Ceng213 - Data Structures
Programming Assignment 2
Airline Reservation System Implementation via Binary Search Trees and Queues

Fall 2021

1 Objectives

In this programming assignment, you are first expected to implement a *binary search tree* data structure, in which each node will contain the data and two pointers to the root nodes of its left and right subtrees. The binary search tree data structure will include a single pointer that points to the root node of the binary search tree. The details of the structure are explained further in the following sections. Then, you will use this specialized binary search tree structure to implement an *airline reservation system*.

Keywords: *C++, Data Structures, Binary Search Trees, Queues, Airline Reservation System Implementation*

2 Binary Search Tree Implementation (70 pts)

The binary search tree data structure used in this assignment is implemented as the class template `BST` with the template argument `T`, which is used as the type of the data stored in the nodes. The node of the binary search tree is implemented as the class template `BSTNode` with the template argument `T`, which is the type of the data stored in nodes. `BSTNode` class is the basic building block of the `BST` class. `BST` class has a single `BSTNode` pointer in its private data field (namely `root`) which points to the root node of the binary search tree.

The `BST` class has its definition and implementation in *BST.h* file and the `BSTNode` class has its in *BSTNode.h* file.

2.1 `BSTNode`

`BSTNode` class represents nodes that constitute binary search trees. A `BSTNode` keeps two pointers (namely `left` and `right`) to the root nodes of its left and right subtrees, and a data variable of type `T` (namely `data`) to hold the data. The class has two constructors and the overloaded output operator. They are already implemented for you. You should not change anything in file *BSTNode.h*.

2.2 `BST`

`BST` class implements a binary search tree data structure with the `root` pointer. Previously, data members of `BST` class have been briefly described. Their use will be elaborated in the context of utility functions discussed in the following subsections. You must provide implementations for the following public interface methods that have been declared under indicated portions of *BST.h* file.

2.2.1 `BST();`

This is the default constructor. You should make necessary initializations in this function.

2.2.2 `BST(const BST<T> &obj);`

This is the copy constructor. You should make necessary initializations, create new nodes by copying the nodes in the given `obj`, and insert those new nodes into the binary search tree. The structure among the nodes in the given `obj` should also be copied to the binary search tree.

2.2.3 `~BST();`

This is the destructor. You should deallocate all the memory that you were allocated before.

2.2.4 `BSTNode<T> *getRoot() const;`

This function should return a pointer to the root node of the binary search tree. If the binary search tree is empty, it should return `NULL`.

2.2.5 `bool isEmpty() const;`

This function should return `true` if the binary search tree is empty (i.e., there exists no nodes in the binary search tree). If the binary search tree is not empty, it should return `false`.

2.2.6 `bool contains(BSTNode<T> *node) const;`

This function should return `true` if the binary search tree contains the given `node` (i.e., any `root`/`left`/`right` in the binary search tree matches with `node`). Otherwise, it should return `false`.

2.2.7 `void insert(const T &data);`

You should create a new node with the given `data` and insert it at the appropriate location in the binary search tree. Don't forget to make necessary pointer modifications in the tree. ***You will not be asked to insert duplicated elements into the binary search tree.***

2.2.8 `void remove(const T &data);`

You should remove the node with the given `data` from the binary search tree. Don't forget to make necessary pointer modifications in the tree. If there exists no such node in the binary search tree, do nothing. ***There will be no duplicated elements in the binary search tree.***

2.2.9 `void removeAllNodes();`

You should remove all nodes of the binary search tree so that the binary search tree becomes empty. Don't forget to make necessary pointer modifications in the tree.

2.2.10 `BSTNode<T> *search(const T &data) const;`

You should search the binary search tree for the node that has the same data with the given `data` and return a pointer to that node. You can use the `operator==` to compare two `T` objects. If there exists no such node in the binary search tree, you should return `NULL`. ***There will be no duplicated elements in the binary search tree.***

2.2.11 `BSTNode<T> *getSuccessor(BSTNode<T> *node, TraversalPlan tp) const;`

Given a traversal plan (`inorder`, `preorder` or `postorder`) as parameter (namely `tp`), this function should return a pointer to the successor node of the given `node` with respect to the given traversal plan. `TraversalPlan` is an enumerated type defined in *BST.h* file with values `inorder`, `preorder` and `postorder`, which are the possible methods of traversals in binary search trees. If there exists no successor node (i.e., the given `node` is the last node), you should return `NULL`.

2.2.12 `void print(TraversalPlan tp=inorder) const;`

Given a traversal plan (`inorder`, `preorder` or `postorder`) as parameter (namely `tp`), this function prints the binary search tree by traversing the nodes accordingly. Code for the `inorder` printing is already given. You should complete this function for `preorder` and `postorder` printing. You should follow the printing format used for the given `inorder` printing to implement others.

