i. Your name and student ID
Gideon Levi 109006114

ii. How to compile and execute your program and give an execution example.

- ❖ Change directory to DS_Final_Project folder

```
root@LAPTOP-T7IQ0UJ2:~# cd ..
root@LAPTOP-T7IQ0UJ2:/# cd home
root@LAPTOP-T7IQ0UJ2:/home# cd DS_final_project/
root@LAPTOP-T7IQ0UJ2:/home/DS_final_project#
```

- ❖ Compile with g++ -g -std=c++11 -o ./bin/main ./src/*.cpp. This command will compile all the files in “src” folder and generate an executable file “main” in the “bin” folder.

```
root@LAPTOP-T7IQ0UJ2:/home/DS_final_project# g++ -g -std=c++11 -o ./bin/main ./src/*.cpp
root@LAPTOP-T7IQ0UJ2:/home/DS_final_project#
```

- ❖ Type ./bin/main <case> <version> to run the compiled executable file. The above instruction needs to specify the parameters (case and version) to run.

  The options of parameters are listed below: ● case: case1, case2, case3

  ● version: basic, advance

  For example, to test the basic version on case3, execute “./bin/main case3 basic”;

  To test the advanced version on case1, execute “./bin/main case1 advance”

```
root@LAPTOP-T7IQ0UJ2:/home/DS_final_project# ./bin/main case3 basic
You have set case3 as your testcase:
running basic currently
-------------------------------------------
starting basic...
MAP is done.
SP_TABLE is done.
BIKE is done.
BIKE INFO is done.
READ USER is done.
PROCESS USER is done.
USER_RESULT is done.
TRANSFER_LOG is done.
STATION_STATUS is done.
ALL DONE.
-------------------------------------------
finished computation at Wed Jan  4 20:38:14 2023
elapsed time: 0.450504s
root@LAPTOP-T7IQ0UJ2:/home/DS_final_project#
```

```
root@LAPTOP-T7IQ0UJ2:/home/DS_final_project# ./bin/main case1 advance
You have set case1 as your testcase:
running advance currently
-------------------------------------------
starting advance...
MAP is done.
SP_TABLE is done.
BIKE is done.
BIKE INFO is done.
READ USER is done.
PROCESS USER is done.
USER_RESULT is done.
TRANSFER_LOG is done.
STATION_STATUS is done.
ALL DONE.
-------------------------------------------
finished computation at Wed Jan  4 20:38:46 2023
elapsed time: 0.00421235s
root@LAPTOP-T7IQ0UJ2:/home/DS_final_project#
```

- To run the verifier program, first execute “chmod +x bin/verifier”.
  After that execute “./bin/verifier $case”. The variable case could be case1, case2, case3 depending on which case we want to verify. For example, to verify the case3 results, execute “. /bin/verifier case3“.

```
root@LAPTOP-T7IQ0UJ2:/home/DS_final_project# ./bin/verifier case3
You have set case3 as your path:
------------------------------------------
start load result
bike_deprecation_rate : 0.500000 max_rental_count : 40
------------------------------------------
Total Revenue : 30201987
------------------------------------------
finished computation at Wed Jan  4 20:41:32 2023
elapsed time: 0.666767s
root@LAPTOP-T7IQ0UJ2:/home/DS_final_project#
```

iii. The details of your data structures. What data structures did you use, and how did you implement those data structures.

To store stations, I use BST with the station’s id as a key, and for each station, I use another BST to store bikes with the bike id as a key.

To store users, I also use BST. There are 2 types of BST to store users, one is a BST with the user’s accepted start time as a key, and the other one is a BST with the user id as a key.

To store bike initial prices, I use an 1d array and store it according to their types.

To store bike discount price and rent count limit, I just save them as global integer variables.

