GROUP ASSIGNMENT

TECHNOLOGY PARK MALAYSIA

CT077-3-2 DSTR

DATA STRUCTURES

APD2F2209CS(CYB) / APU2F2209CS(CYB) / APU2F2209CS(DA) / APU2F2209CS / APU2F2209CS(IS) / APU2F2209SE / APD2F2209CS(DA) / APD2F2209SE / APD2F2209CS(IS) / APU2F2209CS(DF) / APD2F2209CS(DF) / APD2F2209CS

HAND OUT DATE: 17 – MARCH - 2023

HAND IN DATE: 29 – MAY - 2023

WEIGHTAGE: 50%

INSTRUCTIONS TO CANDIDATES:

1 **Submit your assignment at the Moodle System.**

2 **Students are advised to underpin their answers with the use of references (cited using the APA Style System of Referencing)**

3 **Late submission will be awarded zero (0) unless Extenuating Circumstances (EC) are upheld**

4 **Cases of plagiarism will be penalized**

5 **The assignment should be submitted in softcopy, where the softcopy of the written assignment and source code (where appropriate) should be on Moodle System.**

6 **You must obtain 50% overall to pass this module.**

Members	**TP Numbers**
Low Sim Chuan	TP065697
Nivethan Ramesh	TP062192
Andy Wijaya	TP062789
Han Rui Ming	TP066161

Contents

1. Acknowledgment

In no uncertain terms, Miss Chong Mien May was an invaluable resource for us throughout the 'Data Structures' course, and I want to express how much I appreciate everything that she has done to help us succeed. Our education and interest in data structures owe a great deal to her undying enthusiasm for the field and her extraordinary capacity to inspire and encourage inquiry.

We have become interested in the subject because Miss Chong Mien May is so committed to teaching and adept at breaking down complex concepts into manageable chunks. She has created a stimulating learning environment during the classes whether the lecture or the tutorials that encourages students to take risks, think critically, and work together. Miss Chong Mien's dedication to the data structure module has made us to be able to grasp the theoretical underpinnings of data structures as well as their real-world applications because to her manner of instruction, which combines theoretical knowledge with hands-on experience. Those explanations have formed a clear understanding of the module for our team to build this system.

The lessons we learned from Miss Chong Mien May were invaluable as we took on the difficult task of developing a university recommendation system for the Ministry of Higher Education. Her wisdom and experience helped us to make use of cutting-edge methods like hash table and doubly linked lists to create effective, user-centric solutions.

Our expertise of data structures was put to the test, and at the same time, we were encouraged to expand our horizons by adding novel touches to improve the end-user's experience. The importance that Miss Chong Mien May places on originality and overcoming problems has motivated us to do our best in this assignment.

We owe a great debt of gratitude to Miss Chong Mien May for her unending positivity and belief in us as students. The influence of her dedication to student achievement and her ability to instill a love of learning in us has lasted far beyond her time as our teacher. How grateful we are to her goes beyond what words can say.

Finally, we would like to thank Miss Chong Mien May once again for her invaluable assistance throughout the course of the 'Data Structures' unit. She was essential in our education and instilled in us a deep respect for the field. For all miss has taught us, we will be eternally grateful, and I want to earn her approval by putting our newfound knowledge to good use.

Miss Chong Mien May, we are eternally grateful to you for your undying support and the inspiration you have provided us with. You have evolved into something more than a fantastic educator as a result of the impact you have had on your pupils.

2. Executive Summary

To help parents of high school students make educated decisions about their children's post-secondary education, the Ministry of Higher Education (MoHE) has launched a groundbreaking endeavour to compile essential data on universities throughout the globe. MoHE has collaborated to create a state-of-the-art smart recommendation engine that uses data from the comprehensive 2023 QS University Rankings dataset, which profiles more than 1,400 institutions from across the world.

Using C++ programming and the implementation of suitable data structures, our skilled team is committed to building a robust system to store and handle a wide variety of information, including specifics about universities. Following a thorough analysis of MoHE's specifications, we designed a system that meets the needs of everyone from casual browsers to paying customers.

Beyond catering to the essentials, our team has also meticulously crafted supplementary features meant to improve the overall user experience and provide useful data.

The public may use our technology to see extensive university data, filter university details by name, conduct targeted institution-specific searches, and join up as education system customers. More functionality is available to registered users, including the option to filter university data by various ratings, search for schools based on specified criteria, add schools to a "favourites" list, and send comments to MoHE. Administrators may utilise MoHE's dashboard to log in and out, view and update user profiles, delete inactive accounts, organise user feedback, and respond to inquiries.

We have designed two distinct search and sorting algorithms in response to the job requirements. The effectiveness of these algorithms will be thoroughly examined in the final report. In addition, we have ensured that common programming practises, such as naming variables appropriately, including comments, and using the appropriate amount of indentation, have been adhered to throughout the system.

Our technique for recommending colleges and universities is cutting edge. We have created a system that aids both parents and students in making educated decisions about their academic futures by compressing a wealth of information and resources into a straightforward interface. This cutting-edge platform not only raises the bar for educational strategy software, but also exemplifies our unwavering commitment to furthering MoHE's mandate of expanding access to higher education.

In conclusion, our team has developed a world-class recommendation system for the Ministry of Higher Education, one that integrates crucial information about colleges with innovative features developed with individual users in mind. We have built a system using the full potential of C++ programming and sophisticated data structures to assist parents in making well-informed decisions concerning their children's secondary school education.

3. Introduction

In a world where opportunities in higher education are expanding at a dizzying rate, students now have a plethora of institutions from which to choose. Both parents and kids have a difficult challenge when trying to choose which schools would best serve their interests and help them achieve their goals. In light of this difficulty, the Ministry of Higher Education (MoHE) has launched an innovative project to create a university recommendation system that will arm parents of secondary school students with the information they need to make well-informed decisions about their children's higher education.

This recommendation system is built with the great collaborations from the MoHE itself and many other collaborators. They helped to develop the recommendation system by mining the 2023 QS University Ranking dataset. The dataset has the information of detail of more than 1400 institutions from around the world. For this assignment, our group worked hard to create a flexible, user-friendly system that can store and process a broad range of data for the university recommendation system.

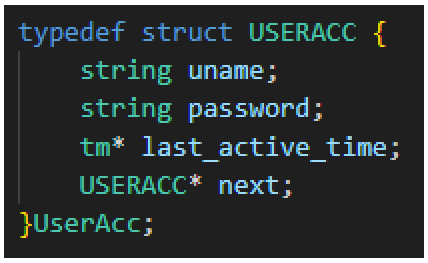
The fundamental objective of the university recommendation system is to provide customers with an easy-to-navigate, user-friendly interface from which they can quickly and easily get access to relevant, up-to-date information about institutions all over the globe. Each kind of user, from casual browsers to paying clients to MoHE execs, has been carefully considered in the creation of this innovative system. Features and functionality are tailored to meet the individual requirements of each user population.

Our team has creatively imagined additional features to improve user experience and deliver vital insights, on top of achieving the fundamental objectives stated by MoHE. These innovative capabilities span a wide range of uses, allowing customers to make educated purchases that suit their unique tastes.

Using C++ and complex data structures, the university's recommendation system strikes a balance between speed and accuracy. Not only does it establish a new standard for educational planning software, but it also illustrates our constant commitment to furthering MoHE's purpose of expanding access to higher education.

To conclude, this system can be a great solution for the navigation of the institution from the dataset chosen. From the system developed, it can now assist the parents to make decisions and look at different institutions for their kids. With its robust features, user-focused layout, and seamless integration of the newest institution rankings data, this cutting-edge platform fosters and supports the ambitions and dreams of countless students by setting a new bar for quality in higher education planning tools.

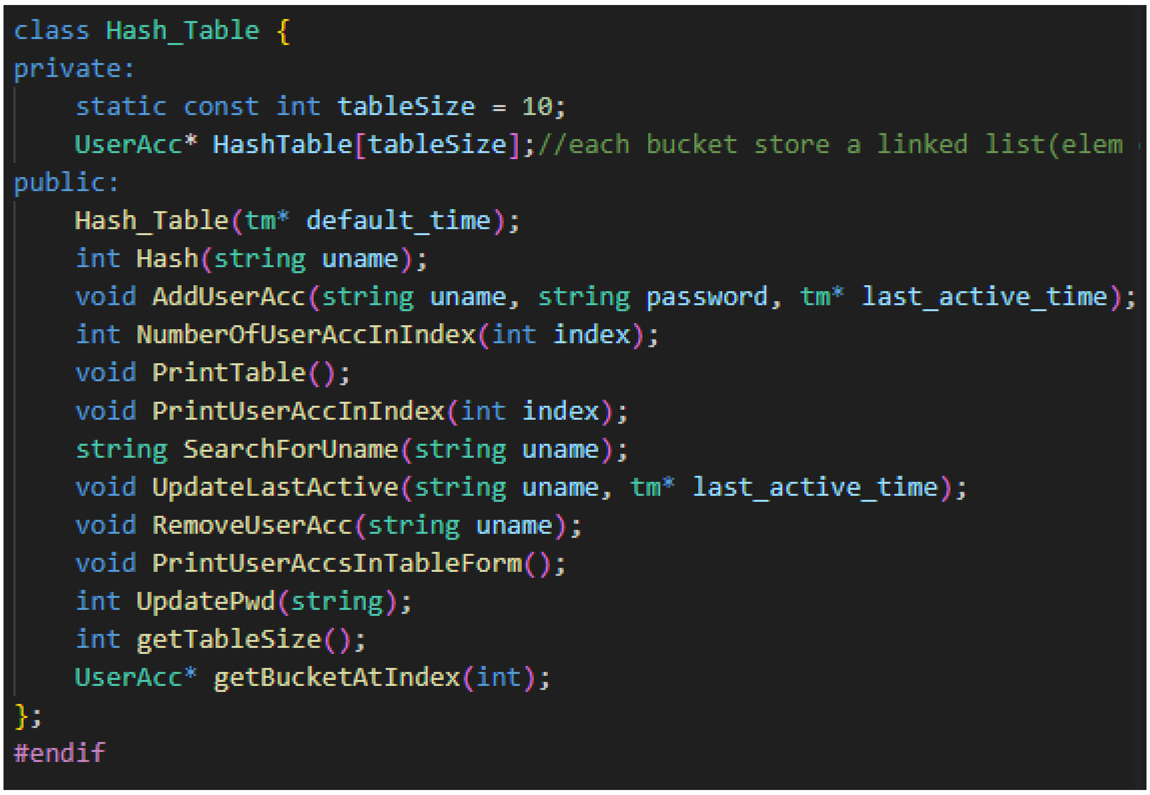
3.1 Hash Table



```
typedef struct USERACC {
    string uname;
    string password;
    tm* last_active_time;
    USERACC* next;
}UserAcc;
```

Figure 3.1.1: UserAcc structure definition

Figure 3.1.1 shows the structure definition of *UserAcc* (it is in **HashTable.h**), it is for representing the accounts of admins and customers of the system. The *UserAcc* contains four members: *uname* which stands for username, *password* which stands for user password, *last_active_time* which stands for the time when the user last active at, and *next* stands for pointer of next *UserAcc* structure. The *UserAcc* structure represents a node in a hash table.



```
class Hash_Table {
private:
    static const int tableSize = 10;
    UserAcc* HashTable[tableSize];//each bucket store a linked list(elem
public:
    Hash_Table(tm* default_time);
    int Hash(string uname);
    void AddUserAcc(string uname, string password, tm* last_active_time);
    int NumberOfUserAccInIndex(int index);
    void PrintTable();
    void PrintUserAccInIndex(int index);
    string SearchForUname(string uname);
    void UpdateLastActive(string uname, tm* last_active_time);
    void RemoveUserAcc(string uname);
    void PrintUserAccsInTableForm();
    int UpdatePwd(string);
    int getTableSize();
    UserAcc* getBucketAtIndex(int);
};
#endif
```

Figure 3.1.2: Hash_Table class definition

Figure 3.1.2 shows the class definition of hash table (in **HashTable.h**). *tableSize* is set to 10, and the array of pointers to *UserAcc*, *HashTable* is defined, and *HashTable* can store 10 pointers to *UserAcc* (called buckets). Each bucket can store a linked list, new *UserAcc* node will be added to hash table at index (the index ranges from 0 to 9 because *tableSize* is set to 10) determined by *Hash(string uname)* function.

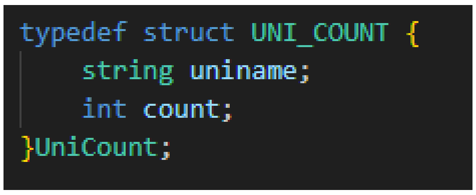
Hash(string uname) is a hash function which maps keys to values, keys here refer to *uname* of the *UserAcc* and values here refer to indexes of the *HashTable* array. If a new *UserAcc* node gets the index where there is already a *UserAcc* node (this is also called as hash collision), then the new *UserAcc* node will be appended to the old *UserAcc* node like a linked list. A good hash function, *Hash(string uname)*, should be efficiently computable so that index of new node can be computed quickly, and should uniformly distribute the nodes so that each linked list is almost of the same length (ranadeepika2409, n.d.).

The hash table is suitable for storing user credentials because of its constant complexity *O(1)* (in average case) when it comes to search, insert and delete data (Fulber-Garcia, 2023). The worst case for these operations is typically *O(n)* – linear time, this case happens when all data in hash table have keys that map to the same index (Fulber-Garcia, 2023).

Hash table is good for searching. For instance, when the user is trying to login, the key, *uname*, will be hashed first to get the index. With that index, the linked list will be traversed until the end to see if there is a matching node. And the linked list is not of the same length as the total number of the users in the system due to the nature of hash function. Consider that a linked list is used to store all the *UserAcc*, and its length is the same as the total number of users, where worst case is traverse through all the users. But an ideal hash table will just make similar length of linked lists to cut down the total number of nodes in each linked list.

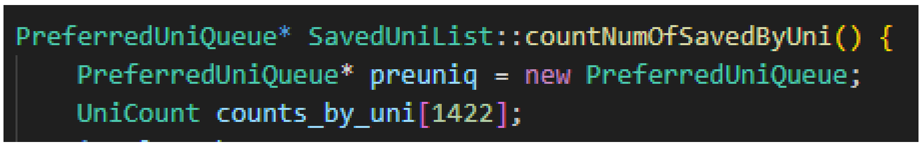
The hash table is also good for insertion as well. For instance, if a user is trying to register him/herself as a customer of the system, the entered username will be hashed to get the index. With that index, the new node will be added to the end of the linked list, and the length of linked list is not the same as the total number of customers in the system.

3.2 Array of Structures



```
typedef struct UNI_COUNT {
    string uniname;
    int count;
}UniCount;
```

Figure 3.2.1: UniCount structure

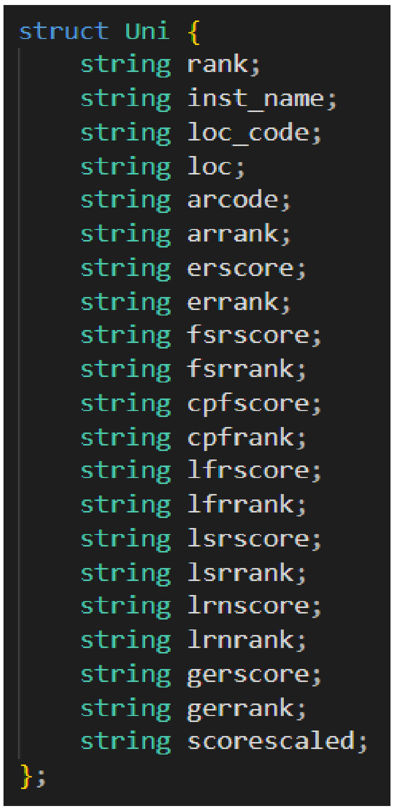


```
PreferredUniQueue* SavedUniList::countNumOfSavedByUni() {
    PreferredUniQueue* preuniq = new PreferredUniQueue;
    UniCount counts_by_uni[1422];
```

Figure 3.2.2: count_by_uni array

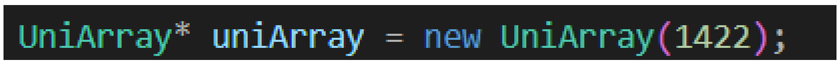
UniCount structure (defined in **SavedUniList.h**) is an element in an array, *count_by_uni* (defined in **SavedUniList.cpp**). *UniCount* structure represents the number of saves of the university, the number is stored in *count*. *counts_by_uni* is used to store the number of saves of each university present in the *SavedUniList* class (will be introduced in the section **3.4 Singly Linked List**).

Array is suitable for this case because the total number of universities is known, it is 1422. If a linked list is used here, each node requires extra memory to store the pointer to the next node, this is a waste of memory. In this use case, only traversal from head to tail and insertion are required, array and linked list have the same complexities for these 2 operations. Although linked list is faster in deleting a node, there is no need for such operation in this use case, plus linked list requires more memory to store pointer to next node, hence an array is preferred over a linked list.



```
struct Uni {
    string rank;
    string inst_name;
    string loc_code;
    string loc;
    string arcode;
    string arrank;
    string erscore;
    string errank;
    string fsrscore;
    string fsrrank;
    string cpfscore;
    string cpfrank;
    string lfrscore;
    string lfrrank;
    string lsrscore;
    string lsrrank;
    string lrnscore;
    string lrnrank;
    string gerscore;
    string gerrank;
    string scorescaled;
};
```

Figure 3.2.3: 'Uni' structure



```
UniArray* uniArray = new UniArray(1422);
```

Figure 3.2.4: 'UniArray' array

The 'Uni' structure and 'UniArray' are used extensively throughout the presented code and are essential to its operation. The 'Uni' structure neatly organizes all the components of a university. This organization is helpful since it simplifies the process of working with different data sets together and makes them more adaptable to our specific needs.

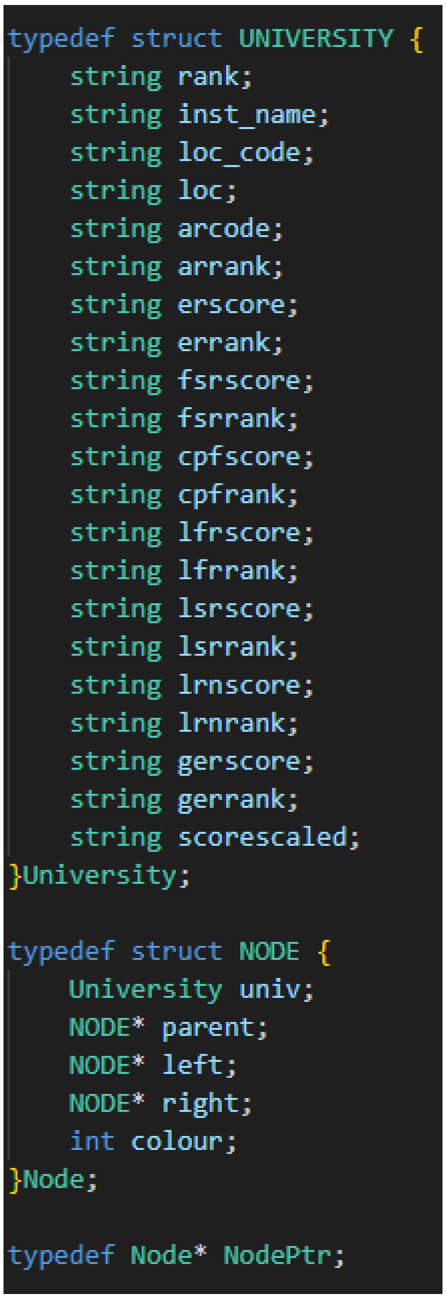
'UniArray' is just an array of 'Uni' objects that can grow and shrink on the fly. The memory use of this array is more efficient than that of a linked list since its size (1422) is known in advance. Keeping track of pointers to other entries in a linked list takes up extra space that is not required.

We would need to be able to traverse and insert data into this system. The temporal complexity of these operations is the same for both arrays and linked lists. The primary benefit of linked lists, however, is not relevant here: faster deletion times. This justifies switching to an array.

Plus, with our own 'UniArray', we will have more leeway in how we manipulate data. We will have an easier time doing advanced tasks that are system specific.

As a whole, the combination of the ‘Uni’ structure and ‘UniArray’ results in a functional and efficient system. The configuration provides enough speed and memory efficiency for the tasks at hand. Also, it gives us the leeway to deal with any peculiarly complicated processes that may arise in our setup.

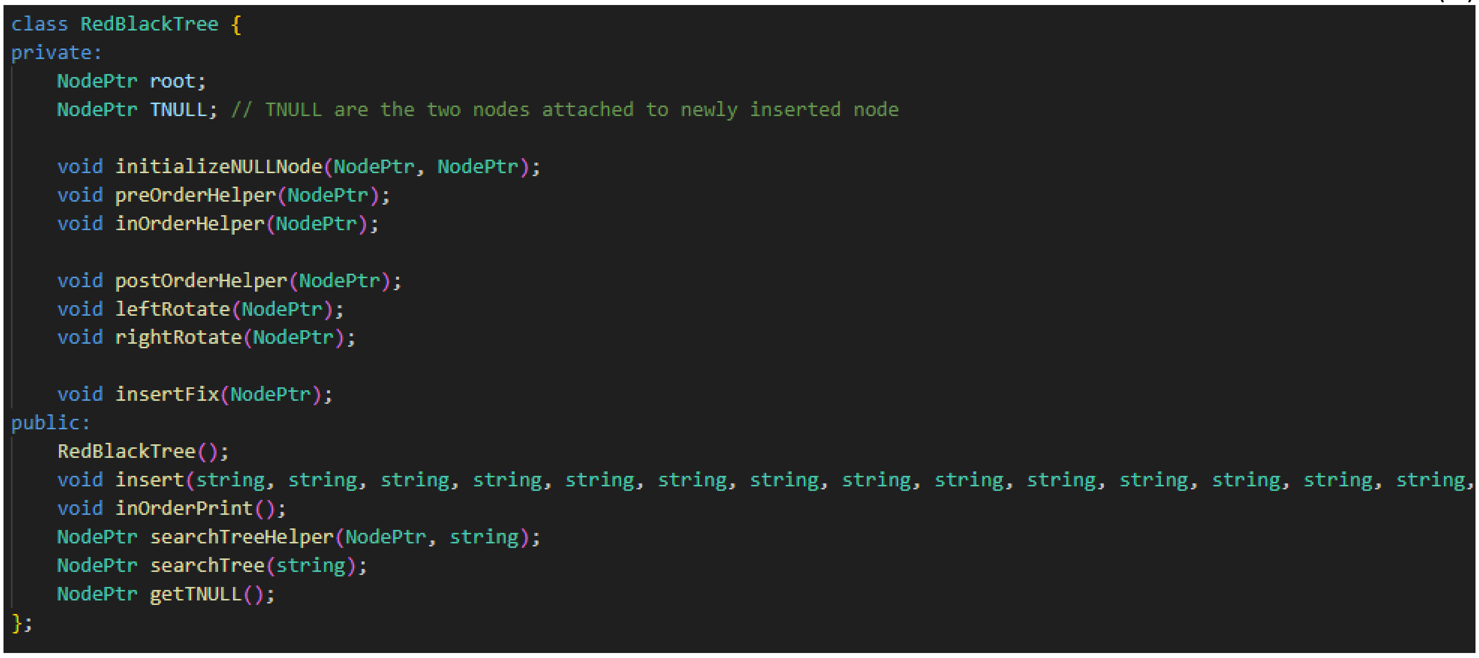
3.3 Red Black Tree



```
typedef struct UNIVERSITY {
    string rank;
    string inst_name;
    string loc_code;
    string loc;
    string arcode;
    string arrank;
    string erscore;
    string errank;
    string fsrscore;
    string fsrrank;
    string cpfscore;
    string cpfrank;
    string lfrscore;
    string lfrrank;
    string lsrscore;
    string lsrrank;
    string lrnscore;
    string lrnrank;
    string gerscore;
    string gerrank;
    string scorescaled;
}University;

typedef struct NODE {
    University univ;
    NODE* parent;
    NODE* left;
    NODE* right;
    int colour;
}Node;

typedef Node* NodePtr;
```

Figure 3.3.1: University structure and Node structure

The *University* structure represents a university record from the CSV file, containing several fields: rank, university name, location code, location, AR score, AR rank, and so on. The *Node* structure represents a node in the red black tree, containing several fields: *University* structure, pointer to parent node, pointer to left child node, pointer to right child node, and colour of the node (either red or black).



```
class RedBlackTree {
private:
    NodePtr root;
    NodePtr TNULL; // TNULL are the two nodes attached to newly inserted node

    void initializeNULLNode(NodePtr, NodePtr);
    void preOrderHelper(NodePtr);
    void inOrderHelper(NodePtr);

    void postOrderHelper(NodePtr);
    void leftRotate(NodePtr);
    void rightRotate(NodePtr);

    void insertFix(NodePtr);
public:
    RedBlackTree();
    void insert(string, string, string, string, string, string, string, string, string, string, string, string, string, string,
    void inOrderPrint();
    NodePtr searchTreeHelper(NodePtr, string);
    NodePtr searchTree(string);
    NodePtr getTNULL();
};
```

Figure 3.3.2: RedBlackTree class

Binary search tree is good at sorting and searching data, but it can perform badly when the tree is not balanced. At worst case, it can degenerate into a linear structure with a time complexity of O(n) (GeeksforGeeks, n.d.). To remedy that, red black tree introduces a set of rules to ensure it is balanced. The balancing mechanisms will keep time complexity of common operations is always O(log n), such as insertion, deletion, and searching.