2.2.13 `BST<T> &operator=(const BST<T> &rhs);`

This is the overloaded assignment operator. You should remove all nodes of the binary search tree and then, you should create new nodes by copying the nodes in the given `rhs` and insert those new nodes into the binary search tree. The structure among the nodes in given `rhs` should also be copied to the binary search tree.

3 Airline Reservation System Implementation (30 pts)

The airline reservation system in this assignment is implemented as the class `AirlineReservationSystem`. `AirlineReservationSy` class has two `BST` objects in its private data field (namely `passengers` and `flights`) with the types `Passenger` and `Flight`, respectively. These two `BST` objects keep the passengers and flights in the airline reservation system. `Passenger` class represents the passengers in the airline reservation system. `Ticket` class represent the tickets issued for the passengers by the airline. `Flight` class represents the flights by the airline. Tickets of a flight are kept in the flight's corresponding `Flight` object. Moreover, `AirlineReservationSystem` class keeps a queue of free ticket requests by the passengers in

the airline reservation system (namely `freeTicketRequests`).

The `AirlineReservationSystem`, `Passenger`, `Ticket` and `Flight` classes has their definitions in *AirlineReservationSystem.h*, *Passenger.h*, *Ticket.h* and *Flight.h* files and their implementations in *AirlineReservationSystem.cpp*, *Passenger.cpp*, *Ticket.cpp* and *Flight.cpp* files, respectively.

3.1 Passenger

`Passenger` objects keep `firstname` and `lastname` variables of type `std::string` to hold the data related with the passengers in the airline reservation system. In this assignment, you may assume that no two passengers will have the same fullname (i.e., *firstname + lastname*). Most of the functions of `Passenger` class are already implemented for you. In *Passenger.cpp* file, you need to provide implementation for the following functions declared under *Passenger.h* header to complete the assignment. You should not change anything in file *Passenger.h*.

3.1.1 `bool operator<(const Passenger &rhs) const;`

This is the overloaded less than comparison operator. First, you should lexicographically compare the `lastname` values of this passenger and the given passenger (`rhs`). If the `lastname` values are the same, you should lexicographically compare the `firstname` values of the passengers. If the compared value of this passenger is less than the given passenger's corresponding value, return `true`. Otherwise, return `false`.

3.2 Ticket

`Ticket` objects keep `ticketId` variable of type `int`, `passenger` variable of type `Passenger *`, `flight` variable of type `Flight *`, and `ticketType` variable of type `TicketType` to hold the data related with the tickets in the airline reservation system. `TicketType` is an enumerated type defined in *Ticket.h* file with values `economy` and `business`, which are the classes for the tickets of the passengers. `Passenger` and `Flight` pointers in a `Ticket` are pointers to the `Passenger` and `Flight` objects stored in the binary search trees in `AirlineReservationSystem` class. All of the functions of `Ticket` class are already implemented for you. You should not change anything in files *Ticket.h* and *Ticket.cpp*.

3.3 Flight

`Flight` objects keep `flightCode`, `departureTime`, `arrivalTime`, `departureCity` and `arrivalCity` variables of type `std::string`, `economyCapacity` and `businessCapacity` variables of type `int`, and `completed` variable of type `bool` to hold the data related with the flights in the airline reservation system. A `Flight` object also keeps a `std::vector` of tickets to that flight (namely `tickets`). Most of the functions of `Flight` class are already implemented for you. In *Flight.cpp* file, you need to provide implementations for following functions declared under *Flight.h* header to complete the assignment. You should not change anything in file *Flight.h*.

3.3.1 `bool addTicket(const Ticket &ticket);`

This is the member function to add a new ticket to this flight. If there exists no empty seat in the asked ticket class (economy or business), do nothing.

3.3.2 `bool operator<(const Flight &rhs) const;`

This is the overloaded less than comparison operator. First, you should lexicographically compare the `flightCode` values of this flight and the given flight (`rhs`). If the compared value of this flight is less than the given flight's corresponding value, return `true`. Otherwise, return `false`.

3.4 AirlineReservationSystem

In `AirlineReservationSystem` class, all member functions should utilize `categories` member variable to operate as described in the following subsections. In *AirlineReservationSystem.cpp* file, you need to provide implementations for following functions declared under *AirlineReservationSystem.h* header to complete the assignment.

3.4.1 `void addPassenger(const std::string &firstname, const std::string &lastname);`

This function adds a new passenger to the airline reservation system. It takes the passenger information (`firstname` and `lastname`) as parameter and inserts a new `Passenger` object to the `passengers` binary search tree. If there already exists a passenger with the same fullname, do nothing.