- UserNode class: store user data and methods.
  - I use this class to store user-related data such as user id, decision, all accepted bike types, accepted start and end time, revenue, start and end point, left and right links to another UserNode, etc.

```
Ubuntu-20.04 > home > DS_final_project > src > basic.cpp

//to store status of all requests
class UserNode {
    // friend class UserBST;
    // friend class UserInorderIterator;
    // private:
    public:
        string user_id_str;
        int user_id_int;
        int decision;
        ListNode<int>* all_acc_bike;
        int bike_id;
        int acc_start_time;
        int acc_end_time;
        int ride_start_time;
        int ride_end_time;
        int revenue;
        int start_point_int;
        int end_point_int;
        UserNode* left;
        UserNode* right;
        UserNode(string u_id_str, int deci, int b_id, int r_str_t, int r_end_t, int rev
            user_id_str = u_id_str;

            string temp;
            for(int i = 1; i < user_id_str.length(); i++) {
                temp.push_back(user_id_str[i]);
            }
            user_id_int = stoi(temp);
            decision = deci;
            bike_id = b_id;
            ride_start_time = r_str_t;
            ride_end_time = r_end_t;
            revenue = rev;
            left = NULL;
            right = NULL;
```

```
Ubuntu-20.04 > home > DS_final_project > src > basic.cpp
            }
            user_id_int = stoi(temp);
            decision = deci;
            bike_id = b_id;
            ride_start_time = r_str_t;
            ride_end_time = r_end_t;
            revenue = rev;
            left = NULL;
            right = NULL;
        }

        void print_data() {
            // std::cout << user_id_str << " " << decision << " " << bike_id << " "
            // << ride_start_time << " " << ride_end_time << " " << revenue << endl;
            std::cout << user_id_str << " " << 0 <<  " " << acc_start_time << " " << ac
            << " " << start_point_int << " "  << end_point_int << endl;
        }

        void set_all_acc_bike(ListNode<int>* all_acc_b) {
            all_acc_bike = all_acc_b;
        }

        void set_acc_st_time(int acc_st_t) {
            acc_start_time = acc_st_t;
        }

        void set_acc_end_time(int acc_end_t) {
            acc_end_time = acc_end_t;
        }

        void set_st_end_point(int st_p, int end_p) {
            start_point_int = st_p;
            end_point_int = end_p;
        }
};
```

- UserBST class: to implement User BST with UserNode as its nodes.
    - I use this class to store the root of the user binary search tree.

```cpp
//UserNode BST
class UserBST {
    private:
        UserNode* user_bst_root;

    public:
        UserBST(UserNode* root) {
            user_bst_root = root;
        }

        UserNode* get_root() {
            return user_bst_root;
        }
```

    - I also use this class to store methods to use for user BST, such as get_root, inorder_print, insert with the accepted start time as a key, and insert with user id as a key. For User BST with start time as a key, there might be more than 1 user with the same start_time value, so I modify the implementation of this BST a little bit to handle this. First, just insert the user normally, and if it encounters another user with the same start_time value, move to the right child until the right child's start_time value is not the same anymore, then set the inserted user as the current user's right child and set the inserted node's right child to the current node's old child. This way I can store 2 or more users with the same start_time value and if there is more than 1 user with the same start_time value, the user with a larger id will also have larger rank (since users in user.txt file are specified in ascending order of their id).`

Ubuntu-20.04 > home > DS_final_project > src > basic.cpp

```cpp
            rec_inorder_print(root->right);
        }
    }

    void insert_user_st_t(UserNode* user) {
        if (!user_bst_root)
        {
            // Insert the first node, if root is NULL.
            user_bst_root = user;
            return;
        }
        UserNode *prev = NULL;
        UserNode *temp = user_bst_root;
        while (temp)
        {
            if (temp->acc_start_time > user->acc_start_time)
            {
                prev = temp;
                temp = temp->left;
            }
            else if (temp->acc_start_time < user->acc_start_time)
            {
                prev = temp;
                temp = temp->right;
            }
            else {
                while(temp->right && temp->right->acc_start_time == user->acc_start_time) {
                    temp = temp->right;
                }
                user->right = temp->right;
                temp->right = user;
                return;
            }
        }
```

Ubuntu-20.04 > home > DS_final_project > src > basic.cpp

```cpp
            return;
        }
        UserNode *prev = NULL;
        UserNode *temp = user_bst_root;
        while (temp)
        {
            if (temp->acc_start_time > user->acc_start_time)
            {
                prev = temp;
                temp = temp->left;
            }
            else if (temp->acc_start_time < user->acc_start_time)
            {
                prev = temp;
                temp = temp->right;
            }
            else {
                while(temp->right && temp->right->acc_start_time == user->acc_start_time) {
                    temp = temp->right;
                }
                user->right = temp->right;
                temp->right = user;
                return;
            }
        }
        if (prev->acc_start_time > user->acc_start_time)
        {
            prev->left = user;
        }
        else
        {
            prev->right = user;
        }
    }
```