Red black tree is just an advanced binary search tree, it has to satisfy all properties of a binary search tree, then with following additional properties (GeeksforGeeks, n.d.) to ensure it is balanced:

1. Root property: The root is black. For implementation, 0 indicates black while 1 indicates red.
2. External property: Every leaf is black, and the leaf is a NULL child of a node. For implementation, the leaf is represented by *TNULL* as shown in *Figure 3.3.2*.
3. Internal property: The children of a red node are black.
4. Depth property: All the leaves have the same black depth.
5. Path property: Every simple path from root to descendant leaf node contains the same number of black nodes.

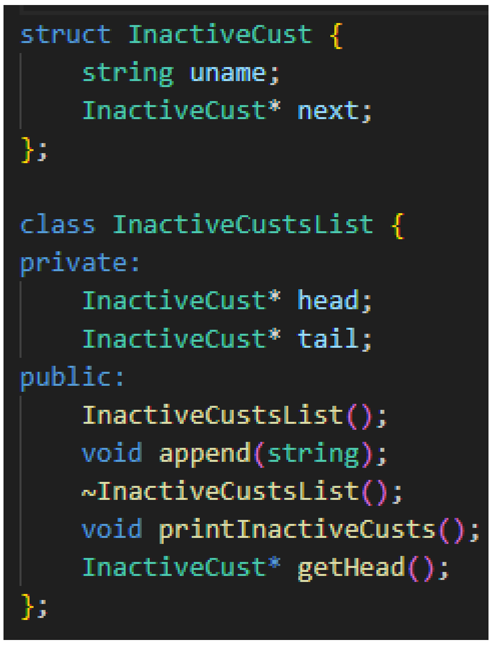
To do the balancing of red black when a new node is being inserted, Recolouring and Rotation are introduced (GeeksforGeeks, n.d.).

As in *Figure 3.3.2*, constructor method is to create a black node and set its children nodes to be NULL, then assign to *root* node of the red black tree. *TNULL* is also initialized in the constructor, which is black, and it has two NULL children nodes.

In *RedBlackTree* class, *insert()* method will create a new node which is red, has two TNULL as children nodes, has NULL parent, and all fields of *University* structure will be initialized.

In *RedBlackTree* class, *insertFix(), leftRotate()* and *rightRotate()* methods are used to do the recolouration and rotation when a new node is being inserted into the red black tree.

3.4 Singly Linked List



```
struct InactiveCust {
    string uname;
    InactiveCust* next;
};

class InactiveCustsList {
private:
    InactiveCust* head;
    InactiveCust* tail;
public:
    InactiveCustsList();
    void append(string);
    ~InactiveCustsList();
    void printInactiveCusts();
    InactiveCust* getHead();
};
```

Figure 3.4.1: InactiveCust structure and InactiveCustsList class

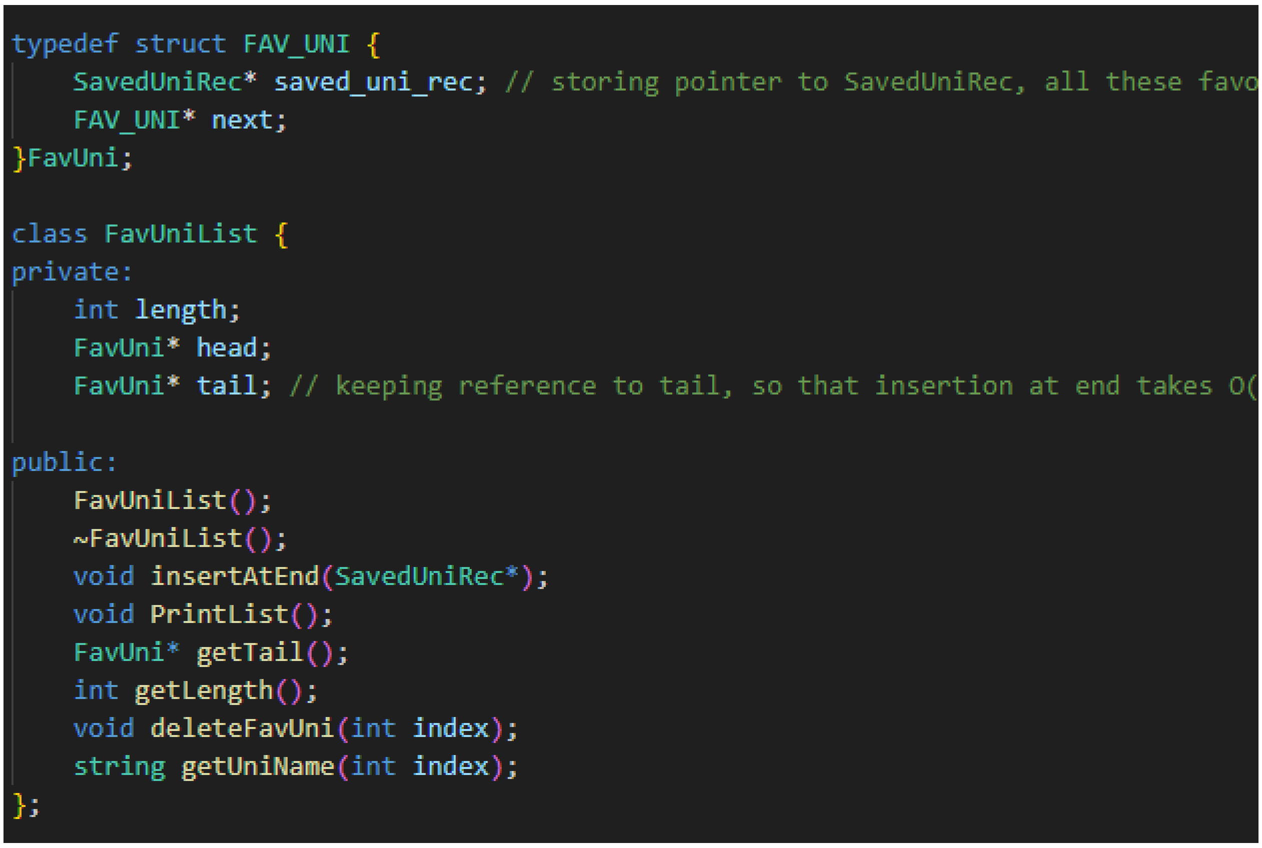
InactiveCust is a structure representing an inactive customer in the system (in **HashTable.h**), *uname* stands for his/her username and *next* stands for next inactive customer. *InactiveCustsList* is a class defined based on the concept of a singly linked list, *head* stores the pointer to the first node of the linked list, while *tail* stores the pointer to the last node of the linked list. *tail* here facilitates the *append(string)* to add a new node at the end of the singly linked list, without traversing from the first node to last node. *InactiveCustsList* is defined for storing all the inactive customers after traversing through the hash table of customer accounts, and the inactive customers are defined as the customers who have been inactive for at least 1 year.

In *InactiveCustsList* class, the destructor *~InactiveCustsList()* is crucial because the singly linked list will be stored in heap once the admins are redirected to the page of *DeleteCustAcc(Hash_Table*)* (in **AdminPages.cpp**). The destructor should be called if the admins quit the *DeleteCustAcc(Hash_Table*)* page, so that the allocated space at heap will be freed, ultimately memory leakage can be avoided. Memory leakage occurs in C++ when programmers allocates memory by using **new** keyword and forget to deallocate the memory by using **delete()** function or **delete[]** operator (ankit15697, n.d.).

In *InactiveCustsList* class, the *getHead()* method is to tell admins whether there are inactive customers in the system. If the *getHead()* returns NULL, it means there are no inactive customers after traversing through the hash table, then appropriate message can be displayed to the admins.

In the system outputs, the *InactiveCustsList* list will be traversed to show all the inactive customers to admins and ask for further confirmation on deletion of inactive customers' accounts. If the admins agree, then the usernames in *InactiveCustsList* will be passed to *Hash_Table::RemoveUserAcc(string uname)* and delete all of them.

A singly linked list is good in this case due to its length can vary during program execution. Without visiting all nodes in hash table, the exact length of the *InactiveCustsList* is still unknown, hence singly linked list is suitable due to its ease of insertion of new node. This is contrast to arrays, which its size must be predefined and cannot be changed if the arrays are already full.



```
typedef struct FAV_UNI {
    SavedUniRec* saved_uni_rec; // storing pointer to SavedUniRec, all these favo
    FAV_UNI* next;
}FavUni;

class FavUniList {
private:
    int length;
    FavUni* head;
    FavUni* tail; // keeping reference to tail, so that insertion at end takes O(

public:
    FavUniList();
    ~FavUniList();
    void insertAtEnd(SavedUniRec*);
    void PrintList();
    FavUni* getTail();
    int getLength();
    void deleteFavUni(int index);
    string getUniName(int index);
};
```

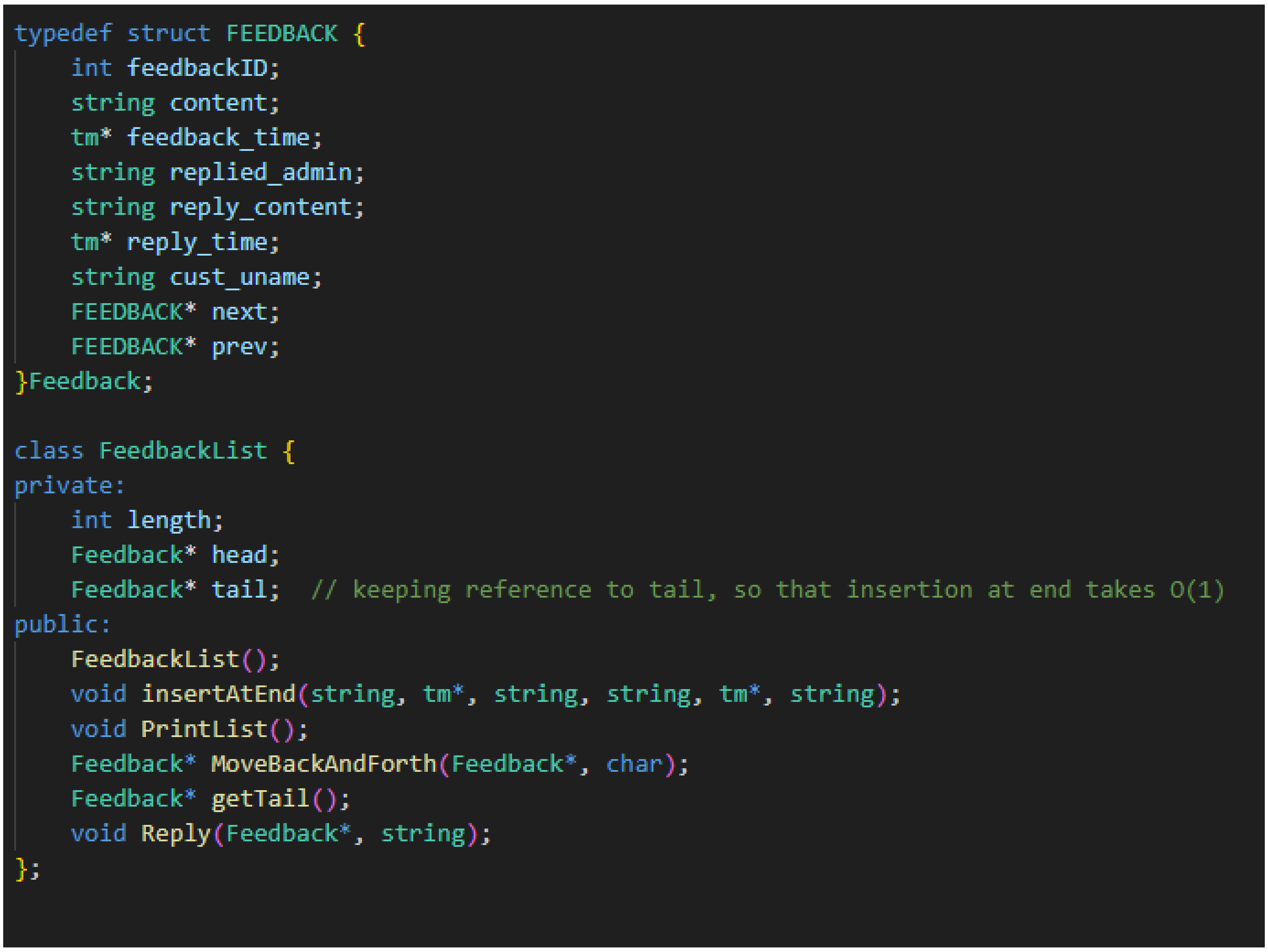
Figure 3.4.3: FavUni structure and FavUniList class

FavUni structure is a node in a singly linked list, *FavUniList* (in **FavUniList.h**). The *FavUni* node stores a pointer to a *SavedUniRec* node containing fields: saved university's name, customer who has saved it, pointer to next node, and pointer to previous node. The *FavUniList* is for storing the pointers of saved university records belonging to a specific customer, it can be utilized for disaplying purpose so that the customer can check a list of saved university and delete a saved university from the favourite list. In *FavUniList* class, the *head* and *tail* are also used here to store the pointers to first node and last node respectively. The *tail* facilitates the *insertAtEnd(SavedUniRec*)* to add a new node without traversing the whole linked list until the end every time.

In *FavUniList* class, the destructor *~FavUniList()* is crucial because the singly linked list will be stored in heap once the customers are redirected to the page of *CheckAndDeleteFavUni(string, SavedUniList*)*(in **CustomerPages.cpp**). The destructor should be called if the customers quit the *CheckAndDeleteFavUni(string, SavedUniList*)* page, so that the allocated space at heap will be freed, ultimately memory leakage can be avoided.

To justify why singly linked list being used here, it is due to its variable length again, the use case is similar to *InactiveCustsList* above. Without visiting the *SavedUniList* completely, the number of saved universities by a specific customer is still unknown. Hence, a singly linked list is used here to store pointers to those records, for both displaying purpose and deletion purpose.

3.5 Doubly Linked List



```
typedef struct FEEDBACK {
    int feedbackID;
    string content;
    tm* feedback_time;
    string replied_admin;
    string reply_content;
    tm* reply_time;
    string cust_uname;
    FEEDBACK* next;
    FEEDBACK* prev;
}Feedback;

class FeedbackList {
private:
    int length;
    Feedback* head;
    Feedback* tail;  // keeping reference to tail, so that insertion at end takes O(1)
public:
    FeedbackList();
    void insertAtEnd(string, tm*, string, string, tm*, string);
    void PrintList();
    Feedback* MoveBackAndForth(Feedback*, char);
    Feedback* getTail();
    void Reply(Feedback*, string);
};
```

Figure 3.5.1: Feedback structure and FeedbackList class

Feedback structure represents a node in doubly linked list, *FeedbackList*. *Feedback* structure represents each feedback record in the system, containing *feedbackID* for unique identification of each feedback, *content* for the content of feedback, *feedback_time* for the time when the customer sends the feedback, *replied_admin* for the admin who replied to the feedback, *reply_content* for the content of the reply, *reply_time* for the time when the admin replies to the feedback, *cust_uname* for the customer who sends the feedback, *next* for next feedback node, and *prev* for previous feedback node.

In *FeedbackList* class, as usual there are *head* and *tail* again, storing the pointer to first node and last node respectively. The presence of *tail* is to facilitate the *insertAtEnd(string, tm*, string, string, tm*, string)*, so that there is no need to traverse until the end every time for adding a new feedback node; *tail* is also for admins to read the latest feedback first. The *length* is to store the current number of feedbacks in the doubly linked list, so that the *feedbackID* of new feedback node can be auto generated. The *length* will be increased by 1 when the *insertAtEnd(string, tm*, string, string, tm*, string)* is called.

In *FeedbackList* class, there is a *getTail()* function for accessing the last feedback node so that admins can read and reply to the latest feedback node. The *getTail()* is called and initializes the current feedback node, the current feedback node will be displayed to the admins. The admins can just move back and forth by entering corresponding digits and *MoveBackAndForth(Feedback*, char)* is called to update the current feedback node being displayed to the admins.

In a doubly linked list, each node has the access to previous node and next node, which facilitates navigating in both directions (aayushi2402, n.d.). A doubly linked list is suitable for this case because admins are allowed to read the feedback by moving back and forth (*MoveBackAndForth(Feedback*, char)* function) and reply to any specific feedback (*Reply(Feedback*, string)* function). The presence of *next* and *prev* pointer variables enable admins to access the previous feedback node or next feedback node and reply to the feedback node.

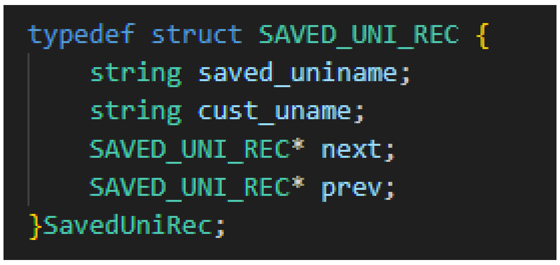

```
typedef struct SAVED_UNI_REC {
    string saved_uniname;
    string cust_uname;
    SAVED_UNI_REC* next;
    SAVED_UNI_REC* prev;
}SavedUniRec;
```

Figure 3.5.2: SavedUniRec structure

Figure 3.5.2 shows the *SavedUniRec* structure, representing a node in doubly linked list, *SavedUniList* class. *SavedUniRec* structure represents a record of saved university (*saved_uniname*) by a customer (*cust_uname*) in the system.

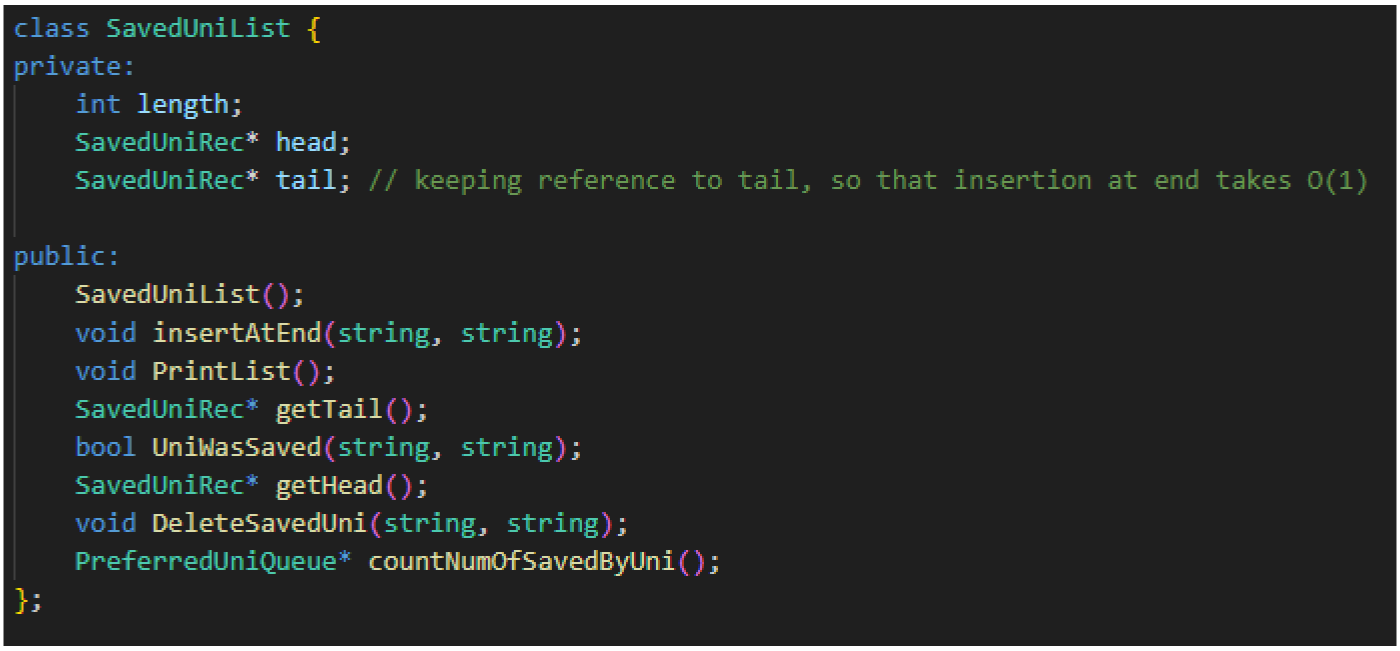
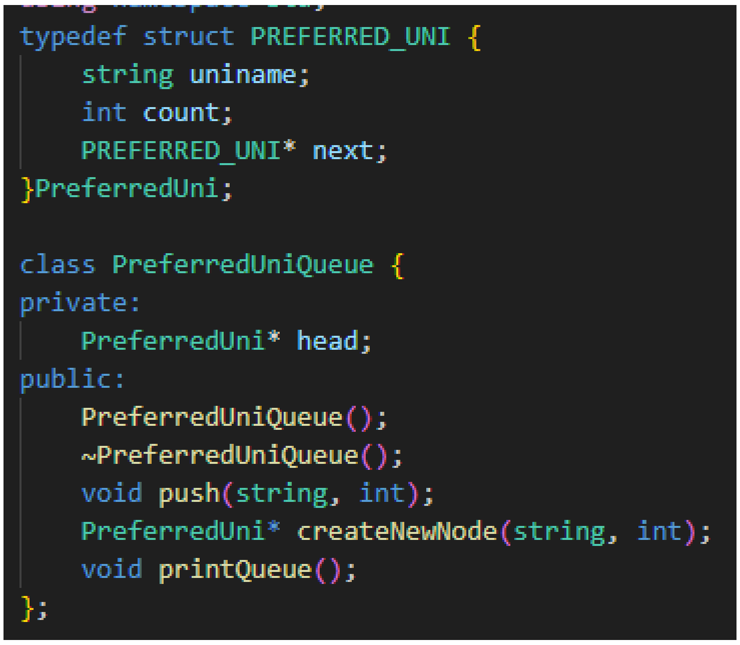

```
class SavedUniList {
private:
    int length;
    SavedUniRec* head;
    SavedUniRec* tail; // keeping reference to tail, so that insertion at end takes O(1)

public:
    SavedUniList();
    void insertAtEnd(string, string);
    void PrintList();
    SavedUniRec* getTail();
    bool UniWasSaved(string, string);
    SavedUniRec* getHead();
    void DeleteSavedUni(string, string);
    PreferredUniQueue* countNumOfSavedByUni();
};
```

Figure 3.5.3: SavedUniList class

In the *SavedUniList* class, there are 3 private attributes: *length* representing the length of the doubly linked list, *head* storing the pointer to first *SavedUniRec* node, and tail storing the pointer to last *SavedUniRec* node. The presence of *tail* here is again for faster insertion of new *SavedUniRec* node at the end of doubly linked list. (prev var keeps reference to previous node, facilitating the insertion and deletion)

A doubly linked list is used here because doubly linked list allows faster insertion and deletion of a node (Anand, 2021). This is because customers in the system can save universities or delete saved universities from favourite list frequently, traversing from front to end of doubly linked list after locating a desired node can be avoided by keeping a reference to previous node. Doubly linked list can greatly enhance the execution time of insertion and deletion of favourite universities.

3.6 Priority Queue



```
typedef struct PREFERRED_UNI {
    string uniname;
    int count;
    PREFERRED_UNI* next;
}PreferredUni;

class PreferredUniQueue {
private:
    PreferredUni* head;
public:
    PreferredUniQueue();
    ~PreferredUniQueue();
    void push(string, int);
    PreferredUni* createNewNode(string, int);
    void printQueue();
};
```

Figure 3.6.1: PreferredUni structure and PreferredUniQueue class

PreferredUni is a node in the priority queue, *PreferredUniQueue*. *PreferredUni* node represents number of saves of the university, and it is stored in *count*. *PreferredUniQueue* is a priority queue which will insert the new node at appropriate position considering its priority, in this case the priority is the number of saves. The *PreferredUniQueue* will have nodes arranged from most saved to least saved for admins to get a list of most preferred universities by customers.

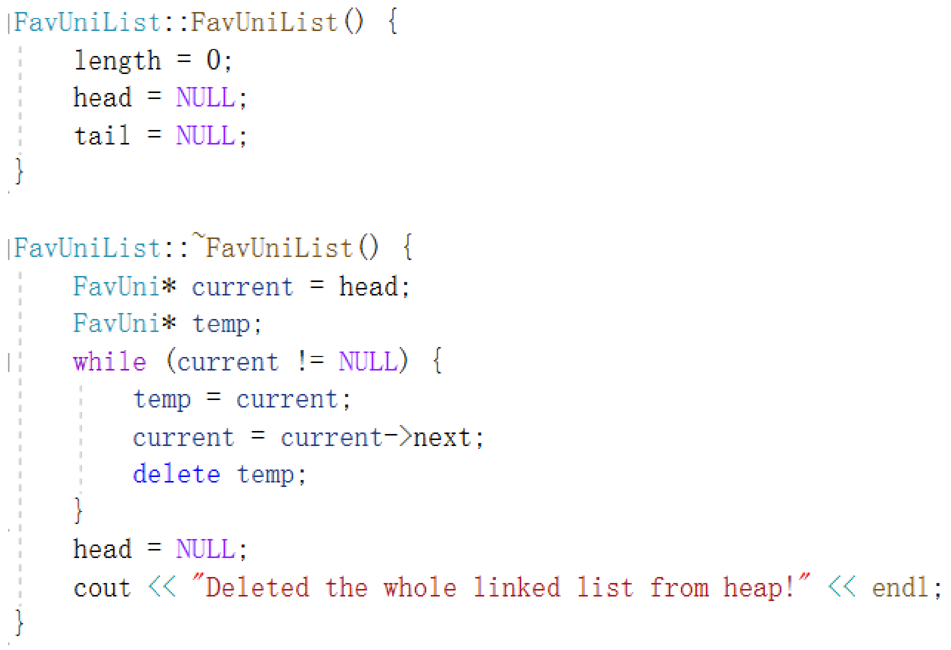
In *PreferredUniQueue* class, the destructor ~ *PreferredUniQueue()* is crucial because the priority queue will be stored in heap once the admins are redirected to the page of *SummarizeTopUniPage(SavedUniList*)*(in **AdminPages.cpp**). The destructor should be called if the admins quit the *SummarizeTopUniPage(SavedUniList*)* page, so that the allocated space at heap will be freed, ultimately memory leakage can be avoided.

