Scheduler

I. Problem Description

Stanford students manage many responsibilities on a daily basis. From school to our own personal lives, we all need systems to help us manage our daily tasks. Software like Google Calendar or Apple Calendar helps users keep track of their daily responsibilities. In this problem, you will implement your own calendar system using classes and abstract data types.

You will implement the **Scheduler** class which manages a collection of events using a data structure of your choice. **Scheduler** should be able to fully remove and add elements, as well as collect all events that share the same date. You are free to add any private member variables or methods that you think are necessary.

We have implemented the **Event** struct for you.

```
struct Event {
   string name;
   int month;
   int date;
   int year;
};
```

Your class should have the following public methods:

Method	**Description**
`addEvent(Event event)`	This function takes in an event, adds an event to a schedule, and ensures that no duplicates are added.
`isEqual(Event e1, Event e2)`	This function takes in two events and checks if the two events share the same date.
`removeEvent(Event event)`	This function takes in an event and removes an event from the schedule.
`getCurrentSchedule(Event curDate)`	This function takes in a date and returns a collection of events that share the same date.

printCurrentSchedule(Event curDate)	This function takes in a date and prints all the events that share the same date.

II. Solution 1: Implementing Scheduler through a Vector

```
/* This function takes in an event. This function adds an event to
a scheduler and ensures that no duplicates are added.*/
void Scheduler::addEvent(Event event) {
   for (int i = 0; i < schedule.size(); i++) {
       if (!isEqual(schedule[i], event) && schedule[i].name !=
event.name) {
           schedule.add(event);
       }
   }
}

/* This function takes in two events. This function checks if two
events are equal.*/
bool Scheduler::isEqual(Event e1, Event e2) {
   if (e1.date == e2.date && e1.month == e2.month && e1.year ==
e2.year) {
       return true;
   }
   return false;
}

/* This function takes in an event. This function removes an event
from the schedule.*/
void Scheduler::removeEvent(Event event) {
   for (int i = 0; i < schedule.size(); i++) {
       if (isEqual(schedule[i], event) && schedule[i].name ==
event.name) {
           schedule.remove(i);
       }
   }
}
/* This function takes in a date. The function returns a vector of
events that share the same date.*/
Vector<Event> Scheduler::getCurrentSchedule(Event curDate) {
   Vector<Event> cur_schedule;
   for (int i = 0; i < schedule.size(); i++) {
       if (isEqual(curDate, schedule[i])) {
            cur_schedule.add(schedule[i]);
       }
   }
   return cur_schedule;
}
```

```
/* This function takes in a date. This function prints all the
events that share the same date,*/
void Scheduler::printCurrentSchedule(Event curDate) {
   Vector<Event> cur_schedule = getCurrentSchedule(curDate);
   cout << "Here is today's schedule:" << endl;
   for (int i = 0; i < cur_schedule.size(); i++) {
       cout << cur_schedule[i].name + ", " +
integerToString(cur_schedule[i].month) + "/" +
       integerToString(cur_schedule[i].date) + "/" +
integerToString(cur_schedule[i].year) << endl;
   }
   cout << "Enjoy your day!" << endl;
}
```

Solution 1 uses a Vector of Events to store the schedule. To access each element, the methods use a “for i in range” loop. Built-in features of the Vector class make operations such as removing events from the schedule easier for the student to implement. Additionally, being able to loop through the elements without altering its position allows you to access individual elements easily.

Function	Big-O
addEvent	O(n)
isEqual	O(1)
removeEvent	O(n^2)
getCurrentSchedule	O(n^2)
printCurrentSchedule	O(n^2)

III. Solution 2: Implementing Scheduler through a Queue

```
/* This function takes in an event. This function adds an event to
a scheduler and ensures that no duplicates are added.*/
void Scheduler::addEvent(Event event) {
   // checking for duplicates
   removeEvent(event);
   schedule.enqueue(event);
}

/* This function takes in two events. This function checks if two
events are equal.*/
```

```
bool Scheduler::isEqual(Event e1, Event e2) {
   if (e1.date == e2.date && e1.month == e2.month && e1.year ==
e2.year) {
       return true;
   }
   return false;
}

/* This function takes in an event. This function removes an event
from the schedule.*/
void Scheduler::removeEvent(Event event) {
   Queue<Event> copy = schedule;
   while (!copy.isEmpty()) {
       Event cur = copy.dequeue();
       if (!isEqual(event, cur) && event.name != cur.name) {
           copy.enqueue(cur);
       }
   }
   schedule = copy;
}

/* This function takes in a date. The function returns a vector of
events that share the same date.*/
Queue<Event> Scheduler::getCurrentSchedule(Event curDate) {
   Queue<Event> cur_schedule;
   Queue<Event> copy = schedule;
   while (!copy.isEmpty()) {
       Event event = copy.dequeue();
       if (isEqual(curDate, event)) {
           cur_schedule.enqueue(event);
       }
   }
   return cur_schedule;
}

/* This function takes in a date. This function prints all the
events that share the same date,*/
void Scheduler::printCurrentSchedule(Event curDate) {
   Queue<Event> cur_schedule = getCurrentSchedule(curDate);
   cout << "Here is today's schedule:" << endl;
   while (cur_schedule.isEmpty()){
       Event cur = cur_schedule.dequeue();
       cout << cur.name + ", " + integerToString(cur.month) + "/" +
       integerToString(cur.date) + "/" + integerToString(cur.year)
<< endl;
   }
   cout << "Enjoy your day!" << endl;
}
```

Solution 2 uses a Queue of Events to store the schedule. To access each element, the student must use a while loop that ends when the Queue is empty and dequeues an element in the beginning of the loop. Additionally, it is much more difficult to access elements in the middle of the queue. It requires displacing the elements and creating a copy to retain the original elements. Despite these implementation drawbacks, the queue is slightly faster to us in Big-O analysis.

