Project 1

- Choose either project 1A 1B or 1C
- **Deadline: March 18, 2018**
- Submit the reports and the source code and the reports on Blackboard by 11:59 PM (end of the day). For the source code and reports, I expect one submission per team, but mention the team members clearly in your submission.
- Please stick to the teams that I assigned you with.

Project 1A

- *Using stacks*, write an infix expression parser. Here are a few examples of expressions your program should parse and evaluate:

Expression	Result
1+2*3	7
2+2^2*3	14
1==2	0 //or false if the type is bool
1+3 > 2	1 // or true if the type is bool
(4>=4) && 0	0 //or false if the type is bool
(1+2)*3	9
++++2-5*(3^2)	-41

Technical Requirements

- (Weight: 20%) Your parser should parse an infix expression that supports the following arithmetic and logical operators with the specified precedencies:

Operator	Precedence	Example
! *//logical not*	8	! 1 // = 0 (false)
++ *//prefix increment*	8	++2 //3
-- *//prefix decrement*	8	--2 //1
- *//negative*	8	-1 //-1
^ *//power*	7	2^3 // 8
*, /, % *//arithmetic*	6	6 * 2 // 12
+, - *//arithmetic*	5	6 - 2 //4
>, >=, <, <= *//comparison*	4	6 > 5 // 1(true)
==, != *//equality comparison*	3	6!=5 // 1(true)
&& *//logical and*	2	6>5 && 4>5 //0(false)
|| *//logical OR*	1	1 || 0 //1 (true)

- (Weight: 30%) Parse an expression given in a string format. Your program should be flexible with the given expressions. For instance, 1+2 is the same as 1 + 2. The user should not worry about writing the spaces between operands and operators.
- (Weight: 20%) Your program should check for invalid expressions, and produce meaningful error messages when necessary. Further, the error message should indicate whether the error happened (Only report the first error the program encounters and exits the program).
  Here are a few examples:

What the user enters	Error message
)3+2	Expression can't start with a closing parenthesis @ char: 0
<3+2	Expression can't start with a binary operator @ char: 0
3&&&& 5	Two binary operators in a row @ char 3
15+3 2	Two operands in a row @ char 5
10+ ++<3	A unary operand can't be followed by a binary operator @ char 6
1/0	Division by zero @ char 2

- (Weight: 30%) Evaluate the given expressions efficiently.

Facts and Assumptions

- You may assume that all operands are integers. The result of a comparison is a boolean (e.g. 6==6 is true). However, C++ compiler allows a boolean can be converted to an integer according to this logic: true =1 and false = 0. An integer can be converted to a boolean according to this logic: a number that is equal to 0 is false. Otherwise, it is true.
- You can be inspired by the postfix evaluator as well as the infix to postfix convertor that we have studied in class (the source code is on Blackboard). However, make sure you come up with an efficient algorithm.
- You may just call the evaluate function in the main function. There is no need for getting input from the user or a menu-based system. For instance, this is an example of how the evaluator is used in the main function:

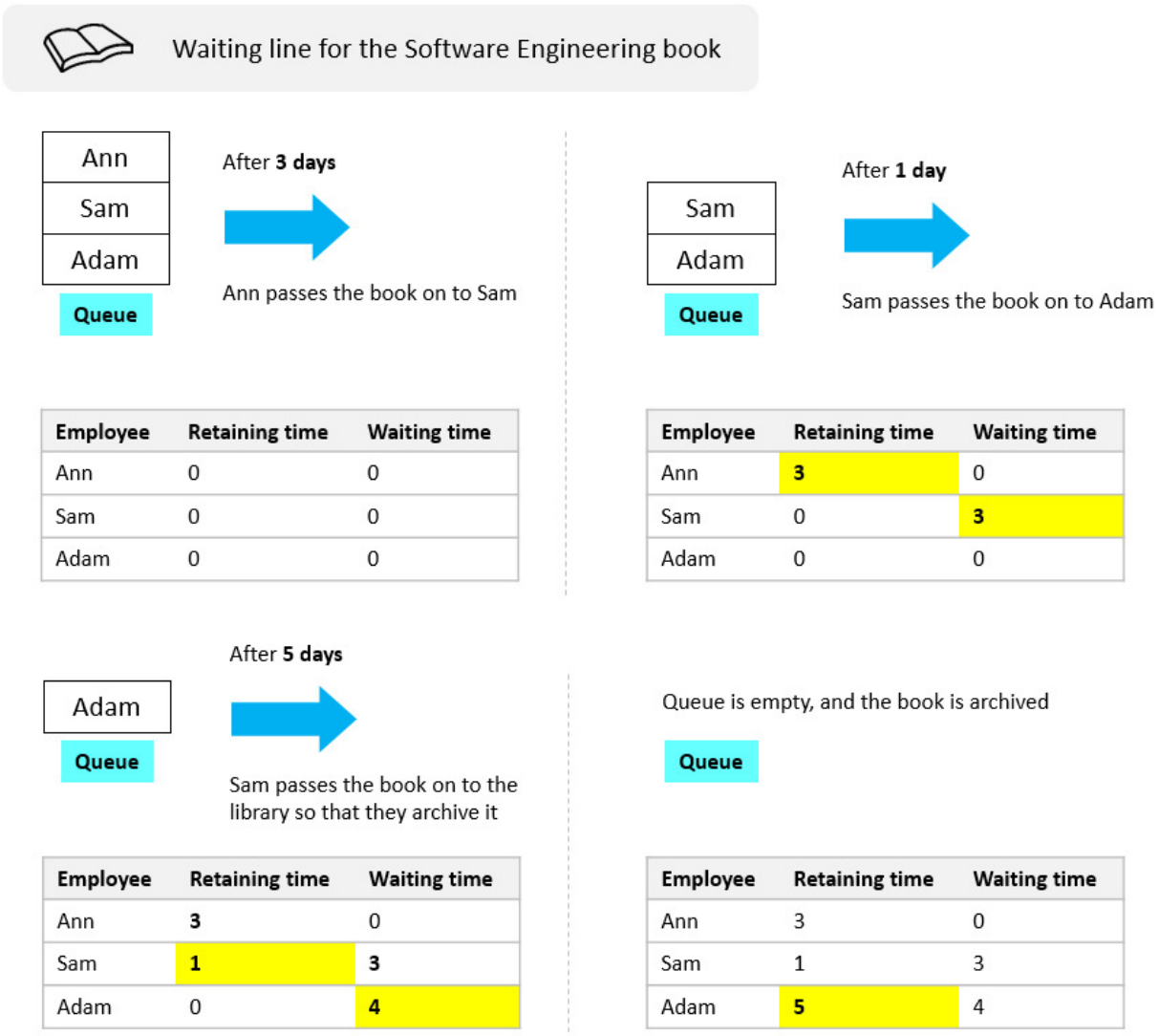
```
int main(){
Evaluator eval;
int result=eval.eval("!!!3+2");

return 0;
}
```

Project 1B

- Using *queues*, design a system that allows the circulation of books to employees.

Technical Requirements

- (Weight: 10%) The system should keep track of two lists: (1) Books to be circulated: these are the books that will be circulated to employees. (2) Archived books: when the last employee on the queue returns the book to the library, the book gets added to the list of archived books.
- (Weight: 5%) The system should keep track of employees: these are the employees that will get the books later on.
- (Weight: 25%) Circulate a book to the employees in the system. The circulation starts on a given date. Further, the system makes a queue of employees that should receive the book. The queue should be prioritized based on two factors: (1) the total waiting time for the employee: How many days the employee waited to get a book since the beginning of the circulation. (2) The total retaining time: How many days the employee retained books.
- (Weight: 20%) Make a data structure (priority queue) that allows the pushing and popping of items. The popped item is the item with the highest priority. The queue should also be updatable whenever an item's priority changes.
- The more the employee waited, the higher the priority. The more she retained a book, the lower the priority. To put it simply, the priority is: *waiting_time – retaining_time.*
  The employee in front of the queue gets the book.
- (Weight: 40%) The system should allow the employee to pass the book on to the next employee on the queue on a given date.
- Passing on the book has the following outcome:
  - ✓ If the employee who is passing on the book is the last in the queue, the book gets archived.
  - ✓ The total retaining time for the employee who passed on the book gets adjusted.
  - ✓ The total waiting time for the employee who got the book gets adjusted.
  - ✓ If there are other queues for other books, and these queues contain the employee who passed on the book and the employee who got the book, then adjust these queues (because the priorities have changed).
  - ✓ See Figure 1 for an illustration.