In *PreferredUniQueue* class, the *push(string, int)* method is to insert the new node at appropriate position based on its number of saves.

To justify why priority queue is used here, priority queue moves the new node with highest priority to the front (Vithlani, 2022). The most preferred universities can be accessed first by admins.

4. Explanations of Algorithms

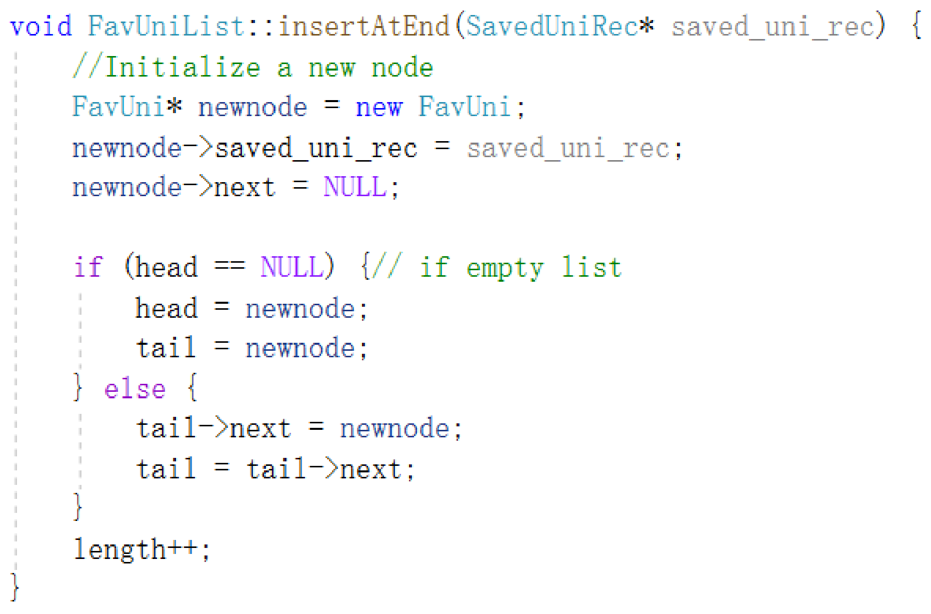
4.1 FavUniList class



```
FavUniList::FavUniList() {
    length = 0;
    head = NULL;
    tail = NULL;
}

FavUniList::~FavUniList() {
    FavUni* current = head;
    FavUni* temp;
    while (current != NULL) {
        temp = current;
        current = current->next;
        delete temp;
    }
    head = NULL;
    cout << "Deleted the whole linked list from heap!" << endl;
}
```

Figure 4.1.1: Constructor and Destructor of FavUniList()

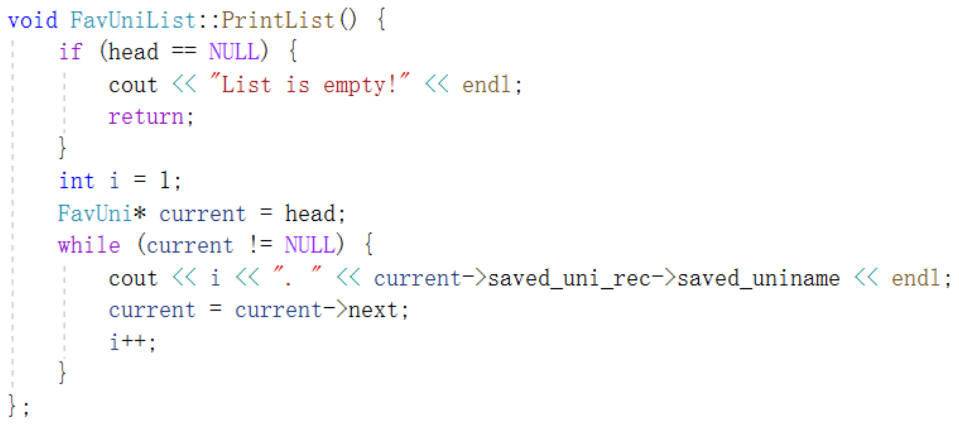
FavUniList class is a singly linked list for storing the university's name as favorite chosen by a customer. The figure above shows the constructor and destructor of the class. *FavUniList()* is the constructor of the class, which initializes the length of the linked list to 0. Then, the head and tail are set to NULL, meaning that there are no nodes inside yet. On the other hand, *~FavUniList()* is the destructor of the class, which is responsible for cleaning up the memory allocated for the linked list. A while loop is used to start from *head*, current node will be stored at *temp* and moves to the next node, then deletes the *temp*. The loop will stop until the *current* becomes NULL which means that it reached the end of the list. After the loop, *head* will set to NULL and a message will print out saying whole linked list is deleted.



```
void FavUniList::insertAtEnd(SavedUniRec* saved_uni_rec) {
    //Initialize a new node
    FavUni* newnode = new FavUni;
    newnode->saved_uni_rec = saved_uni_rec;
    newnode->next = NULL;

    if (head == NULL) {// if empty list
        head = newnode;
        tail = newnode;
    } else {
        tail->next = newnode;
        tail = tail->next;
    }
    length++;
}
```

Figure 4.1.2: insertAtEnd() for FavUniList

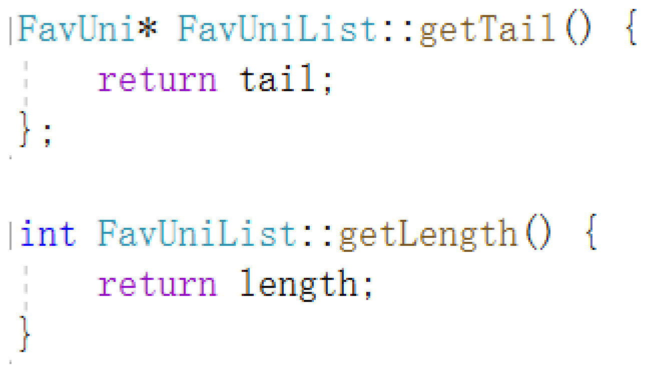
This is a method of inserting a new node at the end of the linked list. It creates a new node called *FavUni* and assigns the *SavedUniRec* pointer to member variables of *saved_uni_rec* and set *next* to NULL. An if statement is made if head is NULL, then both *head* and *tail* will be set to *newnode.* If otherwise, it adds new node after the current *tail* and update it and point it to the new node. When the statement is done, the *length* is increased by 1.



```
void FavUniList::PrintList() {
    if (head == NULL) {
        cout << "List is empty!" << endl;
        return;
    }
    int i = 1;
    FavUni* current = head;
    while (current != NULL) {
        cout << i << ". " << current->saved_uni_rec->saved_uniname << endl;
        current = current->next;
        i++;
    }
};
```

Figure 4.1.3: PrintList() for FavUniList

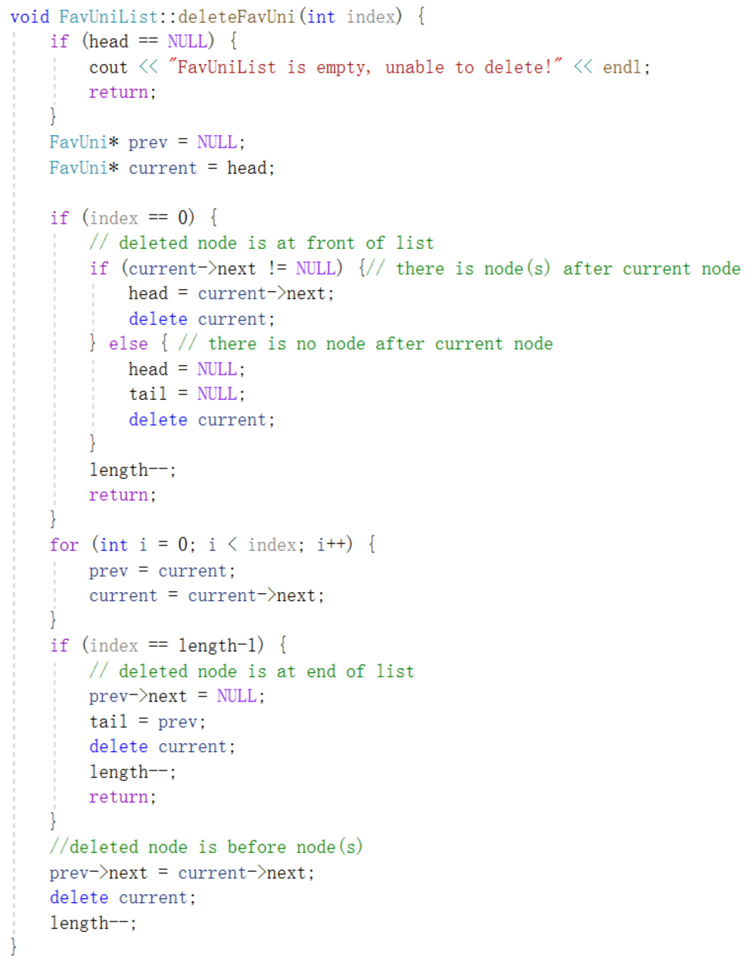
This method is used to print out the contents inside the linked list. It checks if head is NULL using if statement. If yes, message of list is empty will be printed out. Otherwise, it loops using while loop and print index and the names of universities. Then the *saved_uniname* will get university names from *saved_uni_rec*.



```
FavUni* FavUniList::getTail() {
    return tail;
};

int FavUniList::getLength() {
    return length;
}
```

Figure 4.1.4: getTail() and getLength() for FavUniList

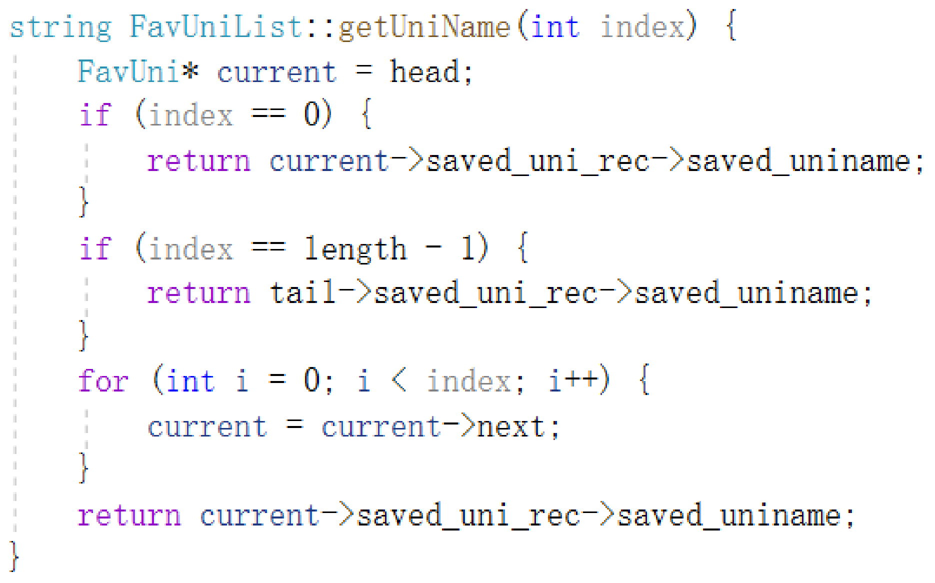
These two functions works alike. The *getTail()* works to returns a pointer to the *tail*. And *getLength()* returns the number of elements in the linked list.



```
void FavUniList::deleteFavUni(int index) {
    if (head == NULL) {
        cout << "FavUniList is empty, unable to delete!" << endl;
        return;
    }
    FavUni* prev = NULL;
    FavUni* current = head;

    if (index == 0) {
        // deleted node is at front of list
        if (current->next != NULL) {// there is node(s) after current node
            head = current->next;
            delete current;
        } else { // there is no node after current node
            head = NULL;
            tail = NULL;
            delete current;
        }
        length--;
        return;
    }
    for (int i = 0; i < index; i++) {
        prev = current;
        current = current->next;
    }
    if (index == length-1) {
        // deleted node is at end of list
        prev->next = NULL;
        tail = prev;
        delete current;
        length--;
        return;
    }
    //deleted node is before node(s)
    prev->next = current->next;
    delete current;
    length--;
}
```

Figure 4.1.5: deleteFavUni() for FavUniList

This method is used to delete the node at the specified index in the linked list. Firstly, it checks *head* if it is NULL, when it does, it prints out message that explains the situation. Then, *prev* and *current* of *FavUni* pointer are set to NULL and *head* respectively. Another if statement is used to check if the index is 0. If yes, it deletes the node which is at front of the list by verifying if there any node after current node. Otherwise, *head* and *tail* will be set to NULL and the current node will be deleted. A for loop is used to update the *prev* with *current* and *current* from *next*. Another for loop is used when *index* is the last element in the list, it will updates *prev->next* to NULL, *tail* is set as *prev*, *current* is deleted and *length* is minus 1 and returns the result. Lastly, *prev->next* is equals to *current->next* and current node will be deleted and *length* will minus 1.



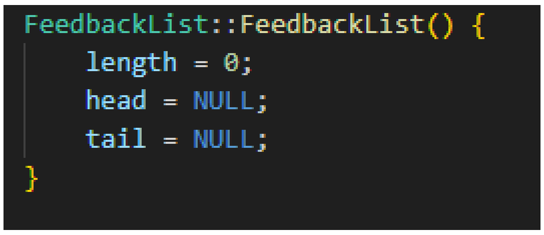
```
string FavUniList::getUniName(int index) {
    FavUni* current = head;
    if (index == 0) {
        return current->saved_uni_rec->saved_uniname;
    }
    if (index == length - 1) {
        return tail->saved_uni_rec->saved_uniname;
    }
    for (int i = 0; i < index; i++) {
        current = current->next;
    }
    return current->saved_uni_rec->saved_uniname;
}
```

Figure 4.1.6: getUniName() for FavUniList

This method is used to return the name of the university at the specified index in the linked list. This method begins with assigning *head* to *current* pointer. First if statement is used to return the name of the university from *current* to *saved_uni_rec* when the *index* is 0. Second if statement is used to return the name of the university from *tail* to *saved_uni_rec* when the *index* is equals to l*ength – 1*. And a for loop is implemented to loop the list and find the node at the specified index for other index. Lastly, the name of the university is returned to *current* when the node is found.

4.2 FeedbackList class

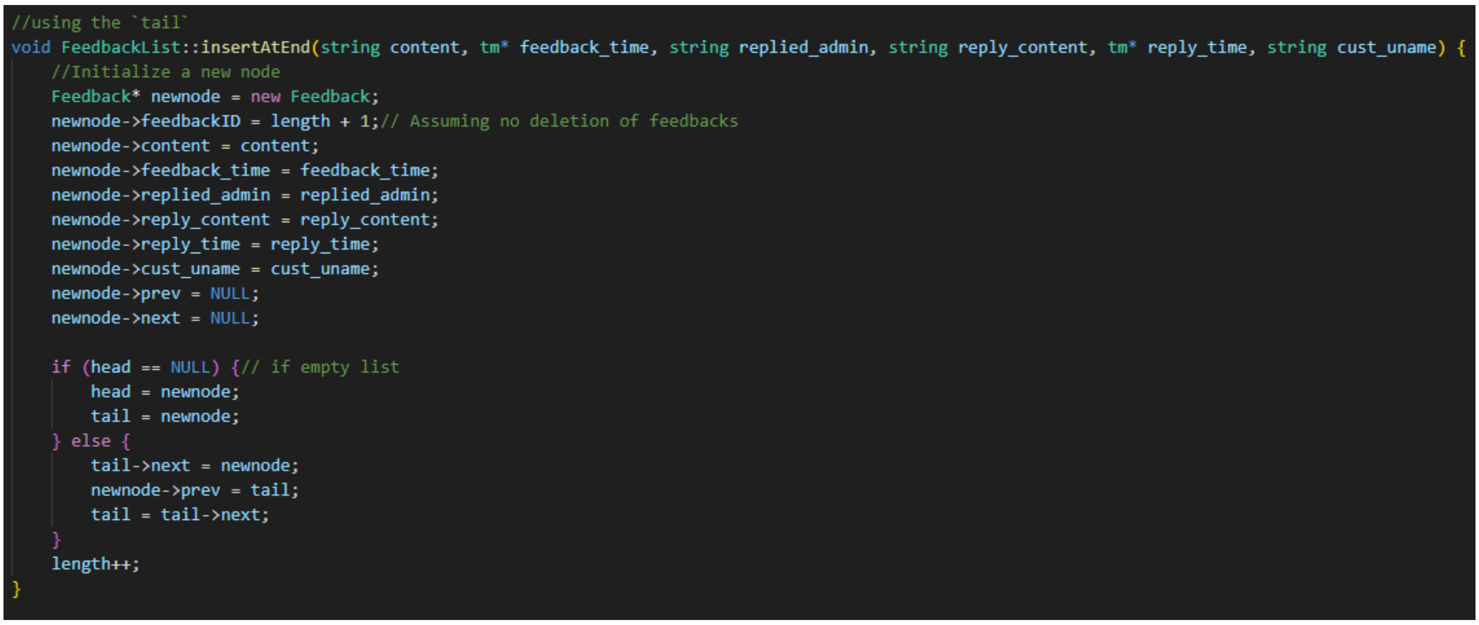
FeedbackList class (located in **FeedbackList.cpp**) is a doubly linked list for keeping all the feedbacks by customers, each feedback can be replied by admins as well.



```
FeedbackList::FeedbackList() {
    length = 0;
    head = NULL;
    tail = NULL;
}
```

Figure 4.2.1: FeedbackList() constructor

Constructor is to initialize the length of the doubly linked list to 0. *head* and *tail* are also pointing to NULL since there is no node in the doubly linked list.

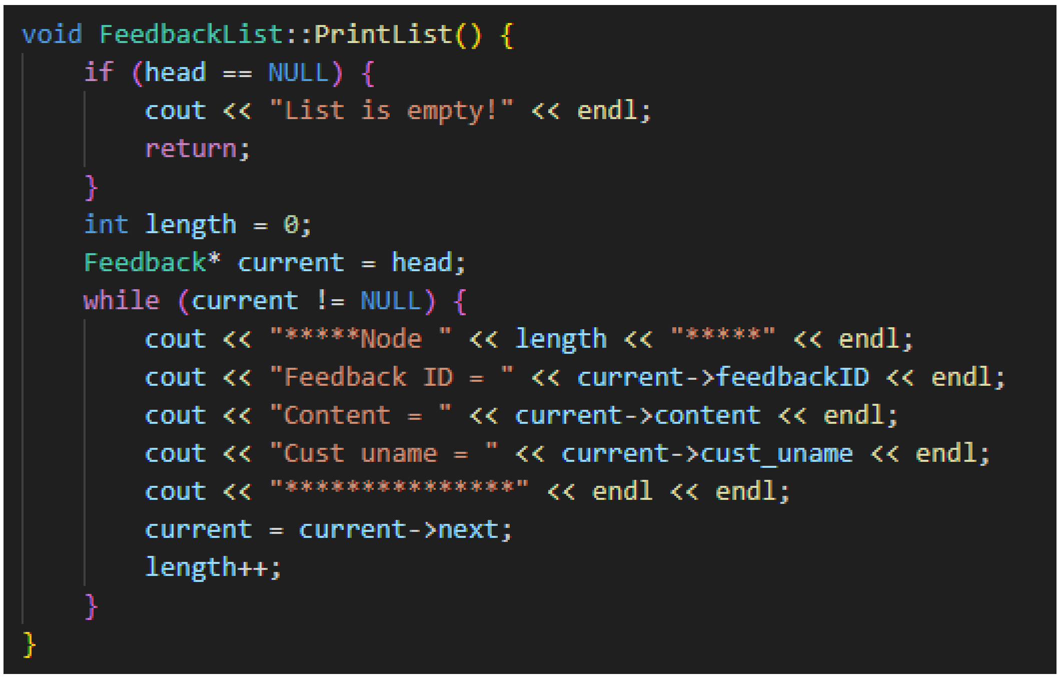


```
//using the `tail`
void FeedbackList::insertAtEnd(string content, tm* feedback_time, string replied_admin, string reply_content, tm* reply_time, string cust_uname) {
    //Initialize a new node
    Feedback* newnode = new Feedback;
    newnode->feedbackID = length + 1;// Assuming no deletion of feedbacks
    newnode->content = content;
    newnode->feedback_time = feedback_time;
    newnode->replied_admin = replied_admin;
    newnode->reply_content = reply_content;
    newnode->reply_time = reply_time;
    newnode->cust_uname = cust_uname;
    newnode->prev = NULL;
    newnode->next = NULL;

    if (head == NULL) {// if empty list
        head = newnode;
        tail = newnode;
    } else {
        tail->next = newnode;
        newnode->prev = tail;
        tail = tail->next;
    }
    length++;
}
```

Figure 4.2.2: insertAtEnd() method

insertAtEnd() method is for adding a new *Feedback* node at the end of the doubly linked list. The new node will be stored at heap using **new** operator, *length* will be used to give *Feedback* node a unique *feedbackID*. Some other initializations of *Feedback* node are also done.

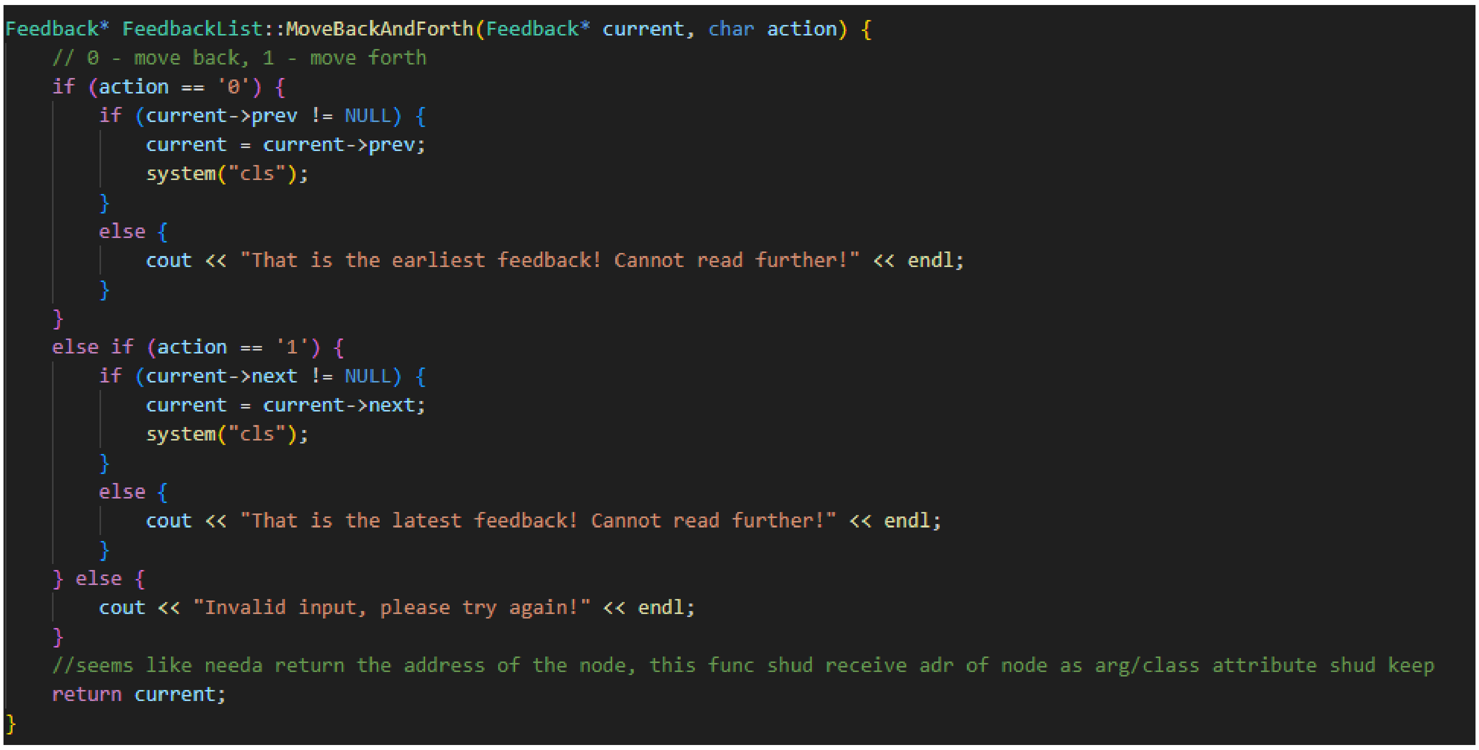
If there is no node before in the doubly linked list, both *head* and *tail* will point to the new node. Otherwise, the *tail* will be utilized to allow faster insertion of new node at the end. There is no need for traversal from front to end of doubly linked list for insertion. Length of the linked list will be updated at the end of this method.



```
void FeedbackList::PrintList() {
    if (head == NULL) {
        cout << "List is empty!" << endl;
        return;
    }
    int length = 0;
    Feedback* current = head;
    while (current != NULL) {
        cout << "*****Node " << length << "*****" << endl;
        cout << "Feedback ID = " << current->feedbackID << endl;
        cout << "Content = " << current->content << endl;
        cout << "Cust uname = " << current->cust_uname << endl;
        cout << "***************" << endl << endl;
        current = current->next;
        length++;
    }
}
```