- BikeNode class: store bike data.
    - I use this class to store bike-related data such as bike type, bike id, rental price, rental count, bike start and end time, and left and right links to another BikeNode.

```
class BikeNode
{
    friend class StationNode;
    friend class BikeInorderIterator;
    // friend StationBST
private:

public:
    BikeNode *left;
    BikeNode *right;
    string bike_type;
    int bike_type_int;
    int bike_id;
    float rental_price; // dollars per minute
    int rental_count;
    int bike_start_time;
    int bike_end_time;

    BikeNode(string b_ty, int b_id, float rent_p, int rent_c)
    {
        bike_type = b_ty;
```

```
    BikeNode(string b_ty, int b_id, float rent_p, int rent_c)
    {
        bike_type = b_ty;

        string temp;
        for (int i = 1; i < bike_type.length(); i++)
        {
            temp.push_back(bike_type[i]);
        }
        bike_type_int = stoi(temp);

        bike_id = b_id;
        // station_id = st_id;
        rental_price = rent_p;
        rental_count = rent_c;
        bike_start_time = -1;
        bike_end_time = -1;
        left = NULL;
        right = NULL;
    }
};
```

- StationNode: store station data, implement Bike BST with BikeNode as its nodes, and store bike BST-related methods.
    - I use this class to store the station id, left and right links to another StationNode, and root of the bike binary search tree.
    - I also use this class to store methods to use for bike BST, such as insert with bike id as a key, search bike by id, search bike by type, and remove bike.

```
class StationNode {
    friend class StationBST;
    friend class StationInorderIterator;

    private:
        string station_id;  // ex: s11
        int station_id_num; // ex: 11
        StationNode *left;
        StationNode *right;

    public:
        BikeNode *bike_bst_root;
        StationNode(string st_id)
        {
            station_id = st_id;
            string temp;
            for (int i = 1; i < st_id.length(); i++)
            {
                temp.push_back(st_id[i]);
            }
            station_id_num = stoi(temp);
            left = NULL;
            right = NULL;
        }
        // insert bike with the station's bike_bst_root as root
        void insert_bike(BikeNode * const bike)
        {
            if (!bike_bst_root)
            {
                // Insert the first node, if root is NULL.
                bike_bst_root = bike;
```

- StationBST: store StationNode as a binary search tree.
  - I use this class to store the root of the station binary search tree.
  - I also use this class to store methods to use for station BST, such as insert station with station id as a key, search station by id, and inorder_print.

```
class StationBST
{
friend class StationInorderIterator;
private:
    StationNode *station_bst_root;

public:
    StationBST(StationNode *root)
    {
        station_bst_root = root;
    };

    // insert station with station_bst_root as the root
    void insert_station(StationNode *station)
    {
        if (!station_bst_root)
```

iv. The details of your algorithm. You could use flow chart(s) and/or pseudo code to help elaborate your algorithm.

MAP

- First, I read the map.txt file from the test cases, and I convert the station id and distances from string to integer and save it to MAP 2d array, I set the distance from stations in which the distance is not specified in the file to MAX_INT. After that, I process the MAP array with the Dijkstra function to find the shortest path from each node to all other nodes. Each time the shortest path from 1 node to all other nodes is found, I save it to another 2d array called sp_table.

BIKE

- Before reading the bike.txt file, I initialize a StationBST instance called StationTree.
- For the bikes, I read the bike.txt file from the test cases. Similarly, I convert the data in each line of the file from a string to an integer if necessary, and initialize a BikeNode with these data. After that, with the bike's current station position I initialize a StationNode and insert it into the StationTree, and I insert the BikeNode I just initialized to this StationNode.

BIKE INFO

- Before reading the bike_info.txt file, I initialize two variables to save the discount price for every rent and the rent count limit, I also initialize an array called bike_initial_price to store the bike's initial price according to their type.
- I read the bike_info.txt files and retrieve the discount price and rent count limit from the first two lines. After that, I convert each line of the rest of the file from a string to an integer and save it to bike_initial_price with bike type as the index.

READ USER

- Before reading the user.txt file, I initialize an UserBST with the user start time as the key called ST_UserTree.
- Then I convert each line of the file from a string to an integer if necessary, and I initialize a UserNode with these data. Finally, I insert the UserNode I just initialized to the ST_UserTree.
- Note that all the data except accepted bike types are stored in a string or int variable, for accepted bike types I store it as a LinkedList with each type of bike in a ListNode as I need to iterate on it later.