Employee	Retaining time	Waiting time
Ann	0	0
Sam	0	0
Adam	0	0

Employee	Retaining time	Waiting time
Ann	**3**	0
Sam	0	**3**
Adam	0	0

Employee	Retaining time	Waiting time
Ann	**3**	0
Sam	**1**	**3**
Adam	0	**4**

Employee	Retaining time	Waiting time
Ann	3	0
Sam	1	3
Adam	**5**	4

Figure 1: An illustration of circulating the software engineering book.

Facts and Assumptions

- A book has the following properties: name, circulation start date, circulation end date, archived (whether the book is archived), a queue of employees (the employees who are planned to get the book).
- An employee has the following properties: name, waiting time, and retaining time.
- There is only one copy of each book the system has.
- You can use any source code on Blackboard or any built-in c++ data structures (apart from priority queue).
- You can use the `Date` class I wrote (it has year, month, day components). You can find a project that shows examples of how to use the `Date` class here: https://www.dropbox.com/sh/1bfc5ayty5kl16x/AAD-bO28NQX7RaFQj39ZRQV1a?dl=0
- There is no need for getting input from the user or a menu-based system. For instance, this is an example of how the circulation system (e.g. library) is used in the main function:

```
int main(){
Library library;
library.add_book("Software Engineering");
library.add_book("Chemistry");

library.add_employee("Adam");
library.add_employee("Sam");
library.add_employee("Ann");

library.circulate_book("Chemistry", Date(2015, 3, 1, DateFormat::US));
library.circulate_book("Software Engineering", Date(2015, 4, 1, DateFormat::US));
library.pass_on("Chemistry", Date(2015, 3, 5, DateFormat::US)); //tell the next employee to pass the book on March 5, 2015
library.pass_on("Chemistry", Date(2015, 3, 7, DateFormat::US));
library.pass_on("Chemistry", Date(2015, 3, 15, DateFormat::US)); //at this point in time, the system will archive the chemistry book.



library.pass_on("Software Engineering", Date(2015, 4, 5, DateFormat::US));
library.pass_on("Software Engineering", Date(2015, 4, 10, DateFormat::US));
library.pass_on("Software Engineering", Date(2015, 4, 15, DateFormat::US));
}
```

Project 1C (Open Project)

- Choose some of the data structures we studied or make your own to simulate how an elevator handles and keeps track of requests made by users.
- Here is how an elevator works: A user pushes either the up or down button to request the elevator. Once the elevator arrives, the user gets in and specifies his/her destination floor. Meanwhile, other users outside or inside the elevator make requests as well. The elevator stores the requests and handles them efficiently.
- A simple algorithm for handling requests works like this: all requests users make are stored. The elevator prioritizes the requests that are on the way where it's going, but also based on a first come first served principle. It is up to you how you design the algorithm. As a recommendation, observe how an elevator works (e.g. the elevator we have at FH) and simulate how it works.

Technical Requirements

- (Weight: 60%) Write an efficient function that handles elevator requests. The function should result in a low waiting time for users. Make sure your function accommodates a single-elevator system and a multi-elevator system.
- (Weight: 40%) Write a simulation function that calculates the waiting time for users. Your simulation algorithm could randomly generate users that come at random times and make random requests. Alternatively, make your own scenarios and test the waiting time for those scenarios. For inspiration, see how we simulated the airline passenger serving algorithm.

Discussion

- Since this is an open project, the grading will largely depend on the thinking process and design decisions. Hence, you are welcome to discuss with me how you will approach the problem.