Figure 4.2.3: PrintList() method

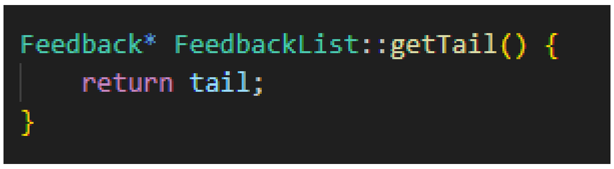
PrintList() method is for printing all the *Feedback* nodes from front to end. This method is useful for debugging purposes.



```
Feedback* FeedbackList::MoveBackAndForth(Feedback* current, char action) {
    // 0 - move back, 1 - move forth
    if (action == '0') {
        if (current->prev != NULL) {
            current = current->prev;
            system("cls");
        }
        else {
            cout << "That is the earliest feedback! Cannot read further!" << endl;
        }
    }
    else if (action == '1') {
        if (current->next != NULL) {
            current = current->next;
            system("cls");
        }
        else {
            cout << "That is the latest feedback! Cannot read further!" << endl;
        }
    } else {
        cout << "Invalid input, please try again!" << endl;
    }
    //seems like needa return the address of the node, this func shud receive adr of node as arg/class attribute shud keep
    return current;
}
```

Figure 4.2.4: MoveBackAndForth() method

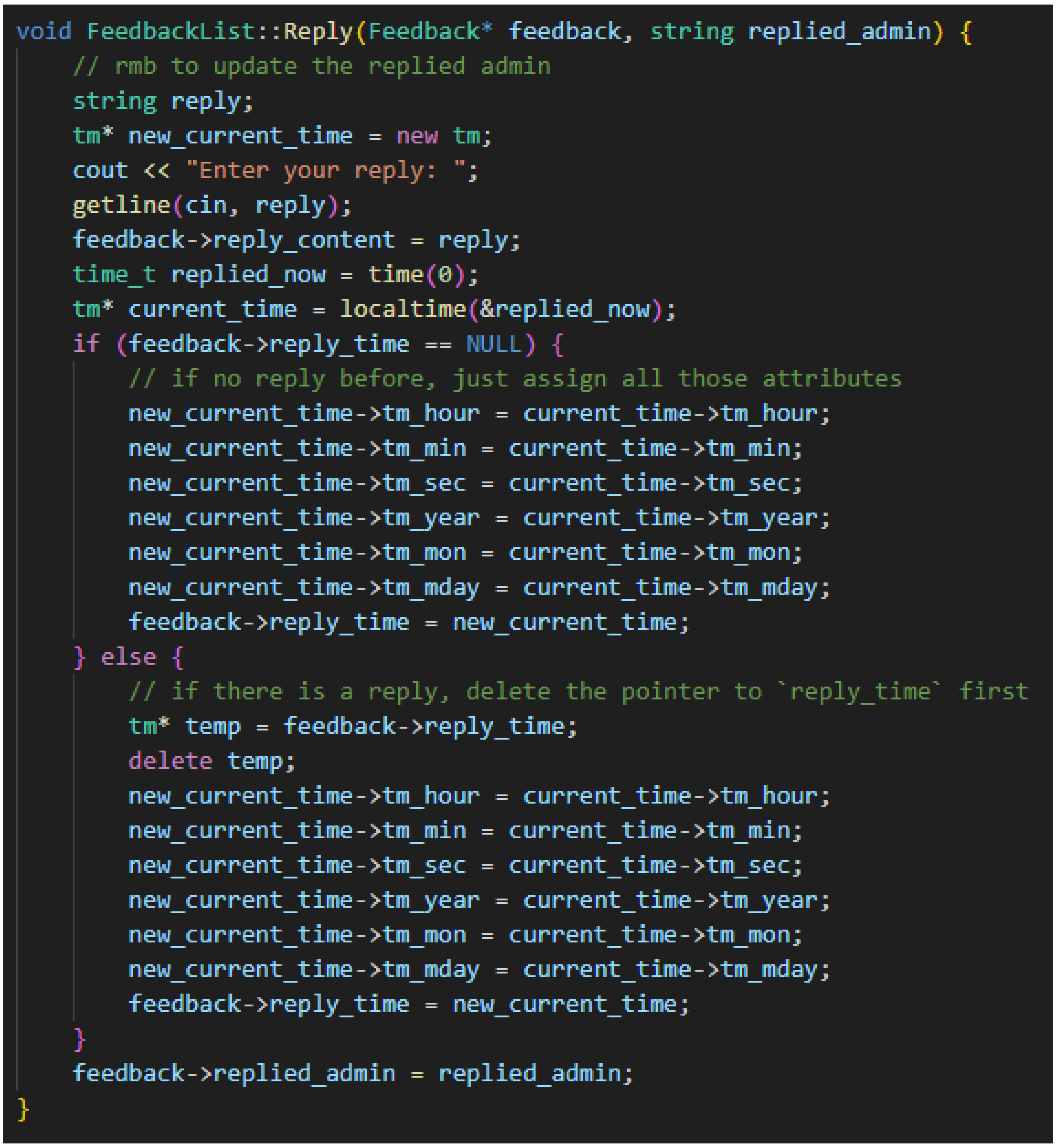
MoveBackAndForth() method returns pointer to a *Feedback* node. Based on the *action* given, the method will return pointer to either previous or next *Feedback* node. Once there is no more previous or next node, warning messages will be displayed to admins as well.



```
Feedback* FeedbackList::getTail() {
    return tail;
}
```

Figure 4.2.5: getTail() method

getTail() method is for returning the pointer to last *Feedback* node, so that admins can read the latest feedback.



```
void FeedbackList::Reply(Feedback* feedback, string replied_admin) {
    // rmb to update the replied admin
    string reply;
    tm* new_current_time = new tm;
    cout << "Enter your reply: ";
    getline(cin, reply);
    feedback->reply_content = reply;
    time_t replied_now = time(0);
    tm* current_time = localtime(&replied_now);
    if (feedback->reply_time == NULL) {
        // if no reply before, just assign all those attributes
        new_current_time->tm_hour = current_time->tm_hour;
        new_current_time->tm_min = current_time->tm_min;
        new_current_time->tm_sec = current_time->tm_sec;
        new_current_time->tm_year = current_time->tm_year;
        new_current_time->tm_mon = current_time->tm_mon;
        new_current_time->tm_mday = current_time->tm_mday;
        feedback->reply_time = new_current_time;
    } else {
        // if there is a reply, delete the pointer to `reply_time` first
        tm* temp = feedback->reply_time;
        delete temp;
        new_current_time->tm_hour = current_time->tm_hour;
        new_current_time->tm_min = current_time->tm_min;
        new_current_time->tm_sec = current_time->tm_sec;
        new_current_time->tm_year = current_time->tm_year;
        new_current_time->tm_mon = current_time->tm_mon;
        new_current_time->tm_mday = current_time->tm_mday;
        feedback->reply_time = new_current_time;
    }
    feedback->replied_admin = replied_admin;
}
```

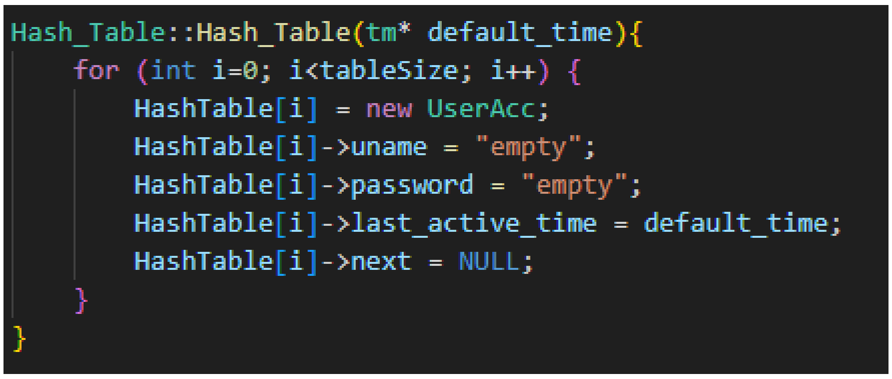
Figure 4.2.6: Reply() method

Reply() method is for admins to reply to a specific feedback. The reply from an admin is prompted, and the current time of the system is gotten as well. If there is no reply before, then just update the feedback's reply time to the current time; Otherwise, the previous reply time needs to be deleted from heap to avoid memory leakage using

delete operator, then update the feedback's reply time to the current time. At the end, whoever replies to this feedback is also updated.

4.3 Hash_Table class

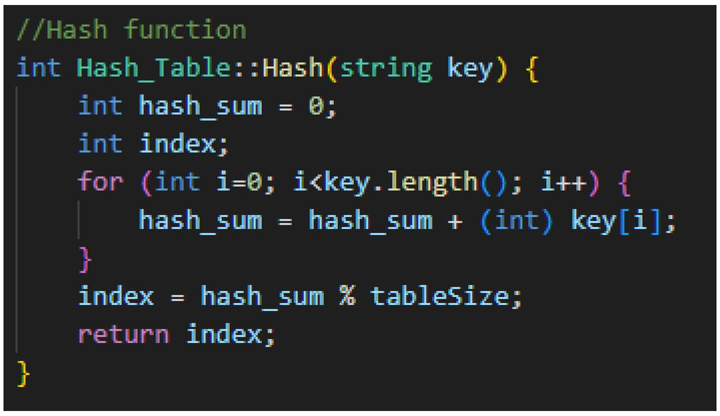
The hash table is used to store the information of admin accounts and customer accounts.



```
Hash_Table::Hash_Table(tm* default_time){
    for (int i=0; i<tableSize; i++) {
        HashTable[i] = new UserAcc;
        HashTable[i]->uname = "empty";
        HashTable[i]->password = "empty";
        HashTable[i]->last_active_time = default_time;
        HashTable[i]->next = NULL;
    }
}
```

Figure 4.3.1: Hash_Table() constructor

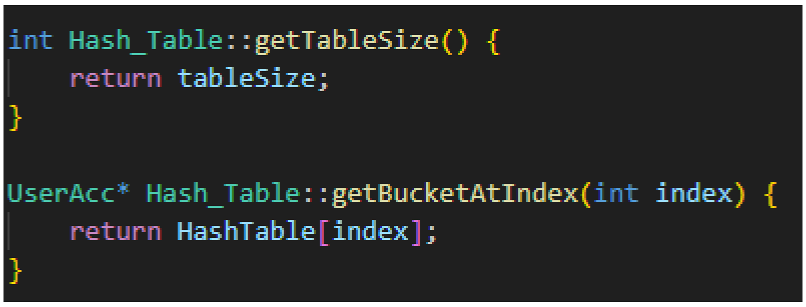
Hash_Table() constructor takes in *default_time* as parameter to initialize the last active time of the user, either an admin or a customer. Inside the constructor, each bucket's username and password will be initialized by the string "empty" and pointing to NULL.



```
//Hash function
int Hash_Table::Hash(string key) {
    int hash_sum = 0;
    int index;
    for (int i=0; i<key.length(); i++) {
        hash_sum = hash_sum + (int) key[i];
    }
    index = hash_sum % tableSize;
    return index;
}
```

Figure 4.3.2: Hash() method

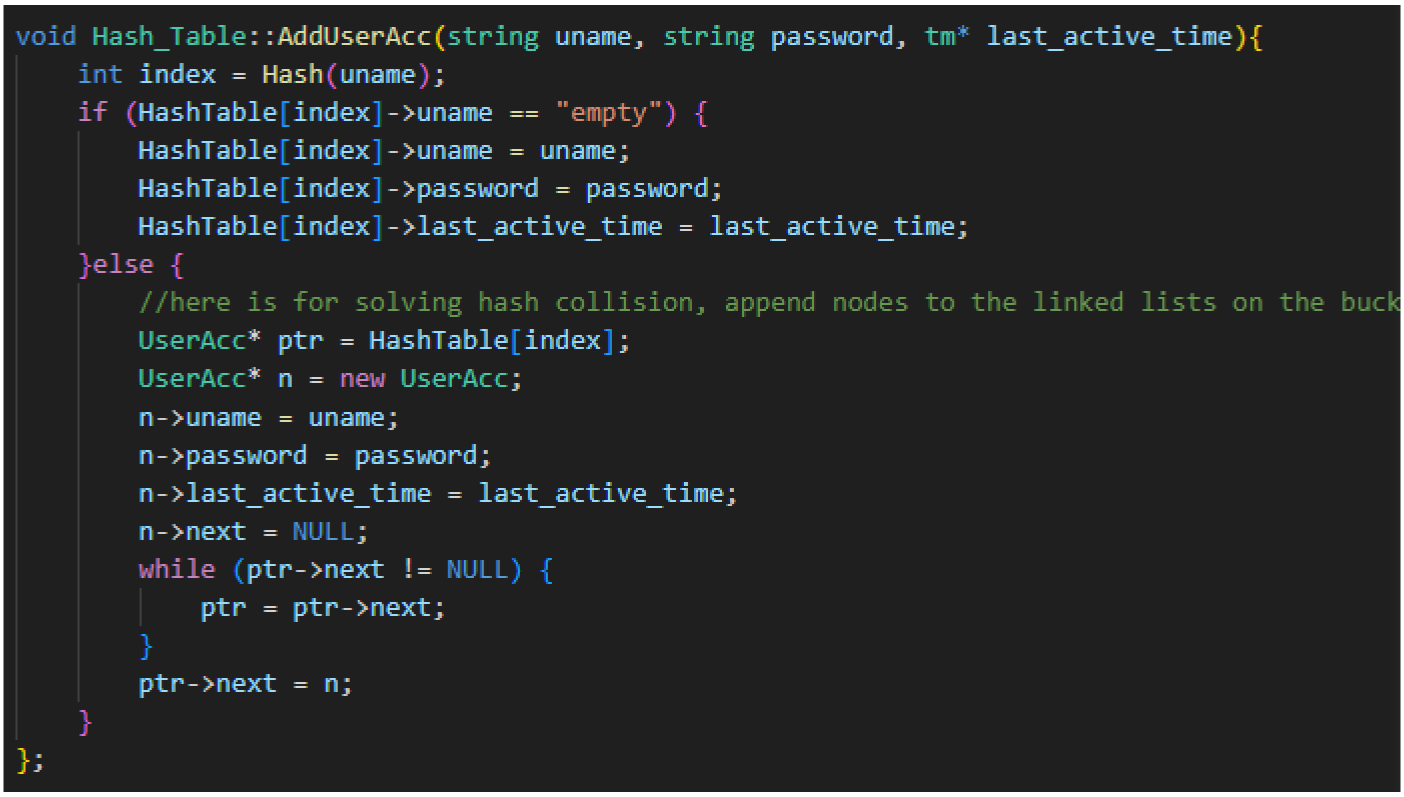
Hash() method is a hash function in the context of hash table. The hash function takes in username as the *key*, mapping the username to an index. The index is calculated based on the sum of ASCII number of each character in the username, *hash_sum*; then, *index = hash_sum % tableSize*, where *tableSize = 10*.



```
int Hash_Table::getTableSize() {
    return tableSize;
}

UserAcc* Hash_Table::getBucketAtIndex(int index) {
    return HashTable[index];
}
```

Figure 4.3.3: Two getter methods

These two getter methods: first is to retrieve a private attribute which is *tableSize*, second is to retrieve the bucket at specified index.

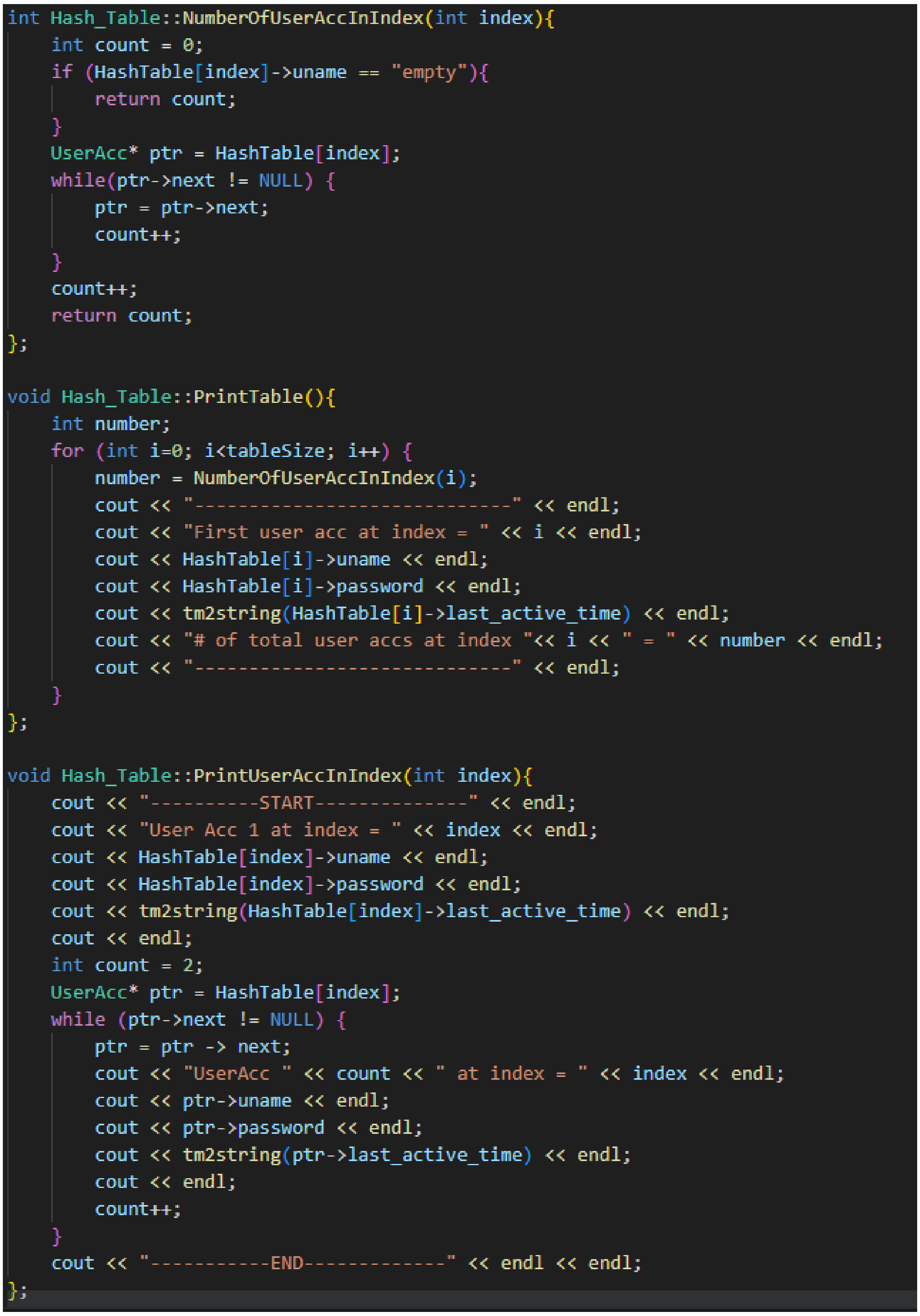


```
void Hash_Table::AddUserAcc(string uname, string password, tm* last_active_time){
    int index = Hash(uname);
    if (HashTable[index]->uname == "empty") {
        HashTable[index]->uname = uname;
        HashTable[index]->password = password;
        HashTable[index]->last_active_time = last_active_time;
    }else {
        //here is for solving hash collision, append nodes to the linked lists on the buck
        UserAcc* ptr = HashTable[index];
        UserAcc* n = new UserAcc;
        n->uname = uname;
        n->password = password;
        n->last_active_time = last_active_time;
        n->next = NULL;
        while (ptr->next != NULL) {
            ptr = ptr->next;
        }
        ptr->next = n;
    }
};
```

Figure 4.3.4: AddUserAcc() method

AddUserAcc() method is for adding new *UserAcc* node in the hash table. The username of the new node will be hashed first to get the index.

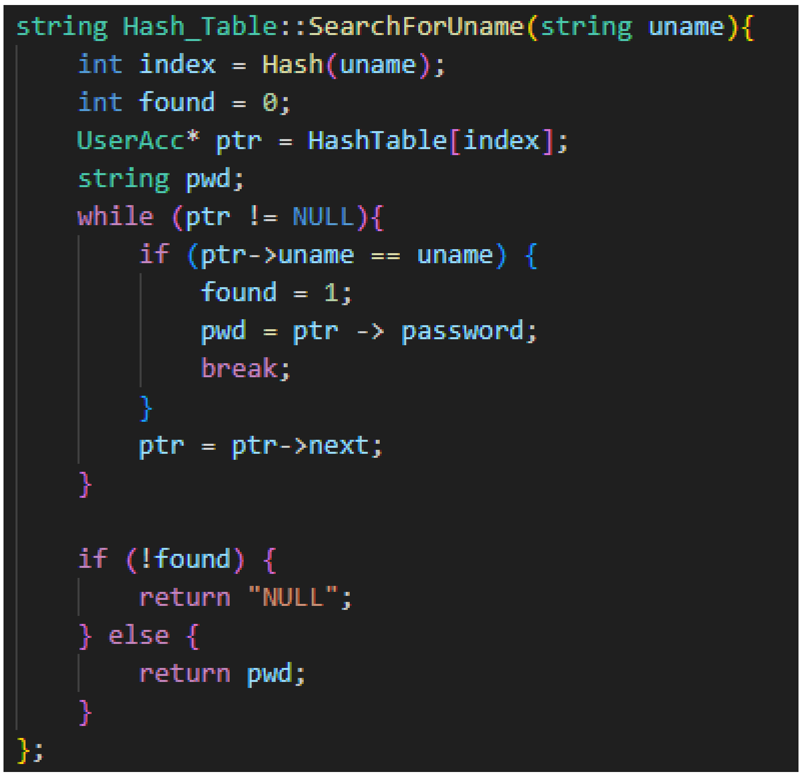
After getting the index, if the bucket is empty, then insert the new *UserAcc* node into the bucket; else traverse till the end of the linked list for insertion of new *UserAcc* node.



```cpp
int Hash_Table::NumberOfUserAccInIndex(int index){
    int count = 0;
    if (HashTable[index]->uname == "empty"){
        return count;
    }
    UserAcc* ptr = HashTable[index];
    while(ptr->next != NULL) {
        ptr = ptr->next;
        count++;
    }
    count++;
    return count;
};

void Hash_Table::PrintTable(){
    int number;
    for (int i=0; i<tableSize; i++) {
        number = NumberOfUserAccInIndex(i);
        cout << "------------------------------" << endl;
        cout << "First user acc at index = " << i << endl;
        cout << HashTable[i]->uname << endl;
        cout << HashTable[i]->password << endl;
        cout << tm2string(HashTable[i]->last_active_time) << endl;
        cout << "# of total user accs at index "<< i << " = " << number << endl;
        cout << "------------------------------" << endl;
    }
};

void Hash_Table::PrintUserAccInIndex(int index){
    cout << "----------START--------------" << endl;
    cout << "User Acc 1 at index = " << index << endl;
    cout << HashTable[index]->uname << endl;
    cout << HashTable[index]->password << endl;
    cout << tm2string(HashTable[index]->last_active_time) << endl;
    cout << endl;
    int count = 2;
    UserAcc* ptr = HashTable[index];
    while (ptr->next != NULL) {
        ptr = ptr -> next;
        cout << "UserAcc " << count << " at index = " << index << endl;
        cout << ptr->uname << endl;
        cout << ptr->password << endl;
        cout << tm2string(ptr->last_active_time) << endl;
        cout << endl;
        count++;
    }
    cout << "-----------END-------------" << endl << endl;
};
```

Figure 4.3.5: Three methods for Debugging Purposes

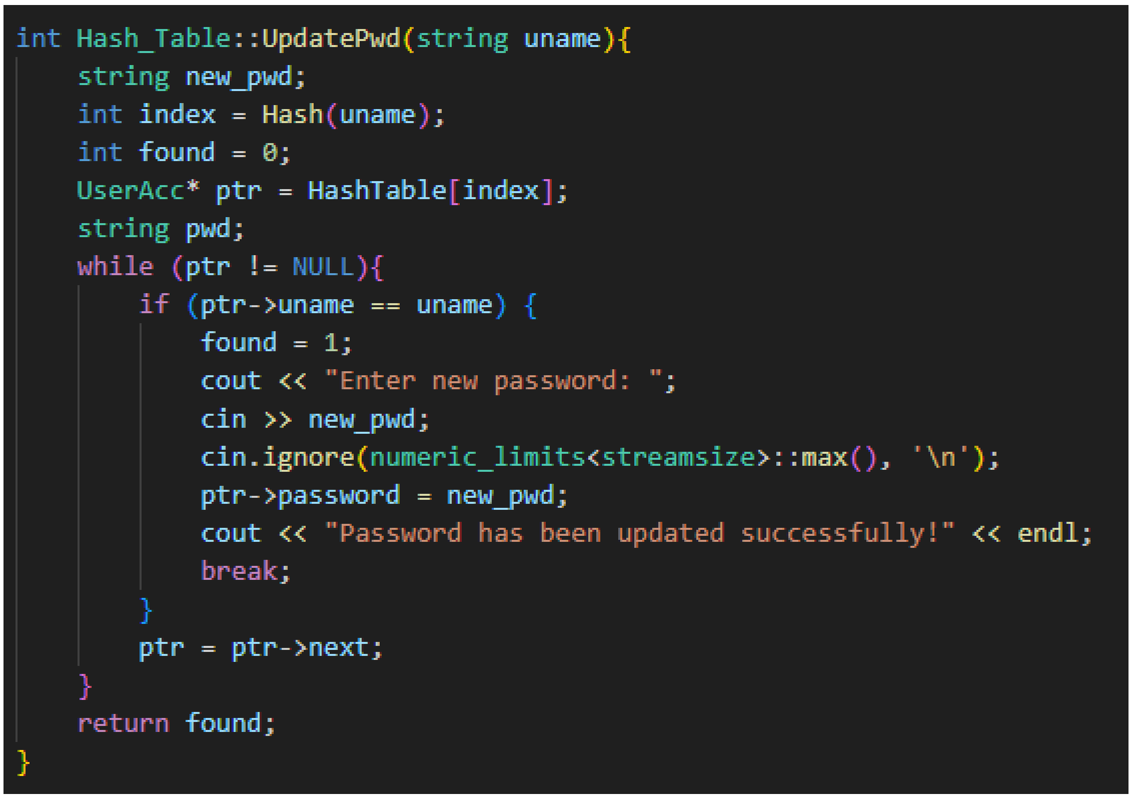
All the three methods shown in Figure 4.3.5 are for debugging purposes, they are for testing if the hash table is working as expected. *NumberOfUserAccInIndex()* is to get the number of *UserAcc* nodes in the specific index of the bucket. *PrintTable()* is to print out all the *UserAcc* nodes in the hash table. *PrintUserAccInIndex()* is to print all the *UserAcc* nodes in the specific index of bucket.



```
string Hash_Table::SearchForUname(string uname){
    int index = Hash(uname);
    int found = 0;
    UserAcc* ptr = HashTable[index];
    string pwd;
    while (ptr != NULL){
        if (ptr->uname == uname) {
            found = 1;
            pwd = ptr -> password;
            break;
        }
        ptr = ptr->next;
    }

    if (!found) {
        return "NULL";
    } else {
        return pwd;
    }
};
```

