data structures, e.g. trees
  - event-driven simulations

Double-Ended Queue

- In the C++ STL, stack and queue are implemented using deque as the default container.
- Deque (pronounced like “deck”) is a double-ended queue. It allows insertion and deletion at both ends.
- The insertion/deletion functions are called
  - push_front
  - push_back
  - pop_front
  - pop_back