3.4.2 `Passenger *searchPassenger(const std::string &firstname, const std::string &lastname);`

This function searchs for an existing passenger from the airline reservation system. It takes the `firstname` and `lastname` of the passenger to search as parameter and returns a pointer to that `Passenger` object from the `passengers` binary search tree. If there exists no such passenger, you should return `NULL`.

3.4.3 `void addFlight(const std::string &flightCode, const std::string &departureTime, const std::string &arrivalTime, const std::string &departureCity, const std::string &arrivalCity, int economyCapacity, int businessCapacity);`

This function adds a new flight to the airline reservation system. It takes the flight information (`flightCode`, `departureTime`, `arrivalTime`, `departureCity`, `arrivalCity`, `economyCapacity` and `businessCapacity`) as parameter and inserts a new `Flight` object to the `flights` binary search tree. If there already exists a flight with the same `flightCode`, do nothing.

3.4.4 `std::vector<Flight *> searchFlight(const std::string &departureCity, const std::string &arrivalCity);`

This function searchs for the existing flights from the airline reservation system. It takes the `departureCity` and `arrivalCity` of the flights to search as parameter and returns a `std::vector` of pointers to those `Flight` objects from the `flights` binary search tree. If there exists no such passenger, you should return an empty `std::vector`.

3.4.5 `void issueTicket(const std::string &firstname, const std::string &lastname, const std::string &flightCode, TicketType ticketType);`

This function creates a new ticket of type `ticketType` for the passenger with `firstname` and `lastname` to the flight with `flightCode`. If there exists no such passenger or flight, do nothing. Also, if there exists no empty seat in the flight for the asked ticket class (economy or business), do nothing.

3.4.6 `void saveFreeTicketRequest(const std::string &firstname, const std::string &lastname, const std::string &flightCode, TicketType ticketType);`

This function creates a new free ticket request of type `ticketType` for the passenger with `firstname` and `lastname` to the flight with `flightCode`. If there exists no such passenger or flight, do nothing.

3.4.7 `void executeTheFlight(const std::string &flightCode);`

This function marks the flight with `flightCode` as completed. Before that, you should evaluate the free ticket requests (first come, first served) to fill the empty seats of the flight. Remove the accepted requests from the queue, and do not change the order of the other requests. If there exists no such flight, do nothing.

4 Driver Programs

To enable you to test your `BST` and `AirlineReservationSystem` implementations, two driver programs, *main_bst.cpp* and *main_ars.cpp* are provided.

5 Regulations

1. **Programming Language:** You will use C++.
2. **S**tandard Template Library is **not** allowed unless you are explicitly asked to use it for a task.
3. **E**xternal libraries other than those already included are **not** allowed.
4. **T**hose who do the operations (insert, remove, search, ...) without utilizing the binary search tree will receive **0 grade**.
5. **T**hose who modify already implemented functions and those who insert other data variables or public functions and those who change the prototype of given functions will receive **0 grade**.
6. **Y**ou can add private member functions whenever it is explicitly allowed.
7. **Late Submission Policy:** Each student receives 5 late days for the entire semester. You may use late days on programming assignments, and each allows you to submit up to 24 hours late without penalty. For example, if an assignment is due on Thursday at 11:30pm, you could use 2 late days to submit on Saturday by 11:30pm with no penalty. Once a student has used up all their late days, each successive day that an assignment is late will result in a loss of 5% on that assignment.

No assignment may be submitted more than 3 days (72 hours) late without permission from the course instructor. In other words, this means there is a practical upper limit of 3 late days usable per assignment. If unusual circumstances truly beyond your control prevent you from submitting an assignment, you should discuss this with the course staff as soon as possible. If you contact us well in advance of the deadline, we may be able to show more flexibility in some cases.

8. **Cheating:** We have zero tolerance policy for cheating. In case of cheating, all parts involved (source(s) and receiver(s)) get zero. People involved in cheating will be punished according to the university regulations. Remember that students of this course are bounded to code of honor and its violation is subject to severe punishment.

9. **Newsgroup:** You must follow the Forum (`odtuclass.metu.edu.tr`) for discussions and possible updates on a daily basis.

6 Submission

- Submission will be done via CengClass (`cengclass.ceng.metu.edu.tr`).
- Don't write a main function in any of your source files.
- A test environment will be ready in CengClass.
  - You can submit your source files to CengClass and test your work with a subset of evaluation inputs and outputs.
  - Additional test cases will be used for evaluation of your final grade. So, your actual grades may be different than the ones you get in CengClass.
  - Only the last submission before the deadline will be graded.