Figure 4.3.6: SearchForUname() method

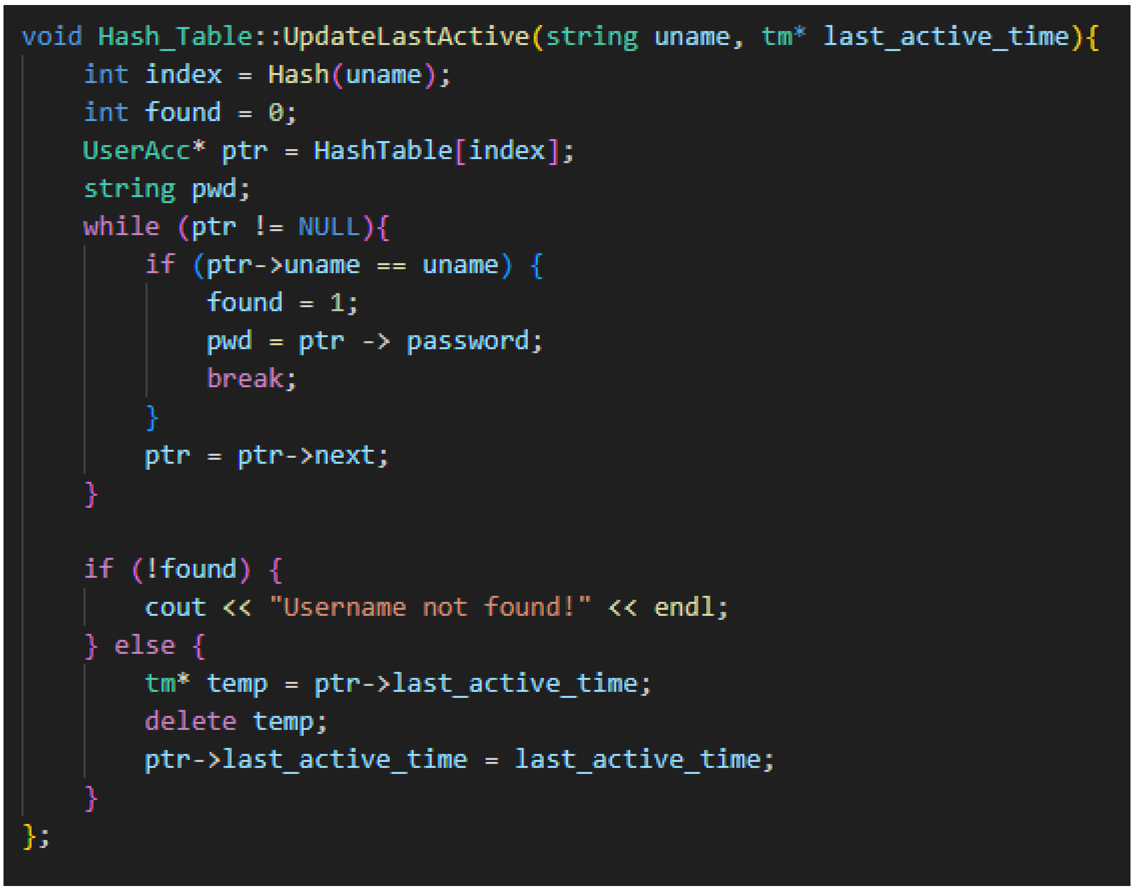
Figure 4.3.6 shows the *SearchForUname()* method which is for looking for a specific *UserAcc* node matching the *uname*. The *uname* will be passed into hash function first, then come up with the index of the bucket. With that index of bucket, all the nodes on the linked list will be traversed through to see if there is any matching *UserAcc* node. If there is no matching node, return a string “NULL”; Else return its password so that further validation of the entered credentials can be done.



```cpp
int Hash_Table::UpdatePwd(string uname){
    string new_pwd;
    int index = Hash(uname);
    int found = 0;
    UserAcc* ptr = HashTable[index];
    string pwd;
    while (ptr != NULL){
        if (ptr->uname == uname) {
            found = 1;
            cout << "Enter new password: ";
            cin >> new_pwd;
            cin.ignore(numeric_limits<streamsize>::max(), '\n');
            ptr->password = new_pwd;
            cout << "Password has been updated successfully!" << endl;
            break;
        }
        ptr = ptr->next;
    }
    return found;
}
```

Figure 4.3.7: UpdatePwd() method

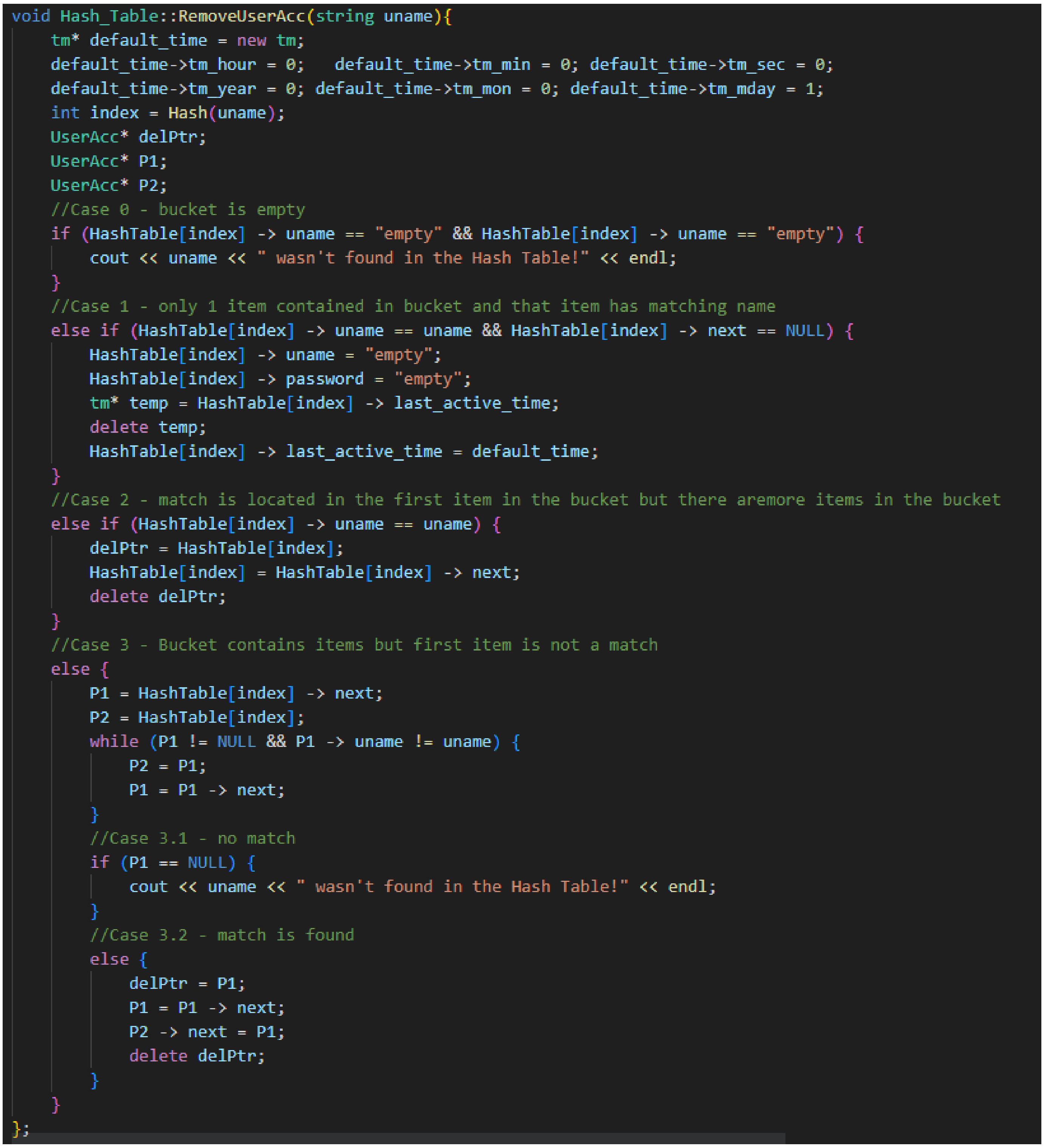
UpdatePwd() method is similar to *SearchForUname()* method, which is about looking for a matching node. If the matching node is found, the admin will be prompted for a new password, so that the admin can modify the customer's details. Otherwise, the 0 will be returned by the method for displaying warning to the admin.



```
void Hash_Table::UpdateLastActive(string uname, tm* last_active_time){
    int index = Hash(uname);
    int found = 0;
    UserAcc* ptr = HashTable[index];
    string pwd;
    while (ptr != NULL){
        if (ptr->uname == uname) {
            found = 1;
            pwd = ptr -> password;
            break;
        }
        ptr = ptr->next;
    }

    if (!found) {
        cout << "Username not found!" << endl;
    } else {
        tm* temp = ptr->last_active_time;
        delete temp;
        ptr->last_active_time = last_active_time;
    }
};
```

Figure 4.3.8: UpdateLastActive() method

UpdateLastActive() method is for updating the last active time of admins and customers. Once admins and customers login to the system, this method will be called, and look for matching *UserAcc* node. The last active time of the UserAcc node will be updated using the time admins or customers login to the system, old last active time will be deleted from the heap using **delete** operator to avoid memory leakage.



```
void Hash_Table::RemoveUserAcc(string uname){
    tm* default_time = new tm;
    default_time->tm_hour = 0;   default_time->tm_min = 0; default_time->tm_sec = 0;
    default_time->tm_year = 0; default_time->tm_mon = 0; default_time->tm_mday = 1;
    int index = Hash(uname);
    UserAcc* delPtr;
    UserAcc* P1;
    UserAcc* P2;
    //Case 0 - bucket is empty
    if (HashTable[index] -> uname == "empty" && HashTable[index] -> uname == "empty") {
        cout << uname << " wasn't found in the Hash Table!" << endl;
    }
    //Case 1 - only 1 item contained in bucket and that item has matching name
    else if (HashTable[index] -> uname == uname && HashTable[index] -> next == NULL) {
        HashTable[index] -> uname = "empty";
        HashTable[index] -> password = "empty";
        tm* temp = HashTable[index] -> last_active_time;
        delete temp;
        HashTable[index] -> last_active_time = default_time;
    }
    //Case 2 - match is located in the first item in the bucket but there aremore items in the bucket
    else if (HashTable[index] -> uname == uname) {
        delPtr = HashTable[index];
        HashTable[index] = HashTable[index] -> next;
        delete delPtr;
    }
    //Case 3 - Bucket contains items but first item is not a match
    else {
        P1 = HashTable[index] -> next;
        P2 = HashTable[index];
        while (P1 != NULL && P1 -> uname != uname) {
            P2 = P1;
            P1 = P1 -> next;
        }
        //Case 3.1 - no match
        if (P1 == NULL) {
            cout << uname << " wasn't found in the Hash Table!" << endl;
        }
        //Case 3.2 - match is found
        else {
            delPtr = P1;
            P1 = P1 -> next;
            P2 -> next = P1;
            delete delPtr;
        }
    }
};
```

Figure 4.3.9: RemoveUserAcc() method

Figure 4.3.9 shows the source code of *RemoveUserAcc()* method, this is for removing a *UserAcc* node from the hash table. Firstly, the method gets the default time again for updating the bucket's last active time if there is one node on the linked list before removal.

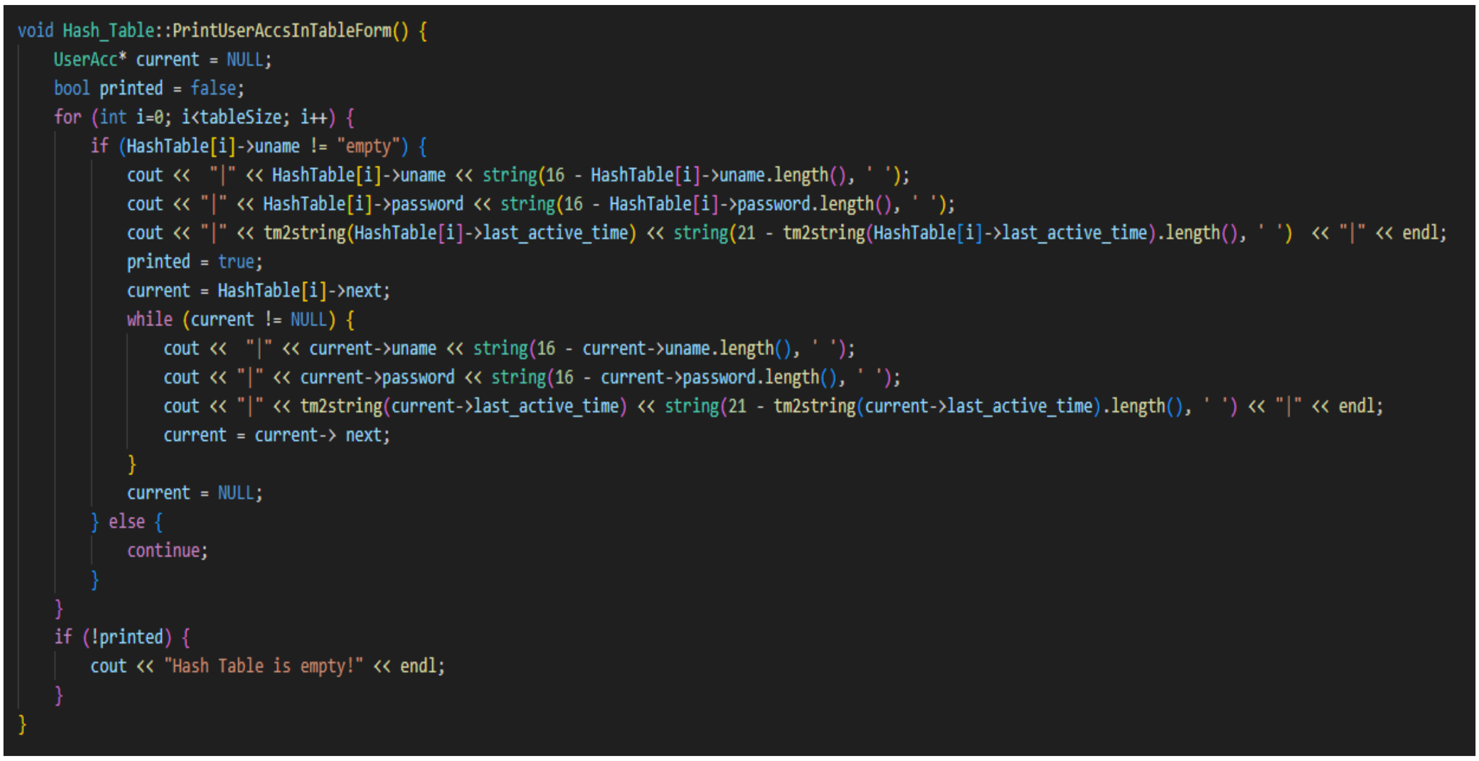
There are 4 cases to be handled while trying to remove a *UserAcc* node: bucket is empty, only 1 node is contained in the bucket and the node has matching name, match is located in the first node but there are more nodes in the bucket, and bucket contains nodes but first node is not a match.

In first case, just raise a warning that the username is not found.

In second case, the username and password of the bucket are set back to "empty" again, its last active time is set back to default time again. The old time will be deleted from heap to avoid memory leakage.

In third case, the pointer to matching node is kept in *delPtr*, then next node of the matching node will be set as the first node of the linked list. Then, delete the *delPtr* from heap.

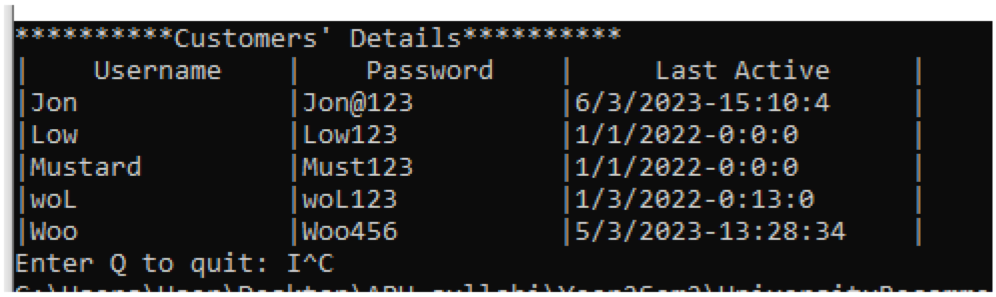
In fourth case, *P2* keeps pointer to previous node of the matching node (it could be pointer to NULL if there is no matching node), while *P1* keeps pointer to the matching node (it could be pointer to NULL if there is no matching node). There will be two more subcases. First subcase is there is no matching node, then raise a warning; Second subcase is there is a matching node, then the nodes will be pointing to appropriate nodes and the matching node will be deleted from heap.



```
void Hash_Table::PrintUserAccsInTableForm() {
    UserAcc* current = NULL;
    bool printed = false;
    for (int i=0; i<tableSize; i++) {
        if (HashTable[i]->uname != "empty") {
            cout <<  "|" << HashTable[i]->uname << string(16 - HashTable[i]->uname.length(), ' ');
            cout << "|" << HashTable[i]->password << string(16 - HashTable[i]->password.length(), ' ');
            cout << "|" << tm2string(HashTable[i]->last_active_time) << string(21 - tm2string(HashTable[i]->last_active_time).length(), ' ')  << "|" << endl;
            printed = true;
            current = HashTable[i]->next;
            while (current != NULL) {
                cout <<  "|" << current->uname << string(16 - current->uname.length(), ' ');
                cout << "|" << current->password << string(16 - current->password.length(), ' ');
                cout << "|" << tm2string(current->last_active_time) << string(21 - tm2string(current->last_active_time).length(), ' ') << "|" << endl;
                current = current-> next;
            }
            current = NULL;
        } else {
            continue;
        }
    }
    if (!printed) {
        cout << "Hash Table is empty!" << endl;
    }
}
```

Figure 4.3.10: PrintUserAccsInTableForm() method

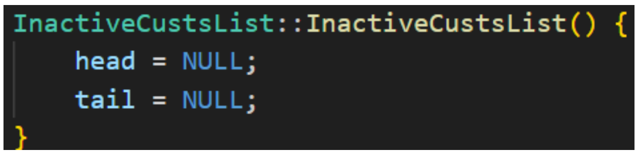
This last method of hash table is for printing information of all users in appropriate table form as shown in *Figure 4.3.11*. In this project, this method will be used for printing information of all customers only. *string()* function is used to fill in the remaining gaps of each entry in the table.



```
**********Customers' Details**********
|    Username     |     Password     |      Last Active      |
|Jon              |Jon@123           |6/3/2023-15:10:4       |
|Low              |Low123            |1/1/2022-0:0:0         |
|Mustard          |Must123           |1/1/2022-0:0:0         |
|woL              |woL123            |1/3/2022-0:13:0        |
|Woo              |Woo456            |5/3/2023-13:28:34      |
Enter Q to quit: I^C
```

Figure 4.3.11: Printing Information of All Customers in Table Form

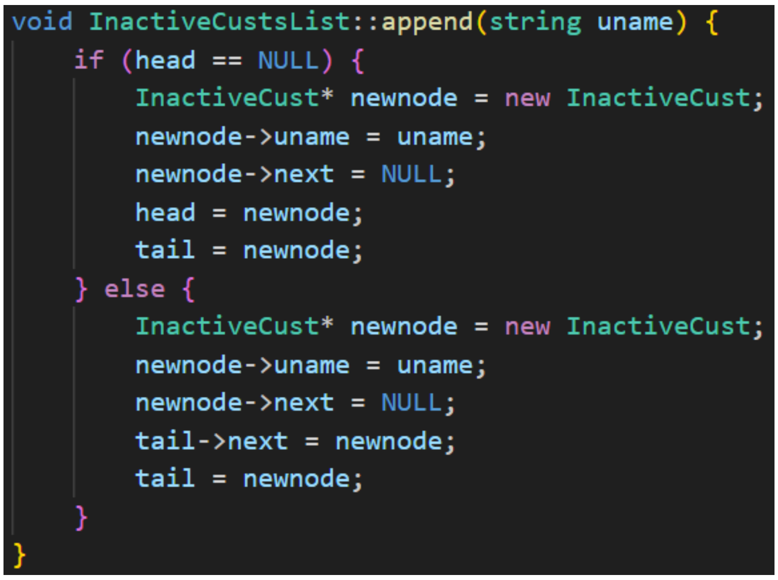
4.4 InactiveCustList class



```
InactiveCustsList::InactiveCustsList() {
    head = NULL;
    tail = NULL;
}
```

Figure 4.4.1: InactiveCustsList() constructor

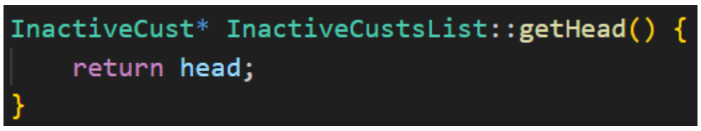
The constructor *InactiveCustsList()* initializes the linked list by setting both the head and tail pointers to NULL. This indicates that the linked list is empty upon initialization, as there are no nodes present.



```
void InactiveCustsList::append(string uname) {
    if (head == NULL) {
        InactiveCust* newnode = new InactiveCust;
        newnode->uname = uname;
        newnode->next = NULL;
        head = newnode;
        tail = newnode;
    } else {
        InactiveCust* newnode = new InactiveCust;
        newnode->uname = uname;
        newnode->next = NULL;
        tail->next = newnode;
        tail = newnode;
    }
}
```

Figure 4.4.2: append() method

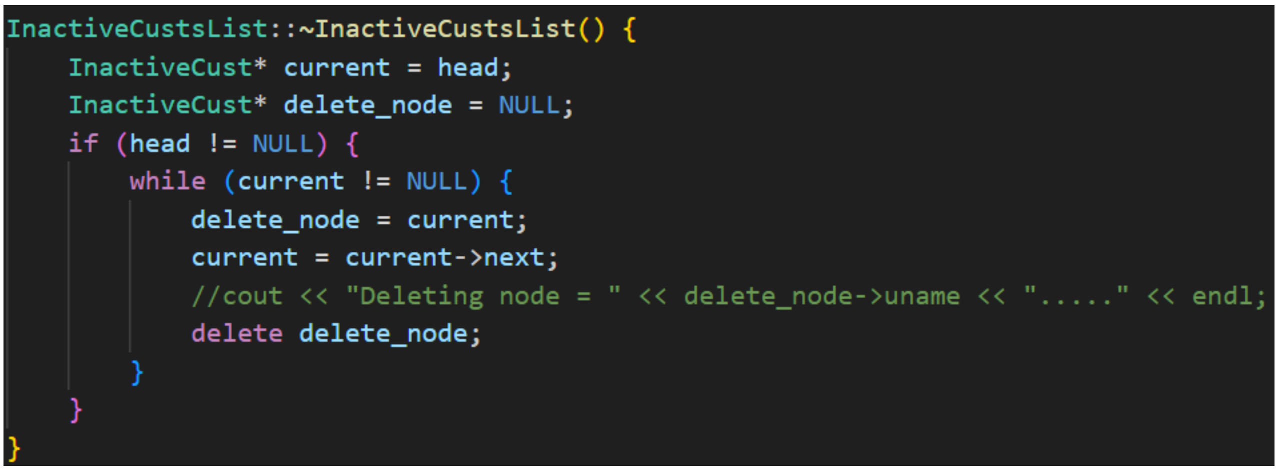
The append method adds a new customer to the linked list by creating a new node and updating the necessary pointers. If the list is currently empty (i.e., the head pointer is NULL), the method creates a new node of type *InactiveCust* using the new operator and assigns it to the pointer variable *newnode*. The *uname* value is then assigned to the *uname* member of the newly created node. Both the next pointer of *newnode* and the head and tail pointers of the list are set to NULL, as this is the first and only node in the list. If the list is not empty, the method follows a similar process. It creates a new node with the provided *uname* value and sets the next pointer of the current tail node (the previous last node in the list) to point to the new node. Then, the tail pointer is updated to point to the newly added node, making it the new last node in the list.



```
InactiveCust* InactiveCustsList::getHead() {
    return head;
}
```