Function	Big-O
addEvent	O(n)
isEqual	O(1)
removeEvent	O(n)
getCurrentSchedule	O(n)
printCurrentSchedule	O(n)

IV. Test Cases

```
STUDENT_TEST("addEvent, regular") {
   Event event = {"Birthday", 5, 23, 2002};
   Scheduler planner;
   planner.addEvent(event);
   Queue<Event> result;
   Queue<Event> cur_schedule = planner.getCurrentSchedule(event);
   result.enqueue(event);

   while (!result.isEmpty()) {
       Event event1 = result.dequeue();
       Event event2 = cur_schedule.dequeue();
       EXPECT(planner.isEqual(event1, event2) && event1.name ==
event2.name);
   }

}

STUDENT_TEST("addEvent, duplicate event") {
   Event event = {"Birthday", 5, 23, 2002};
   Scheduler planner;
   planner.addEvent(event);
   planner.addEvent(event);
   Queue<Event> result;
   Queue<Event> cur_schedule = planner.getCurrentSchedule(event);
   result.enqueue(event);
```

```
    while (!result.isEmpty()) {
        Event event1 = result.dequeue();
        Event event2 = cur_schedule.dequeue();
        EXPECT(planner.isEqual(event1, event2) && event1.name ==
event2.name);
    }

}

STUDENT_TEST("isEqual, true example") {
    Event event1 = {"Birthday", 5, 23, 2002};
    Event event2 = {"Birthday", 5, 23, 2002};
    Scheduler planner;
    EXPECT(planner.isEqual(event1, event2));
}

STUDENT_TEST("isEqual, false example") {
    Event event1 = {"Birthday", 5, 23, 2002};
    Event event2 = {"Mom Birthday", 5, 24, 2002};
    Scheduler planner;
    EXPECT(!planner.isEqual(event1, event2));
}

STUDENT_TEST("removeEvent") {
    Event birthday = {"Birthday", 5, 23, 2002};
    Event mothers_day = {"Mother's Day", 5, 15, 2021};
    Scheduler planner;
    planner.addEvent(birthday);
    planner.addEvent(mothers_day);
    planner.removeEvent(mothers_day);
    Queue<Event> result = {};
    Queue<Event> cur_schedule =
planner.getCurrentSchedule(mothers_day);

    while (!result.isEmpty()) {
        Event event1 = result.dequeue();
        Event event2 = cur_schedule.dequeue();
        EXPECT(!planner.isEqual(event1, event2) && event1.name ==
event2.name);
    }
}

STUDENT_TEST("printCurrentSchedule") {
    Event birthday = {"Birthday", 5, 23, 2021};
    Event math_test = {"Math Test", 5, 23, 2021};
    Event mothers_day = {"Mother's Day", 5, 15, 2021};
    Event cur_date = {"Cur Date", 5, 23, 2021};
    Scheduler planner;
    planner.addEvent(birthday);
    planner.addEvent(mothers_day);
```

```
    planner.addEvent(math_test);
    planner.printCurrentSchedule(cur_date);
}
```

V. Problem Motivation

Conceptual

This problem is a great way to demonstrate mastery of classes and implementing abstract data structures. This problem requires an understanding of:

- Abstract data structures and the tradeoffs between different data structures
- How to define and access member functions and variables in a class
- How to loop through elements of a data structure
- The ability to manipulate nested data structures
- Passing data structures by reference

Students will work through navigating and interacting with both public and private member functions and variables. Students will also have to determine the proper data structure to use to manage a Schedule. This problem combines and tests a range of skills learned in CS106B. It is great for assessing students' ability to apply different concepts learned throughout the course of the class.

Personal

I was interested in this project because I was reflecting on how busy I was in the final weeks of the quarter and how I rely on programs like Google Calendar to manage my time, such as my class times, gym appointments, and club meetings. I thought it would be an interesting problem to implement and a relatable one for Stanford students. Additionally, I often struggle with understanding when to use certain data structures. While I understand how to use them, I find knowing when to apply them very challenging. This problem helped me refine my understanding of ADTs, because, through brainstorming this problem, I was able to determine which data structure made this possible to implement. Other structures I explored were linked lists, priority queues, and maps. I learned that just because it is possible to implement a program with a certain data structure doesn’t mean it is the smartest choice.

VI. Concept mastery, common misconceptions

Common challenges students can run into include:

- Ensuring that no duplicates are added to the data structure
- Difficulty choosing an efficient data structure
- Understanding the Event struct
- Writing test cases

To ensure that no duplicates are added, the student must loop through all the elements of the data structure and compare whether the new event is equal to an existing one. It is difficult to implement this logic as it requires use of the function isEqual to compare the two elements. When working with a Queue it can be especially challenging to find a way to remove an element while retaining the other elements.

It may be challenging for a student to find an efficient data structure to use especially if the student has less familiarity with less common structures like Stacks, Queues, and Maps. Students may feel averse towards data structures that they are less comfortable with which can cause them to have a difficult time writing this assignment.

The Event struct is provided starter code for the student. The student may run into challenges writing test cases because the Event struct is a new collection. QT Creator raises many errors when trying to compare two events. It is important to access the data within the struct or utilize an existing function like isEqual to compare two elements and write effective test cases.

Common bugs that arise are mostly related to comparing two Events.