PROCESS USER

A. Basic

- First, I initialize an UserInorderIterator instance to iterate on ST_UserTree in an inorder sequence, this way I can process user requests in the ascending order of their start_time, and since users in user.txt file are specified in ascending order of their id, so if there are 2 or more users with the same start_time my program will process them in ascending order of their id.
- Before processing the ST_UserTree, I initialize a new UserBST instance called ID_UserTree to store the processed user with user_id as a key.
- To process each user in ST_UserTree, I retrieve the distance of 2 stations the current user wants to travel from sp_table array, and retrieve the start station and the end station of the user.
- I initialize a LinkedListIterator instance to iterate on the user's all_acc_bike linked list and initialize a BikeInorderIterator instance to iterate on the user's start station's bike BST in an inorder sequence. I also initialize a LinkedList instance called all_avail_bike to store all bike that is suitable and also available for the current user. By suitable it means the bike's bike type is in the accepted bike type of the user and by available it means the bike is already there when the user arrives at the station since in the basic part we assume that users don't wait for the bikes.
- With both linked list iterator and bike BST iterator, I filter all the bikes in the user's start station and insert all the suitable and available bikes to the all_avail_bike linked list.
- After every bike is filtered and if there are no suitable bikes found, I set the user_bike to NULL. On the other hand, if there is at least 1 suitable bike found, I pick the bike with the highest rental price and set it as a user_bike, and if there is more than 1 bike with the same highest price, I pick the bike with the smallest id to set as user_bike.
- If the user_bike is NULL, I initialize a new UserNode instance and set all the data except the user_id to 0 to be inserted into the ID_UserTree (reject request). Otherwise, I calculate the user_revenue and initialize a new UserNode instance with data I just calculated and user_bike's data to be inserted into the ID_UserTree. After that, I update data in user_bike like bike_end_time, rental_count, and rental_price. Finally, I remove user_bike from the user's start station and insert it into the user's end station.

B. Advance

- For the advanced part, I implement a similar algorithm as the basic part until bike filtering. In the basic part, we assume that users don't wait for the bikes, but this time users can wait for the bike as long as they are able to arrive at the destination station before their accepted end time.
- Another difference is I implement a free bike transfer service in the advanced part. After filtering the bikes and not 1 suitable and available bike is found, then the free bike transfer service is used. (If there is a suitable and available bike found already, then a similar algorithm as in the basic part is implemented.)
- To implement a free bike transfer service, I first find the closest station to the current user's start station, say station x, and filter the bikes in station x with a similar algorithm as in basic part to find a suitable and available bike for the current user. If user_bike is found (not NULL), then I remove the user_bike from station x, insert it into the user's start station, and insert into ID_UserTree a new UserNode instance with id "-1" to specify that the bike is transferred using bike transfer. On the other hand, if user_bike is NULL, I find the second closest station to the current user's start station and search a suitable and available bike there. If a suitable and available bike is still not found, move to the third closest station, and so on. If all the stations have been visited and bike is still not found, initialize a new UserNode instance and set all the data except the user_id to 0 to be inserted into the ID_UserTree (reject request).

OUTPUT

- I open the user_result.txt file, and using an UserInorderIterator I iterate on ID_UserTree in an inorder sequence, this way I can write the user's request result in the ascending order of the user's id.
- I open the transfer_log.txt file, and similarly, I use UserInorderIterator to iterate on ID_UserTree in an inorder sequence, this way I can write the user's transfer log in the ascending order of the user's id.
- I open the station_status.txt file and use the inorder_print method in StationBST class, and since the StationBST is using station id as a key, using inorder_print method will write the station's bike status in the ascending order of the station's id.

v. Utility class and functions:

- StackNode class //class to be the node in Stack class
- Stack class //class to implement a stack
- Min_distance function //function to aid Djikstra in finding shortest time
- Dijkstra function //function to find shortest time
- ListNode class //class to be the node in LinkedList class
- LinkedList class //class to implement a linked list
- LinkedListIterator class //class for a linked list iterator
- BikeInorderIterator class //class for a bike BST iterator
- UserInorderIterator class //class for a user BST iterator
- StationInorderIterator class //class for a station BST iterator

vi. [Optional] If you have any feedback, please write it here. Such as comments for improving the spec of this assignment, etc.