Figure 4.4.3: getHead() method

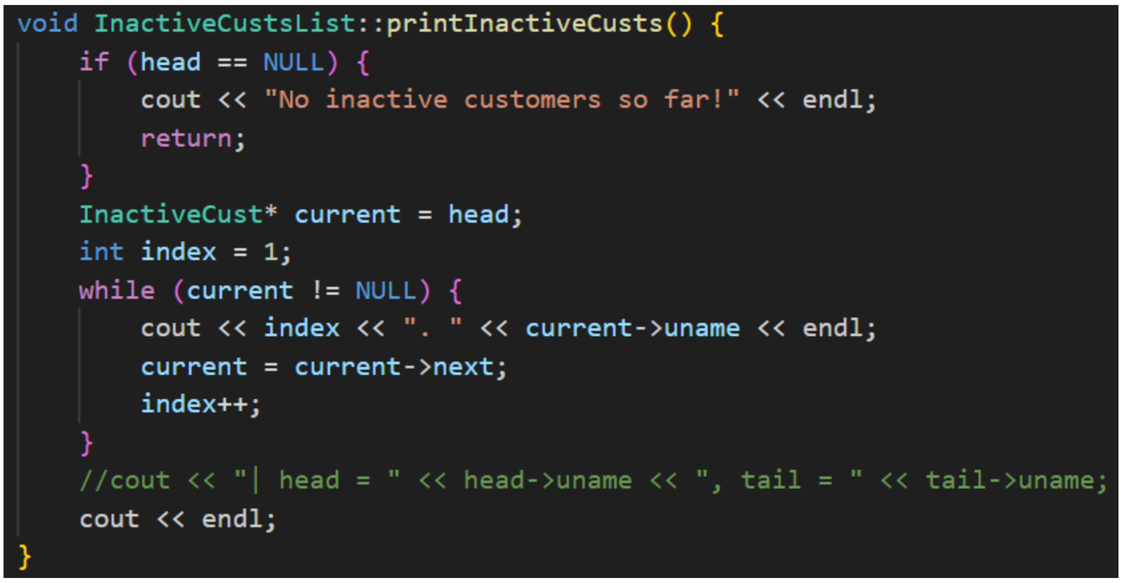
The *getHead* method is used to retrieve the *head* pointer of the *InactiveCustsList* linked list.



```
InactiveCustsList::~InactiveCustsList() {
    InactiveCust* current = head;
    InactiveCust* delete_node = NULL;
    if (head != NULL) {
        while (current != NULL) {
            delete_node = current;
            current = current->next;
            //cout << "Deleting node = " << delete_node->uname << "....." << endl;
            delete delete_node;
        }
    }
}
```

Figure 4.4.4: InactiveCustsList() destructor

The *~InactiveCustsList* destructor is responsible for cleaning up the memory allocated for the linked list. The destructor iterates through each node in the linked list, starting from the *head* and deleting each node to release the allocated memory. It uses a loop that continues until the *current* pointer becomes NULL, indicating that the end of the list has been reached. Within the loop, the current node is assigned to *delete_node* to mark it for deletion. Then, the current pointer is updated to point to the next node in the list. Finally, the *delete* operator is used to free the memory allocated for *delete_node*.



```
void InactiveCustsList::printInactiveCusts() {
    if (head == NULL) {
        cout << "No inactive customers so far!" << endl;
        return;
    }
    InactiveCust* current = head;
    int index = 1;
    while (current != NULL) {
        cout << index << ". " << current->uname << endl;
        current = current->next;
        index++;
    }
    //cout << "| head = " << head->uname << ", tail = " << tail->uname;
    cout << endl;
}
```

Figure 4.4.5: printInactiveCusts() for Debugging Purposes

The *printInactiveCusts()* method iterates through the linked list and prints the usernames of the inactive customers. If the list is empty (i.e., *head* is NULL), it prints a message indicating that there are no inactive customers. If the list is not empty, the method uses a *while* loop to traverse the list. It starts from the *head* node and prints the index (starting from 1) and the username of each node. Then, it moves to the next node by updating the *current* pointer. The *index* is incremented for each iteration.

4.5 PreferredUniQueue class

PreferredUniQueue class is a priority queue for sorting the universities based on the number of saves by customers in the system.

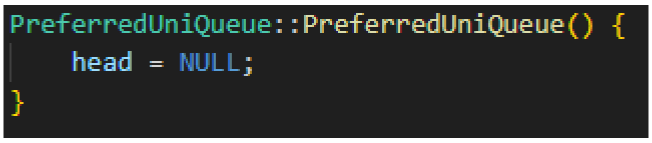

```
PreferredUniQueue::PreferredUniQueue() {
    head = NULL;
}
```

Figure 4.5.1: PreferredUniQueue() constructor

PreferredUniQueue() constructor is for initializing the *head*, by setting it to point to NULL since initially there is no node in the priority queue.

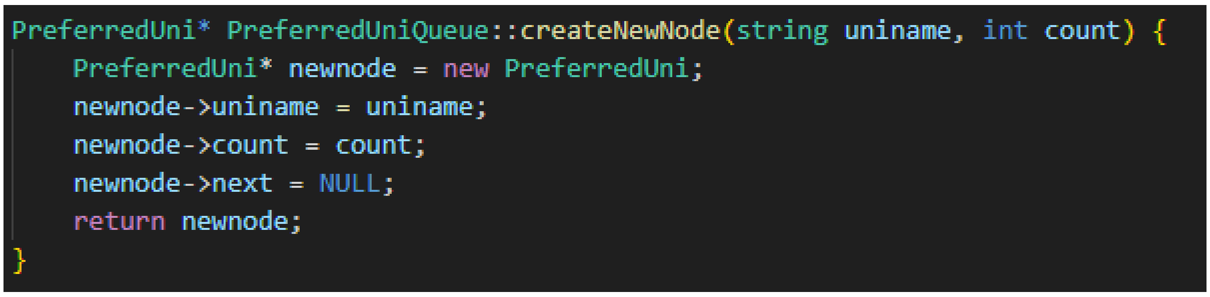

```
PreferredUni* PreferredUniQueue::createNewNode(string uniname, int count) {
    PreferredUni* newnode = new PreferredUni;
    newnode->uniname = uniname;
    newnode->count = count;
    newnode->next = NULL;
    return newnode;
}
```

Figure 4.5.2: createNewNode() method

createNewNode() method is for creating a new node in the priority queue. Spaces from heap will be allocated to the new node by using the **new** operator. The fields on the new *PreferredUni* node will be initialized as well.

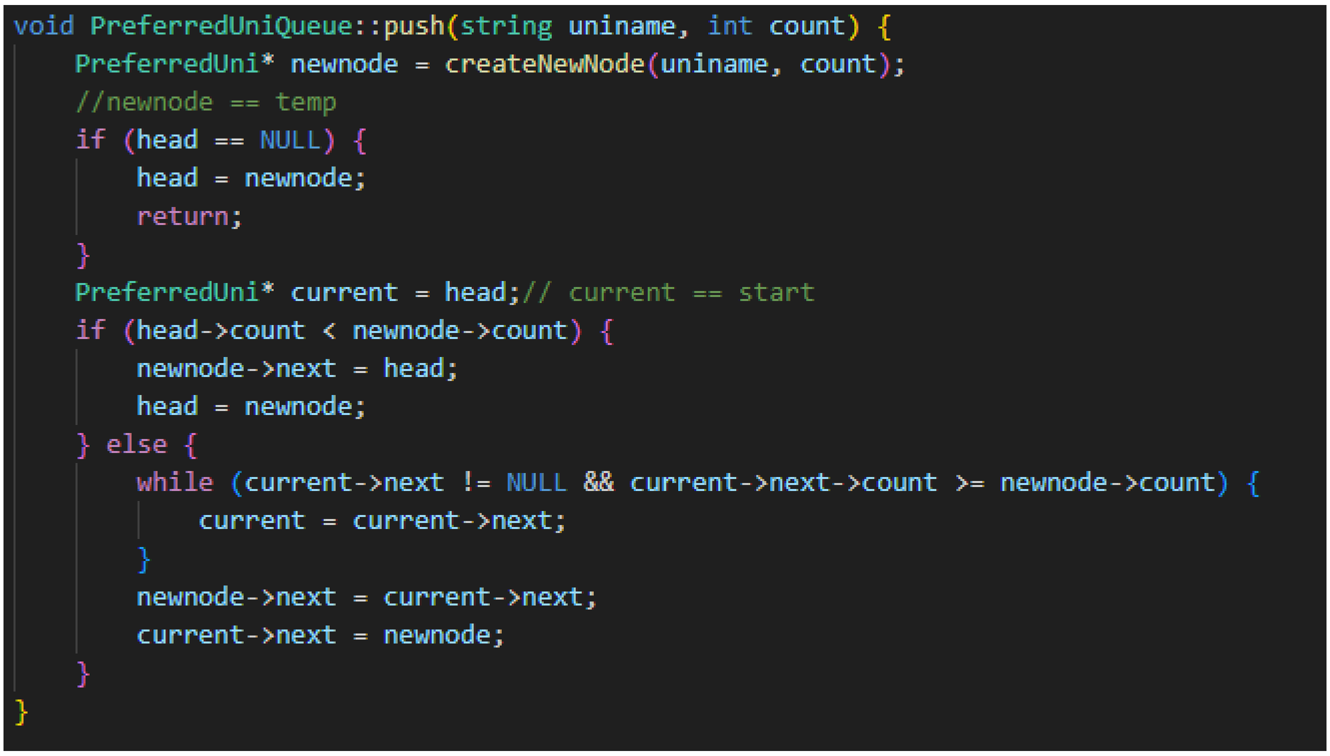
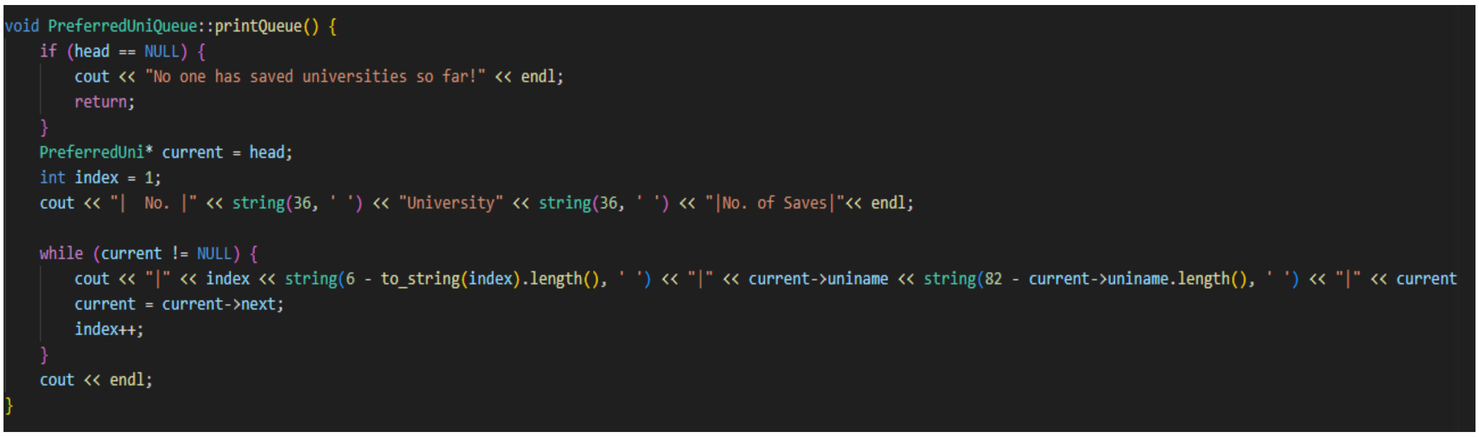

```
void PreferredUniQueue::push(string uniname, int count) {
    PreferredUni* newnode = createNewNode(uniname, count);
    //newnode == temp
    if (head == NULL) {
        head = newnode;
        return;
    }
    PreferredUni* current = head;// current == start
    if (head->count < newnode->count) {
        newnode->next = head;
        head = newnode;
    } else {
        while (current->next != NULL && current->next->count >= newnode->count) {
            current = current->next;
        }
        newnode->next = current->next;
        current->next = newnode;
    }
}
```

Figure 4.5.3: push() method

push() method is for inserting the new node at appropriate location of the queue based on its priority, at this case, the nodes are arranged from high *count* to low *count*.

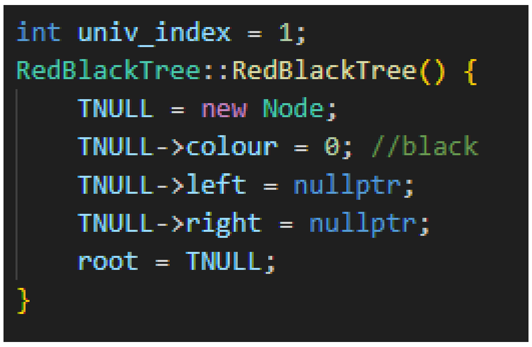


```cpp
void PreferredUniQueue::printQueue() {
    if (head == NULL) {
        cout << "No one has saved universities so far!" << endl;
        return;
    }
    PreferredUni* current = head;
    int index = 1;
    cout << "|  No. |" << string(36, ' ') << "University" << string(36, ' ') << "|No. of Saves|"<< endl;

    while (current != NULL) {
        cout << "|" << index << string(6 - to_string(index).length(), ' ') << "|" << current->uniname << string(82 - current->uniname.length(), ' ') << "|" << current
        current = current->next;
        index++;
    }
    cout << endl;
}
```

Figure 4.5.4: printQueue() method

printQueue() method is for printing the most preferred universities in descending order based on number of saves.

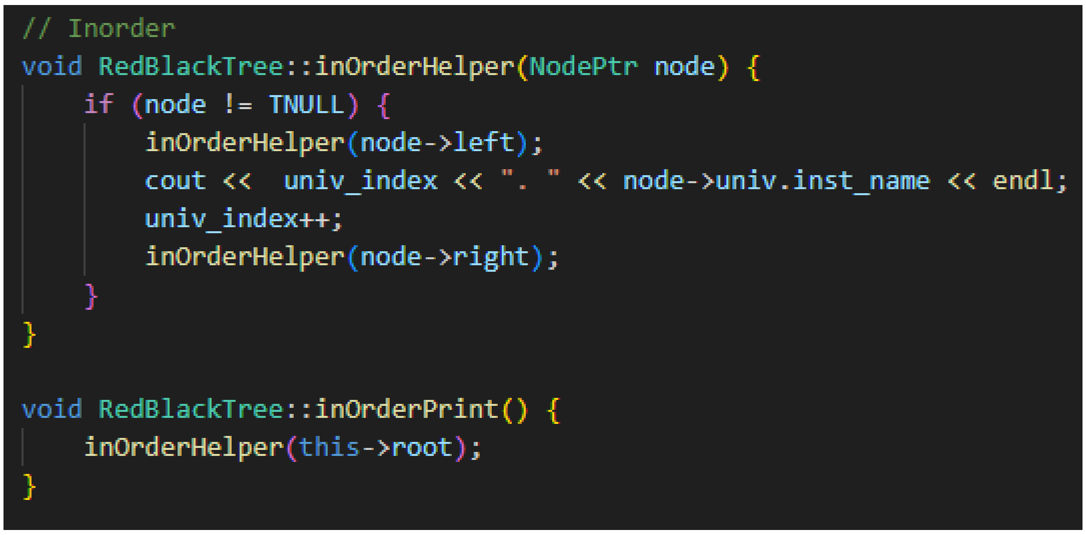
4.6 RedBlackTree class



```cpp
int univ_index = 1;
RedBlackTree::RedBlackTree() {
    TNULL = new Node;
    TNULL->colour = 0; //black
    TNULL->left = nullptr;
    TNULL->right = nullptr;
    root = TNULL;
}
```

Figure 4.6.1: RedBlackTree() constructor

RedBlackTree() constructor is used to create the *TNULL* node, which was one of the properties of red black tree as mentioned in Section 3.3. The root of the tree is initialized as the TNULL node.



```cpp
// Inorder
void RedBlackTree::inOrderHelper(NodePtr node) {
    if (node != TNULL) {
        inOrderHelper(node->left);
        cout <<  univ_index << ". " << node->univ.inst_name << endl;
        univ_index++;
        inOrderHelper(node->right);
    }
}

void RedBlackTree::inOrderPrint() {
    inOrderHelper(this->root);
}
```

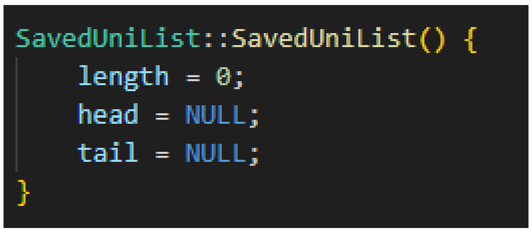
Figure 4.6.2: inOrderPrint() and inOrderHelper() methods

Figure 4.6.2 shows the methods for performing in order traversal of the red black tree. This traversal method can print out the universities in ascending order based on university names. Left subtree is traversed first, followed by root, finally right subtree is traversed.

The remaining methods of *RedBlackTree* class are explained in section 5.1 and section 5.3 since they are related to sort and search.

4.7 SavedUniList class

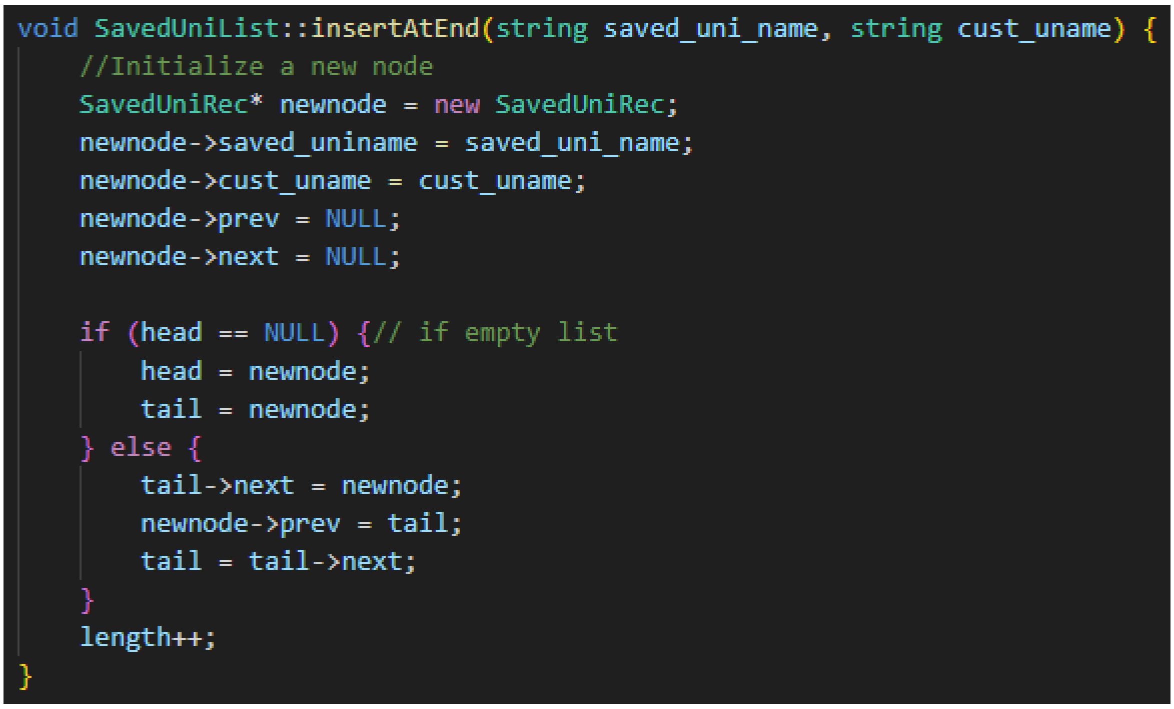
SavedUniList class is a doubly linked list for keeping all the saved universities by customers in the system.



```
SavedUniList::SavedUniList() {
    length = 0;
    head = NULL;
    tail = NULL;
}
```

Figure 4.7.1: SavedUniList() constructor

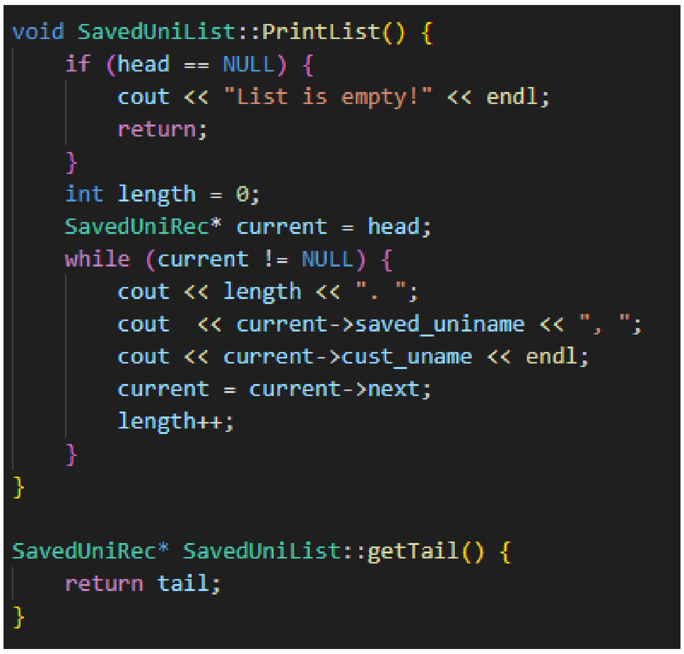
Figure 4.7.1 shows the constructor to initialize the length of the doubly linked list to 0. Both *head* and *tail* should point to NULL.



```
void SavedUniList::insertAtEnd(string saved_uni_name, string cust_uname) {
    //Initialize a new node
    SavedUniRec* newnode = new SavedUniRec;
    newnode->saved_uniname = saved_uni_name;
    newnode->cust_uname = cust_uname;
    newnode->prev = NULL;
    newnode->next = NULL;

    if (head == NULL) {// if empty list
        head = newnode;
        tail = newnode;
    } else {
        tail->next = newnode;
        newnode->prev = tail;
        tail = tail->next;
    }
    length++;
}
```

Figure 4.7.2: insertAtEnd() method

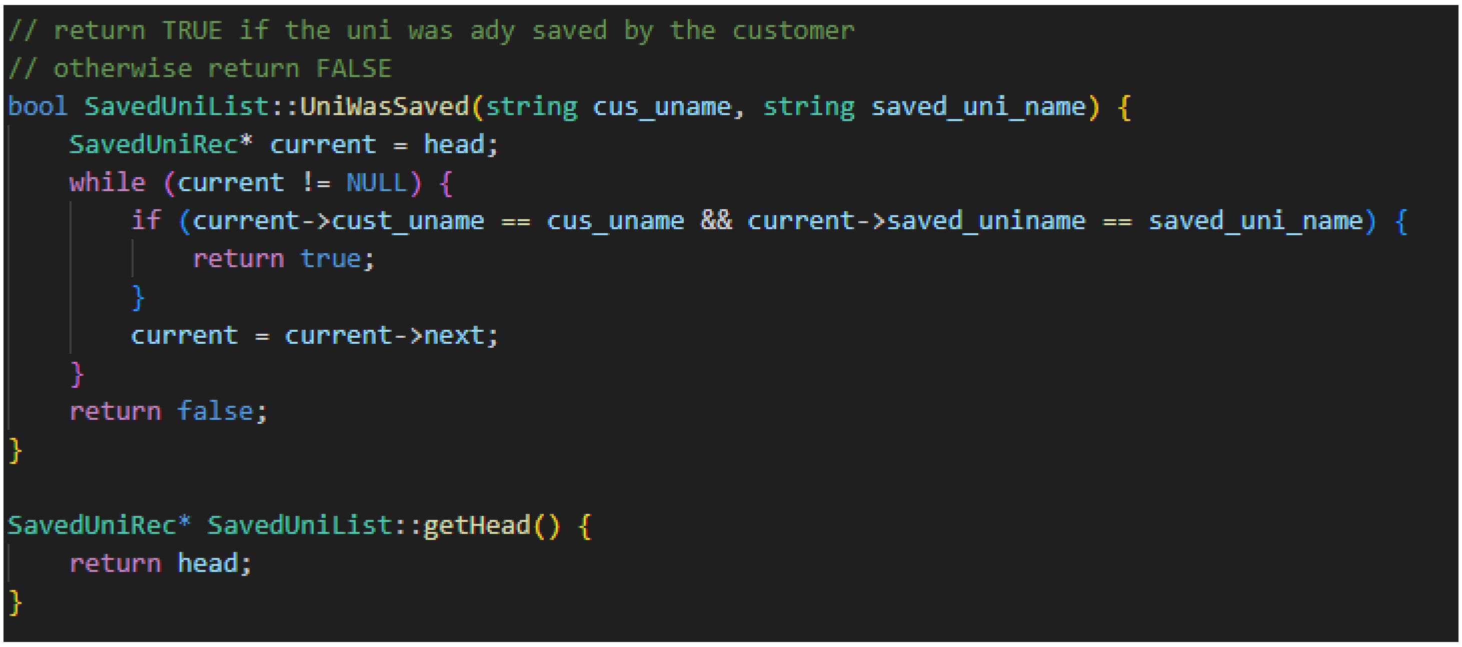
insertAtEnd() method is to insert a new *SavedUniRec* node at the end of doubly linked list. Firstly, a new *SavedUniRec* node is kept at the heap using the **new** operator, then its fields are all initialized. If it is an empty list, then set both head and tail to point to the new node; Otherwise, the *tail* attribute of the class is fully utilized here to ensure faster insertion at the end, so that there is no need to traverse from the front to the end of the linked list.



```cpp
void SavedUniList::PrintList() {
    if (head == NULL) {
        cout << "List is empty!" << endl;
        return;
    }
    int length = 0;
    SavedUniRec* current = head;
    while (current != NULL) {
        cout << length << ". ";
        cout  << current->saved_uniname << ", ";
        cout << current->cust_uname << endl;
        current = current->next;
        length++;
    }
}

SavedUniRec* SavedUniList::getTail() {
    return tail;
}
```

Figure 4.7.3: PrintList() and getTail() method

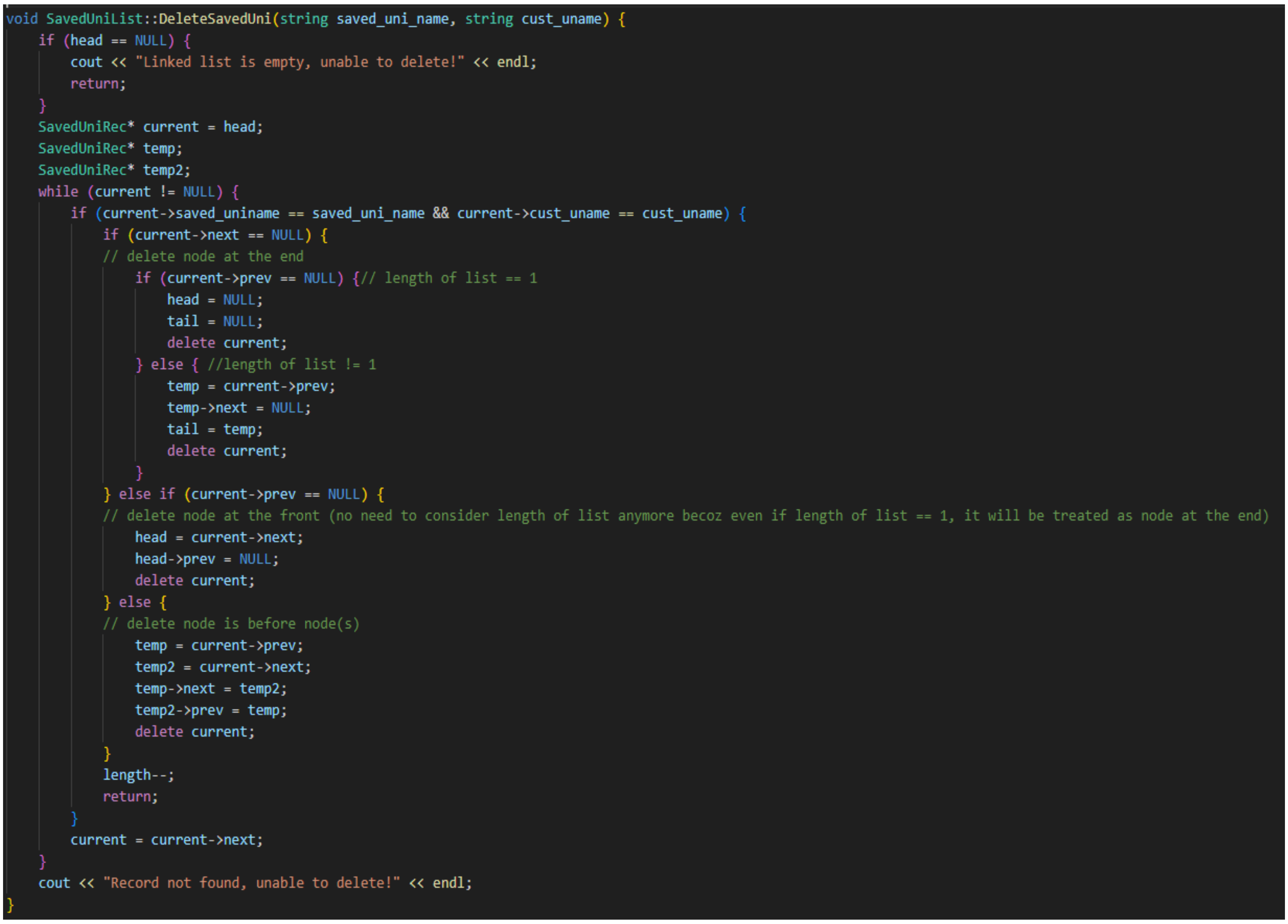
The *PrintList()* is just using a while loop to traverse from front to end of linked list, and print out information of each *SavedUniRec* node. *getTail()* method is just to get the last *SavedUniRec* node, so that printing all records in reverse order is feasible.



```cpp
// return TRUE if the uni was ady saved by the customer
// otherwise return FALSE
bool SavedUniList::UniWasSaved(string cus_uname, string saved_uni_name) {
    SavedUniRec* current = head;
    while (current != NULL) {
        if (current->cust_uname == cus_uname && current->saved_uniname == saved_uni_name) {
            return true;
        }
        current = current->next;
    }
    return false;
}

SavedUniRec* SavedUniList::getHead() {
    return head;
}
```

Figure 4.7.4: UniWasSaved() method

UniWasSaved() method returns True if the university has been saved by the customer, otherwise returns False. This method can be used to avoid customers from saving a university twice and this would lead the extra insertion into the doubly linked list, eventually leading to waste of memory. The linked list is traversed from front to end to see if there is any matching node, if there is, stop the *while* loop and returns True. After traversing the whole linked list, it means that there is no matching node then returns False.



```cpp
void SavedUniList::DeleteSavedUni(string saved_uni_name, string cust_uname) {
    if (head == NULL) {
        cout << "Linked list is empty, unable to delete!" << endl;
        return;
    }
    SavedUniRec* current = head;
    SavedUniRec* temp;
    SavedUniRec* temp2;
    while (current != NULL) {
        if (current->saved_uniname == saved_uni_name && current->cust_uname == cust_uname) {
            if (current->next == NULL) {
            // delete node at the end
                if (current->prev == NULL) {// length of list == 1
                    head = NULL;
                    tail = NULL;
                    delete current;
                } else { //length of list != 1
                    temp = current->prev;
                    temp->next = NULL;
                    tail = temp;
                    delete current;
                }
            } else if (current->prev == NULL) {
            // delete node at the front (no need to consider length of list anymore becoz even if length of list == 1, it will be treated as node at the end)
                head = current->next;
                head->prev = NULL;
                delete current;
            } else {
            // delete node is before node(s)
                temp = current->prev;
                temp2 = current->next;
                temp->next = temp2;
                temp2->prev = temp;
                delete current;
            }
            length--;
            return;
        }
        current = current->next;
    }
    cout << "Record not found, unable to delete!" << endl;
}
```

Figure 4.7.5: DeleteSavedUni() method

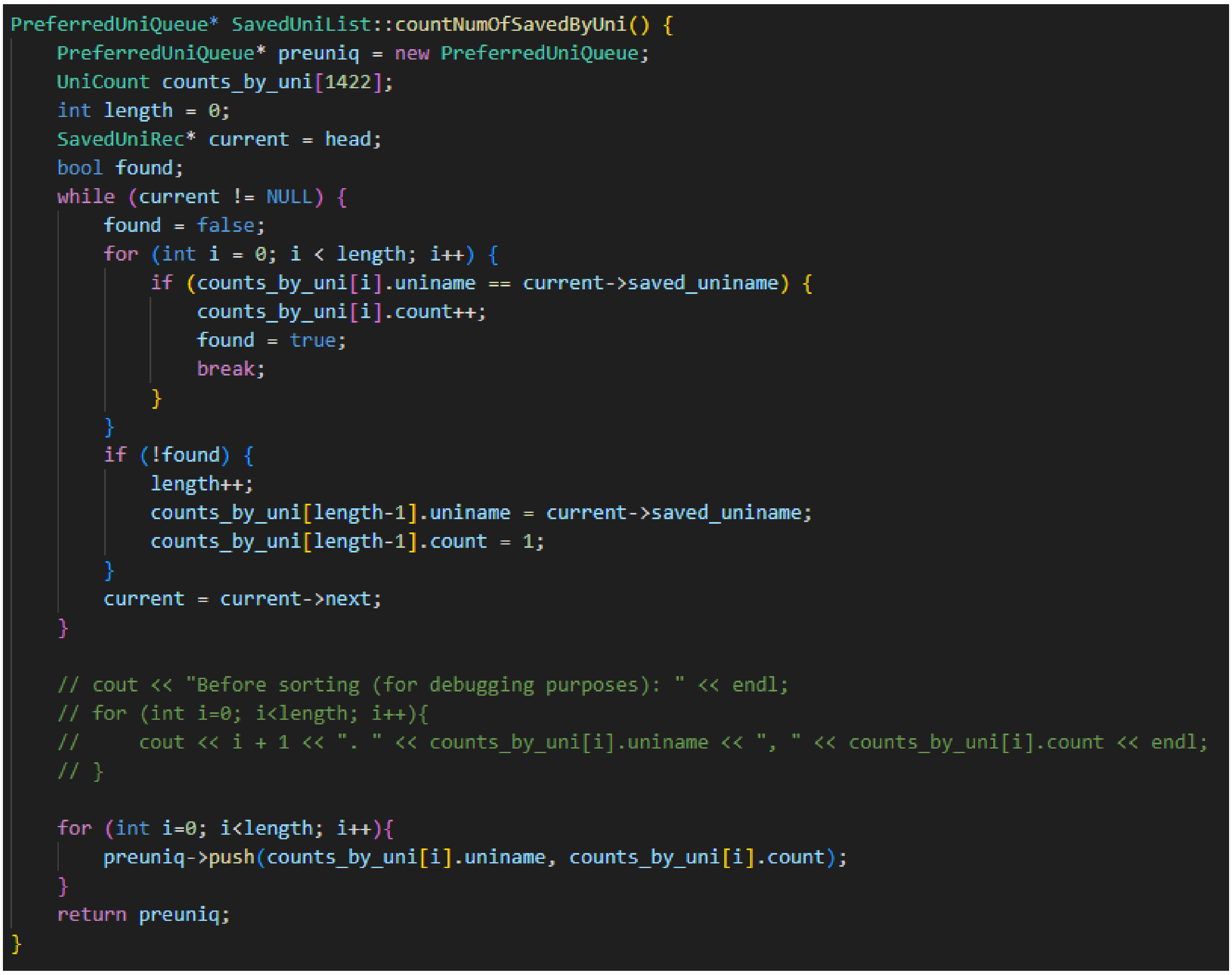
DeleteSavedUni() method is to delete a saved university by a customer from his/her favourite list. A *while* loop is used to traverse the linked list from front to end, if there is a matching node, then there will be 3 subcases: the matching node is at the end of list, the matching node is at the front of list, and the matching node is between 2 nodes. The *temp* and *temp2* are to keep reference to previous node and next node of the matching node to allow faster deletion.

If the matching node is at the end of list, then there will be 2 subcases again: length of list equals 1 (which means this node is also the head of list), and length of list more than 1. Slightly different methods will be used to handle the different subcases as shown in *Figure 4.7.5*, finally a **delete** operator is used to free the allocated heap memory.

If the matching node is at the front of list, the *head* will be pointing to next node of matching node, then the matching node will be freed from the heap using **delete** operator.

If the matching node is between 2 nodes, then the *temp* and *temp2* will be pointing to previous node and next node of the matching node respectively. Then the next node of *temp* should be updated as *temp2*, previous node of *temp2* should be updated as *temp* as well.

After handling all these subcases, the length of the linked list should be decreased by 1. Then stop the while loop since the matching node is found.



```
PreferredUniQueue* SavedUniList::countNumOfSavedByUni() {
    PreferredUniQueue* preuniq = new PreferredUniQueue;
    UniCount counts_by_uni[1422];
    int length = 0;
    SavedUniRec* current = head;
    bool found;
    while (current != NULL) {
        found = false;
        for (int i = 0; i < length; i++) {
            if (counts_by_uni[i].uniname == current->saved_uniname) {
                counts_by_uni[i].count++;
                found = true;
                break;
            }
        }
        if (!found) {
            length++;
            counts_by_uni[length-1].uniname = current->saved_uniname;
            counts_by_uni[length-1].count = 1;
        }
        current = current->next;
    }

    // cout << "Before sorting (for debugging purposes): " << endl;
    // for (int i=0; i<length; i++){
    //     cout << i + 1 << ". " << counts_by_uni[i].uniname << ", " << counts_by_uni[i].count << endl;
    // }

    for (int i=0; i<length; i++){
        preuniq->push(counts_by_uni[i].uniname, counts_by_uni[i].count);
    }
    return preuniq;
}
```

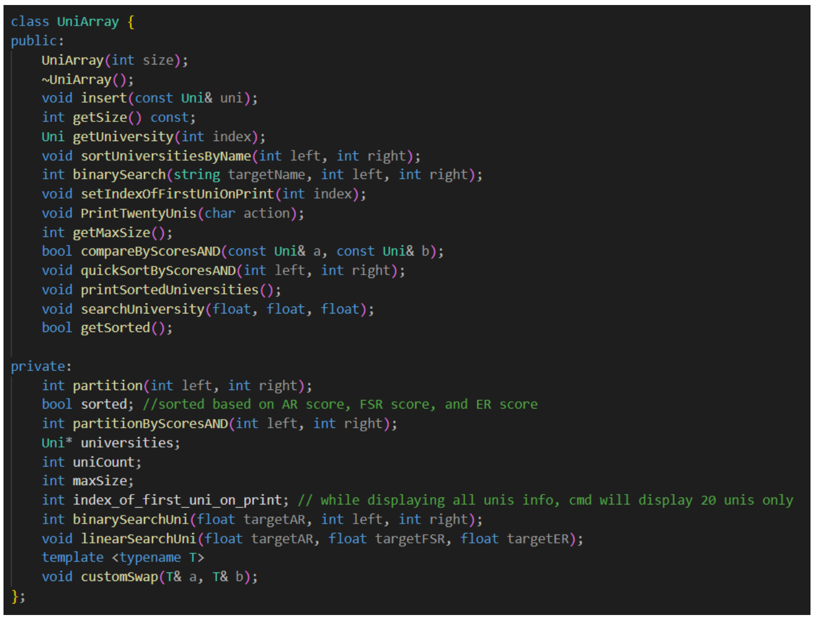
Figure 4.7.6: countNumOfSavedByUni() method

countNumOfSavedByUni() method is to calculate the numbers of saved of each university, and all the numbers are stored in the array of structures, *counts_by_uni*. The *counts_by_uni* array is first initialized to be of length 1422 since the CSV file has only 1422 universities.

A *while* loop is used to traverse the *SavedUniList* doubly linked list from front to end to count the numbers and update the numbers in the array. Inside the *while* loop, the array is traversed from front to end to see if there is a record inside. If there is, then update the number of the record; Otherwise, add a new record at the position length-1 of the array and set its number as 1.

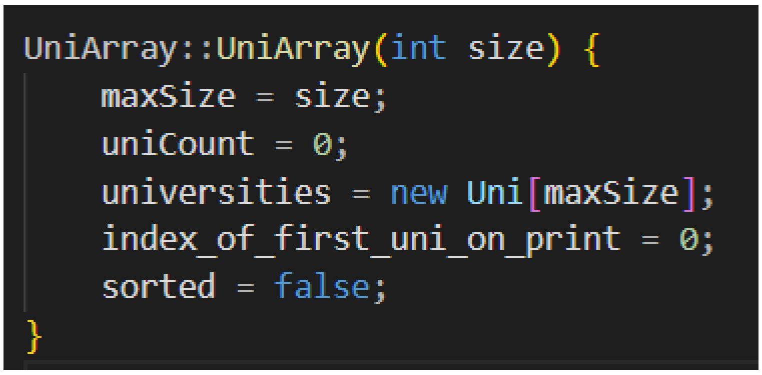
Then, all the records in the array will be pushed into the priority queue *PreferredUniQueue*, so that admins can see a list of most preferred universities in descending order based on the number of saves.

4.8 UniArray class



```
class UniArray {
public:
    UniArray(int size);
    ~UniArray();
    void insert(const Uni& uni);
    int getSize() const;
    Uni getUniversity(int index);
    void sortUniversitiesByName(int left, int right);
    int binarySearch(string targetName, int left, int right);
    void setIndexOfFirstUniOnPrint(int index);
    void PrintTwentyUnis(char action);
    int getMaxSize();
    bool compareByScoresAND(const Uni& a, const Uni& b);
    void quickSortByScoresAND(int left, int right);
    void printSortedUniversities();
    void searchUniversity(float, float, float);
    bool getSorted();

private:
    int partition(int left, int right);
    bool sorted; //sorted based on AR score, FSR score, and ER score
    int partitionByScoresAND(int left, int right);
    Uni* universities;
    int uniCount;
    int maxSize;
    int index_of_first_uni_on_print; // while displaying all unis info, cmd will display 20 unis only
    int binarySearchUni(float targetAR, int left, int right);
    void linearSearchUni(float targetAR, float targetFSR, float targetER);
    template <typename T>
    void customSwap(T& a, T& b);
};
```

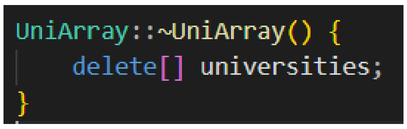
Figure 4.8.1.1: 'UniArray' class



```
UniArray::UniArray(int size) {
    maxSize = size;
    uniCount = 0;
    universities = new Uni[maxSize];
    index_of_first_uni_on_print = 0;
    sorted = false;
}
```

Figure 4.8.1.2: 'UniArray' method

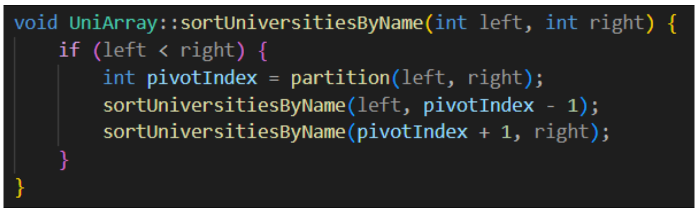
In this case, we have the *UniArray(int size)* builder. This clever bit of programming generates a dynamic, malleable array. So, because of the flexibility in data size, it demonstrates intelligence. This exemplifies efficient dynamic memory management in action.



```
UniArray::~UniArray() {
    delete[] universities;
}
```

Figure 4.8.1.3: 'UniArray' method

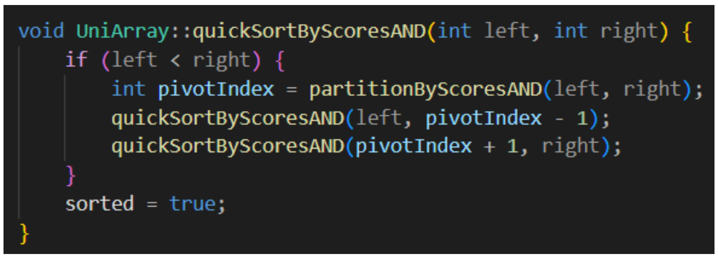
The *UniArray()* destructor is in charge of doing the last sweep. As soon as a 'UniArray' object is no longer needed, it efficiently handles memory allocation. It puts an end to memory leaks, so your system continues running smoothly. A superb comprehension of the lifetime of dynamic memory is also shown.



```
void UniArray::sortUniversitiesByName(int left, int right) {
    if (left < right) {
        int pivotIndex = partition(left, right);
        sortUniversitiesByName(left, pivotIndex - 1);
        sortUniversitiesByName(pivotIndex + 1, right);
    }
}
```

Figure 4.8.1.4: ‘UniArray’ method

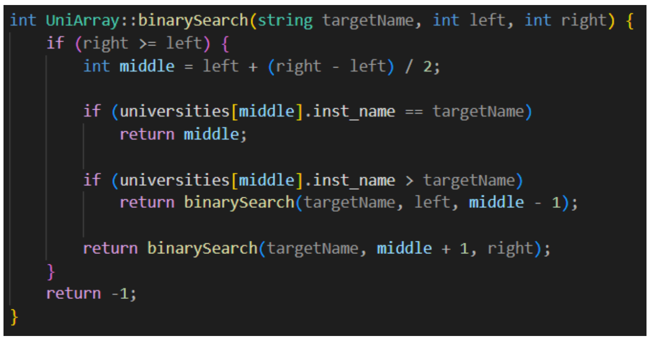
Now, we move on to the *sortUniversitiesByName()* procedure. The college library collection data is now alphabetized and easy to find thanks to this useful application. Helpful for getting about, it also lays the groundwork for effective searches. The arranged list makes it simple for visitors to locate the desired educational institution. It is also necessary for the operation of some search algorithms, such as binary search.



```
void UniArray::quickSortByScoresAND(int left, int right) {
    if (left < right) {
        int pivotIndex = partitionByScoresAND(left, right);
        quickSortByScoresAND(left, pivotIndex - 1);
        quickSortByScoresAND(pivotIndex + 1, right);
    }
    sorted = true;
}
```

Figure 4.8.1.5: ‘UniArray’ method

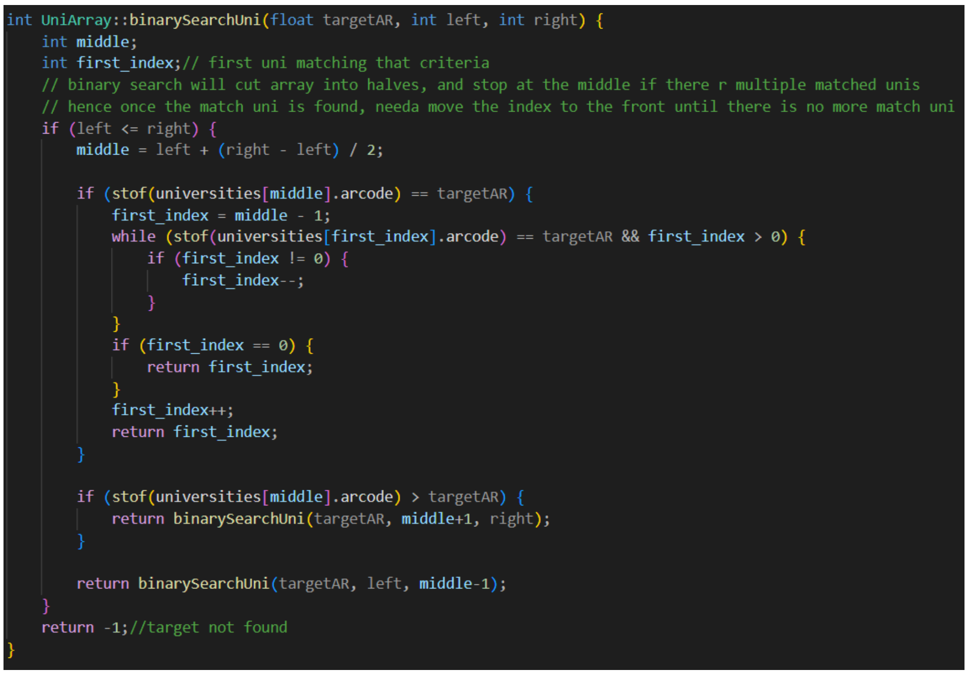
So, now on *quickSortByScoresAND()* is the heavy hitter of sorting functions; it employs the quicksort algorithm to rank schools according to their students' test results. Since it works well with enormous amounts of data, it has gained widespread popularity. Useful for comparing schools and highlighting exceptional students.



```
int UniArray::binarySearch(string targetName, int left, int right) {
    if (right >= left) {
        int middle = left + (right - left) / 2;

        if (universities[middle].inst_name == targetName)
            return middle;

        if (universities[middle].inst_name > targetName)
            return binarySearch(targetName, left, middle - 1);

        return binarySearch(targetName, middle + 1, right);
    }
    return -1;
}
```

Figure 4.8.1.6: ‘UniArray’ method

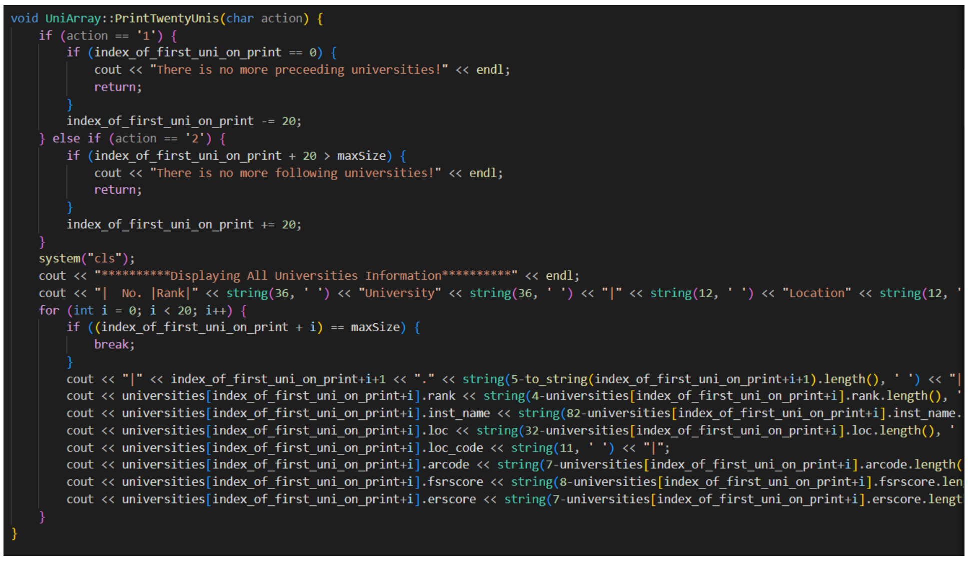
The *binarySearch()* method, which does a binary search to locate objects in a sorted list, is the most effective detective of the bunch. The speed is excellent, and it improves as more data is added. This lightning-fast data retrieval mechanism maintains the system agile as data volumes increase.



```cpp
int UniArray::binarySearchUni(float targetAR, int left, int right) {
    int middle;
    int first_index;// first uni matching that criteria
    // binary search will cut array into halves, and stop at the middle if there r multiple matched unis
    // hence once the match uni is found, needa move the index to the front until there is no more match uni
    if (left <= right) {
        middle = left + (right - left) / 2;

        if (stof(universities[middle].arcode) == targetAR) {
            first_index = middle - 1;
            while (stof(universities[first_index].arcode) == targetAR && first_index > 0) {
                if (first_index != 0) {
                    first_index--;
                }
            }
            if (first_index == 0) {
                return first_index;
            }
            first_index++;
            return first_index;
        }

        if (stof(universities[middle].arcode) > targetAR) {
            return binarySearchUni(targetAR, middle+1, right);
        }

        return binarySearchUni(targetAR, left, middle-1);
    }
    return -1;//target not found
}
```

Figure 4.8.1.7: ‘UniArray’ method

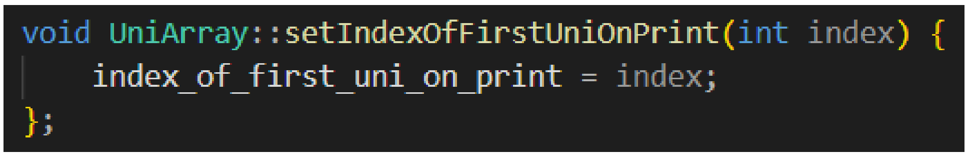
A little variation on *binarySearch()*, *binarySearchUni()* searches for a given university inside the ordered set. The traditional algorithm of binary search will only return one matching university, but in this system there could be multiple universities matching the target score. Hence, *first_index* is utilized here to keep the reference to first matching university out of multiple matching universities. Traditional binary search will stop at the middle of multiple matching universities, a *while* loop is used to see if the previous university matches the target score. If the previous university matches the target score, decrement the *first_index*; Else, just return *first_index*.



```cpp
void UniArray::PrintTwentyUnis(char action) {
    if (action == '1') {
        if (index_of_first_uni_on_print == 0) {
            cout << "There is no more preceeding universities!" << endl;
            return;
        }
        index_of_first_uni_on_print -= 20;
    } else if (action == '2') {
        if (index_of_first_uni_on_print + 20 > maxSize) {
            cout << "There is no more following universities!" << endl;
            return;
        }
        index_of_first_uni_on_print += 20;
    }
    system("cls");
    cout << "**********Displaying All Universities Information**********" << endl;
    cout << "|  No. |Rank|" << string(36, ' ') << "University" << string(36, ' ') << "|" << string(12, ' ') << "Location" << string(12, '
    for (int i = 0; i < 20; i++) {
        if ((index_of_first_uni_on_print + i) == maxSize) {
            break;
        }
        cout << "|" << index_of_first_uni_on_print+i+1 << "." << string(5-to_string(index_of_first_uni_on_print+i+1).length(), ' ') << "|
        cout << universities[index_of_first_uni_on_print+i].rank << string(4-universities[index_of_first_uni_on_print+i].rank.length(), '
        cout << universities[index_of_first_uni_on_print+i].inst_name << string(82-universities[index_of_first_uni_on_print+i].inst_name.
        cout << universities[index_of_first_uni_on_print+i].loc << string(32-universities[index_of_first_uni_on_print+i].loc.length(), '
        cout << universities[index_of_first_uni_on_print+i].loc_code << string(11, ' ') << "|";
        cout << universities[index_of_first_uni_on_print+i].arcode << string(7-universities[index_of_first_uni_on_print+i].arcode.length(
        cout << universities[index_of_first_uni_on_print+i].fsrscore << string(8-universities[index_of_first_uni_on_print+i].fsrscore.len
        cout << universities[index_of_first_uni_on_print+i].erscore << string(7-universities[index_of_first_uni_on_print+i].erscore.lengt
    }
}
```

Figure 4.8.1.8: ‘UniArray’ method

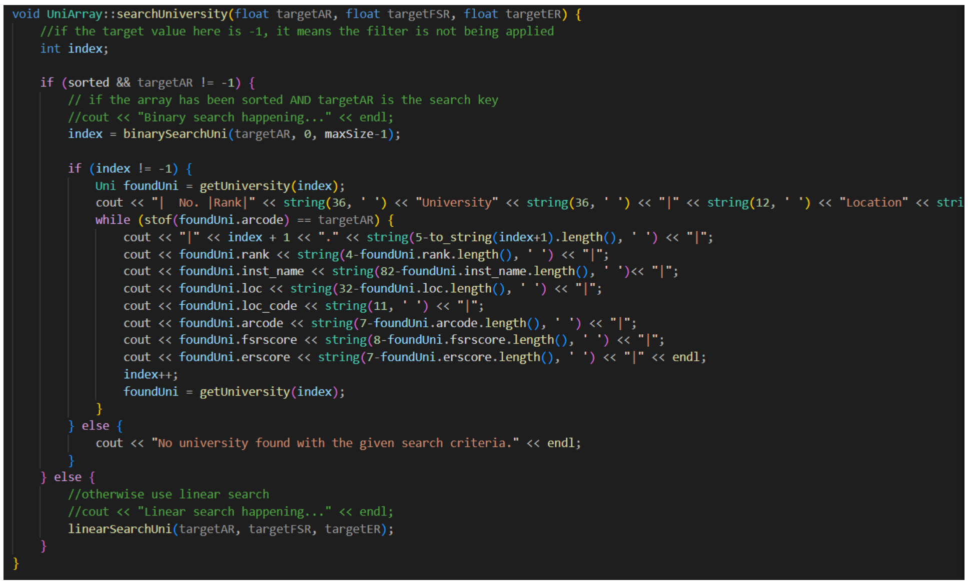
When a user requests a list of universities, our *PrintTwentyUnis(char action)* method will display twenty of them. If we are using a console application, the data will be presented in chunks that are simpler to absorb. Designing for ease of use is paramount.



```
void UniArray::setIndexOfFirstUniOnPrint(int index) {
    index_of_first_uni_on_print = index;
};
```

Figure 4.8.1.9: 'UniArray' method

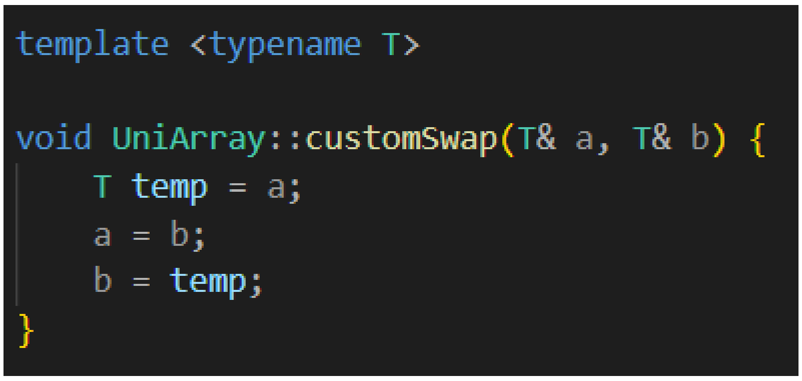
With the help of the *setIndexOfFirstUniOnPrint(int index)* method, the user's input is put front and center. When presenting institutions, users may choose a specific beginning point. So, visitors may see their preferred content at their convenience. Users will have a better experience.



```
void UniArray::searchUniversity(float targetAR, float targetFSR, float targetER) {
    //if the target value here is -1, it means the filter is not being applied
    int index;

    if (sorted && targetAR != -1) {
        // if the array has been sorted AND targetAR is the search key
        //cout << "Binary search happening..." << endl;
        index = binarySearchUni(targetAR, 0, maxSize-1);

        if (index != -1) {
            Uni foundUni = getUniversity(index);
            cout << "|  No. |Rank|" << string(36, ' ') << "University" << string(36, ' ') << "|" << string(12, ' ') << "Location" << stri
            while (stof(foundUni.arcode) == targetAR) {
                cout << "|" << index + 1 << "." << string(5-to_string(index+1).length(), ' ') << "|";
                cout << foundUni.rank << string(4-foundUni.rank.length(), ' ') << "|";
                cout << foundUni.inst_name << string(82-foundUni.inst_name.length(), ' ')<< "|";
                cout << foundUni.loc << string(32-foundUni.loc.length(), ' ') << "|";
                cout << foundUni.loc_code << string(11, ' ') << "|";
                cout << foundUni.arcode << string(7-foundUni.arcode.length(), ' ') << "|";
                cout << foundUni.fsrscore << string(8-foundUni.fsrscore.length(), ' ') << "|";
                cout << foundUni.erscore << string(7-foundUni.erscore.length(), ' ') << "|" << endl;
                index++;
                foundUni = getUniversity(index);
            }
        } else {
            cout << "No university found with the given search criteria." << endl;
        }
    } else {
        //otherwise use linear search
        //cout << "Linear search happening..." << endl;
        linearSearchUni(targetAR, targetFSR, targetER);
    }
}
```

Figure 4.8.2.0: 'UniArray' method

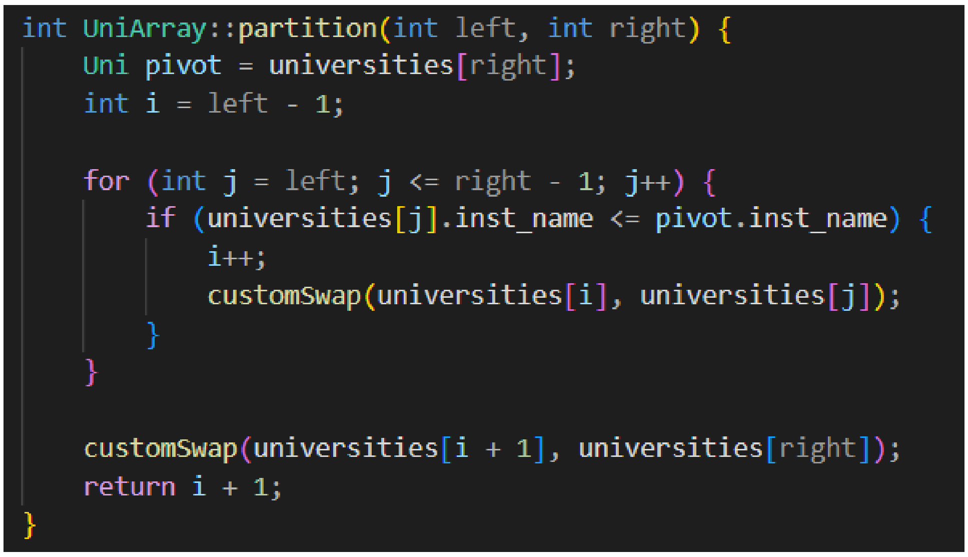
So, now on to the *searchUniversity()* method. It is like having a Swiss Army knife for searching. If the array is sorted, it uses that information to choose which search technique to use. It can easily adjust to new environments and continue performing at a high level. The customers are allowed to search for universities by applying any one filter based on AR score, FSR score, or ER score. If the customer searches for universities that match the target filter score, then the other target filter score will be set to -1. If the universities had been sorted based on AR scores and the applied filter is AR score, then binary search will be performed to ensure quick retrieval; Else linear search will be performed.



```cpp
template <typename T>

void UniArray::customSwap(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}
```

Figure 4.8.2.1: 'UniArray' method

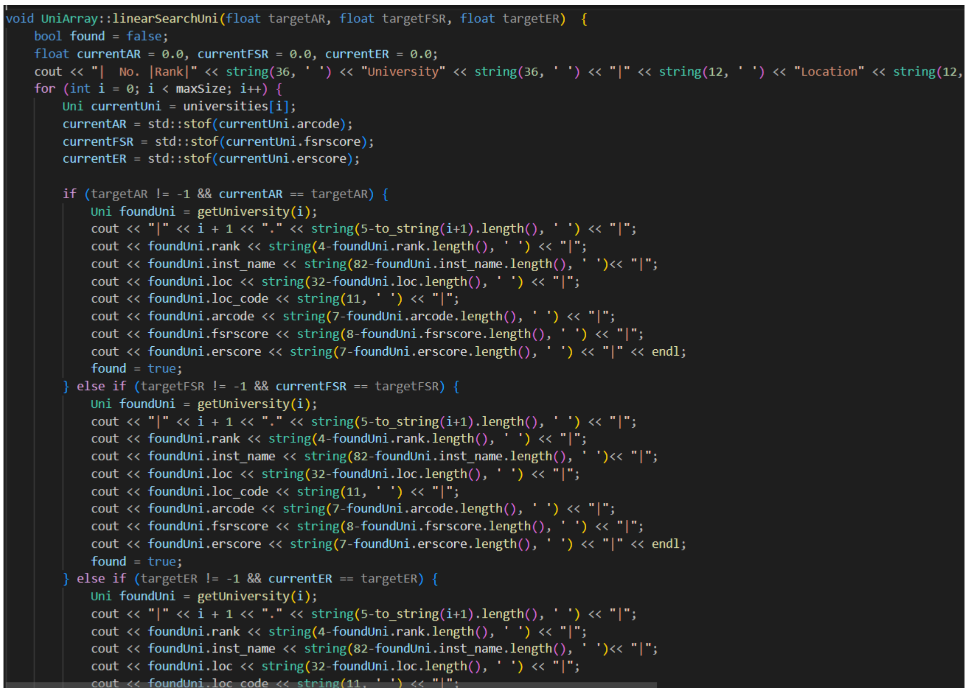
The *customSwap(T& a, T& b)* function is a handy shorthand for swapping the positions of two integer values, and it does it quickly and efficiently. As a crucial part of sorting algorithms, its independent use helps maintain clean, modular code. The template is used to maintain the same data type for a more consistent and efficient code or performance for this system.



```cpp
int UniArray::partition(int left, int right) {
    Uni pivot = universities[right];
    int i = left - 1;

    for (int j = left; j <= right - 1; j++) {
        if (universities[j].inst_name <= pivot.inst_name) {
            i++;
            customSwap(universities[i], universities[j]);
        }
    }

    customSwap(universities[i + 1], universities[right]);
    return i + 1;
}
```

Figure 4.8.2.2: 'UniArray' method

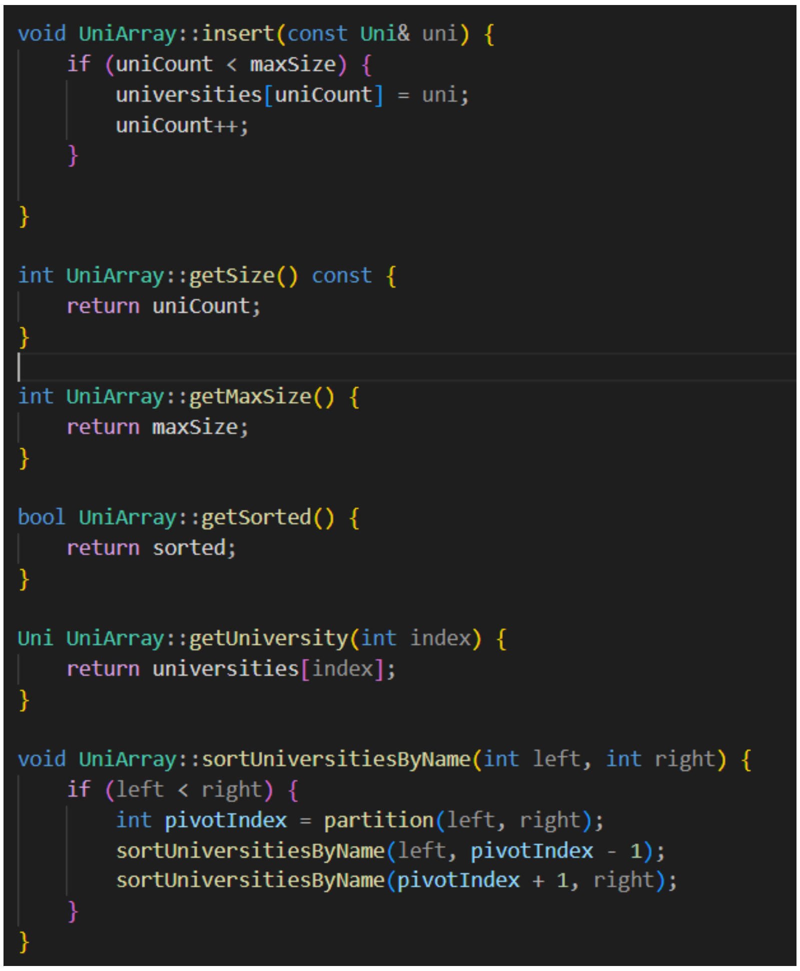
QuickSort is really calculated using the *partition(int left, int right)* function. To quickly sort data, the array is split in two around a central element. This basic capability has the potential to completely alter the game.



```cpp
void UniArray::linearSearchUni(float targetAR, float targetFSR, float targetER)  {
    bool found = false;
    float currentAR = 0.0, currentFSR = 0.0, currentER = 0.0;
    cout << "|  No. |Rank|" << string(36, ' ') << "University" << string(36, ' ') << "|" << string(12, ' ') << "Location" << string(12,
    for (int i = 0; i < maxSize; i++) {
        Uni currentUni = universities[i];
        currentAR = std::stof(currentUni.arcode);
        currentFSR = std::stof(currentUni.fsrscore);
        currentER = std::stof(currentUni.erscore);

        if (targetAR != -1 && currentAR == targetAR) {
            Uni foundUni = getUniversity(i);
            cout << "|" << i + 1 << "." << string(5-to_string(i+1).length(), ' ') << "|";
            cout << foundUni.rank << string(4-foundUni.rank.length(), ' ') << "|";
            cout << foundUni.inst_name << string(82-foundUni.inst_name.length(), ' ')<< "|";
            cout << foundUni.loc << string(32-foundUni.loc.length(), ' ') << "|";
            cout << foundUni.loc_code << string(11, ' ') << "|";
            cout << foundUni.arcode << string(7-foundUni.arcode.length(), ' ') << "|";
            cout << foundUni.fsrscore << string(8-foundUni.fsrscore.length(), ' ') << "|";
            cout << foundUni.erscore << string(7-foundUni.erscore.length(), ' ') << "|" << endl;
            found = true;
        } else if (targetFSR != -1 && currentFSR == targetFSR) {
            Uni foundUni = getUniversity(i);
            cout << "|" << i + 1 << "." << string(5-to_string(i+1).length(), ' ') << "|";
            cout << foundUni.rank << string(4-foundUni.rank.length(), ' ') << "|";
            cout << foundUni.inst_name << string(82-foundUni.inst_name.length(), ' ')<< "|";
            cout << foundUni.loc << string(32-foundUni.loc.length(), ' ') << "|";
            cout << foundUni.loc_code << string(11, ' ') << "|";
            cout << foundUni.arcode << string(7-foundUni.arcode.length(), ' ') << "|";
            cout << foundUni.fsrscore << string(8-foundUni.fsrscore.length(), ' ') << "|";
            cout << foundUni.erscore << string(7-foundUni.erscore.length(), ' ') << "|" << endl;
            found = true;
        } else if (targetER != -1 && currentER == targetER) {
            Uni foundUni = getUniversity(i);
            cout << "|" << i + 1 << "." << string(5-to_string(i+1).length(), ' ') << "|";
            cout << foundUni.rank << string(4-foundUni.rank.length(), ' ') << "|";
            cout << foundUni.inst_name << string(82-foundUni.inst_name.length(), ' ')<< "|";
            cout << foundUni.loc << string(32-foundUni.loc.length(), ' ') << "|";
            cout << foundUni.loc_code << string(11, ' ') << "|";
```

Figure 4.8.2.3: 'UniArray' method

The QuickSort algorithm may be run by using the *linearSearchUni(float targetAR, float targetFSR, float targetER)* function. It is a lightning-fast machine built for maximum productivity in a number of contexts. Prove our proficiency with complex sorting procedures by using this approach.



```
void UniArray::insert(const Uni& uni) {
    if (uniCount < maxSize) {
        universities[uniCount] = uni;
        uniCount++;
    }

}

int UniArray::getSize() const {
    return uniCount;
}

int UniArray::getMaxSize() {
    return maxSize;
}

bool UniArray::getSorted() {
    return sorted;
}

Uni UniArray::getUniversity(int index) {
    return universities[index];
}

void UniArray::sortUniversitiesByName(int left, int right) {
    if (left < right) {
        int pivotIndex = partition(left, right);
        sortUniversitiesByName(left, pivotIndex - 1);
        sortUniversitiesByName(pivotIndex + 1, right);
    }
}
```

Figure 4.8.2.4: 'UniArray' methods

The library's *sortUniversitiesByName(int left, int right)* method acts as its librarian. Finding a school by name demonstrates the system's prowess with strings of text. For information retrieval to be straightforward, this feature is essential.

In this approach, *getUniversity(int index)* serves as the statistician. It monitors the growth of the college mailing list, providing valuable insight into the health of the data infrastructure. It is an important instrument for fixing problems and adjusting to new circumstances.

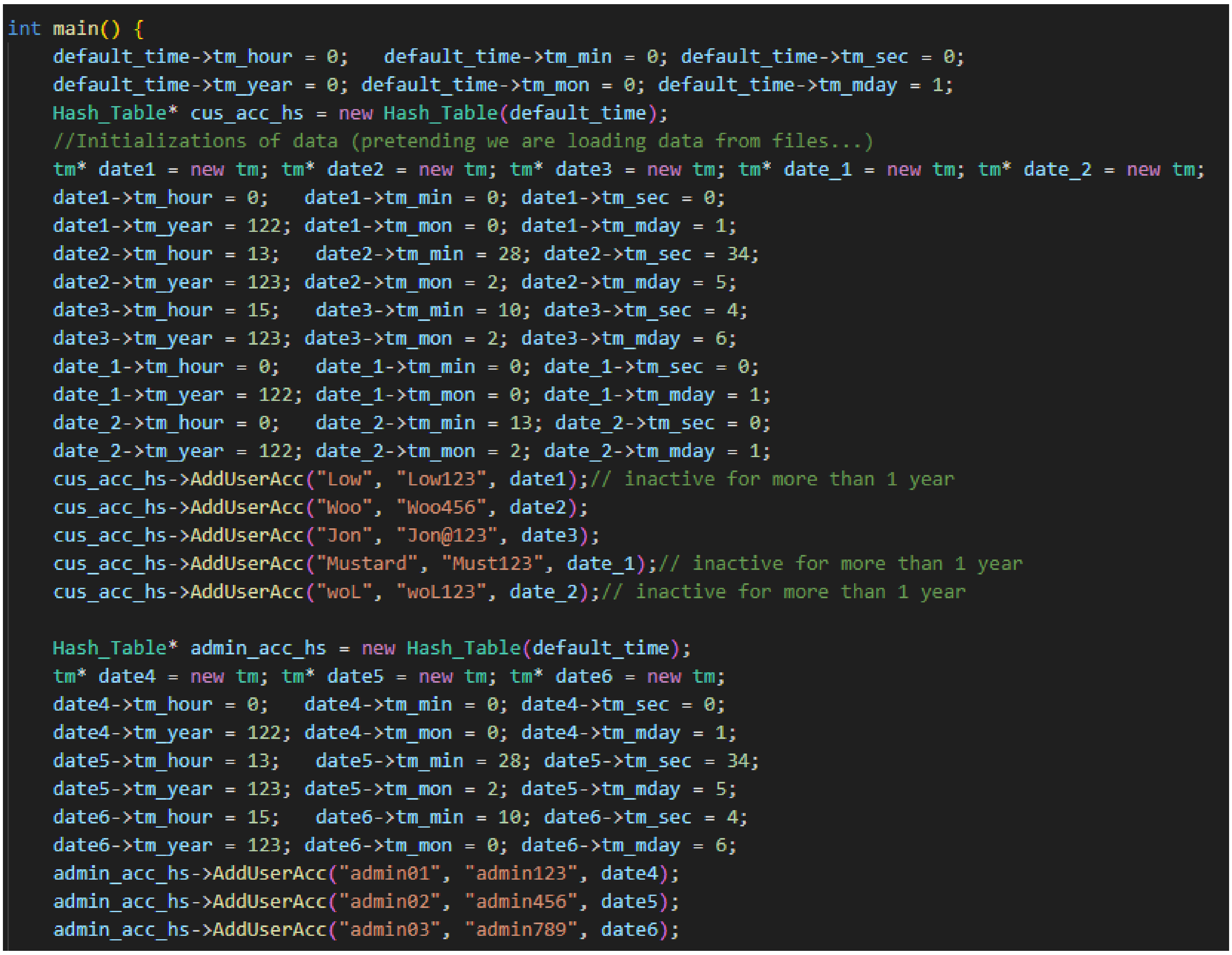
As an integral part of arrays, the *getUniversity(int index)* method is crucial. In order to maintain the system's essential operation, it fetches a university at a certain index.

The system itself is speaking via the *PrintTwentyUnis(char action)* function. Thus, users can comprehend information. A data processing system is incomplete without it.

To summarize, all the 'UniArray' class's methods are important. Important topics including data storage, organization and user cooperation are all addressed. We have shown our C++ expertise by creating a system that is sophisticated but straightforward and easy to navigate.

4.9 main.cpp

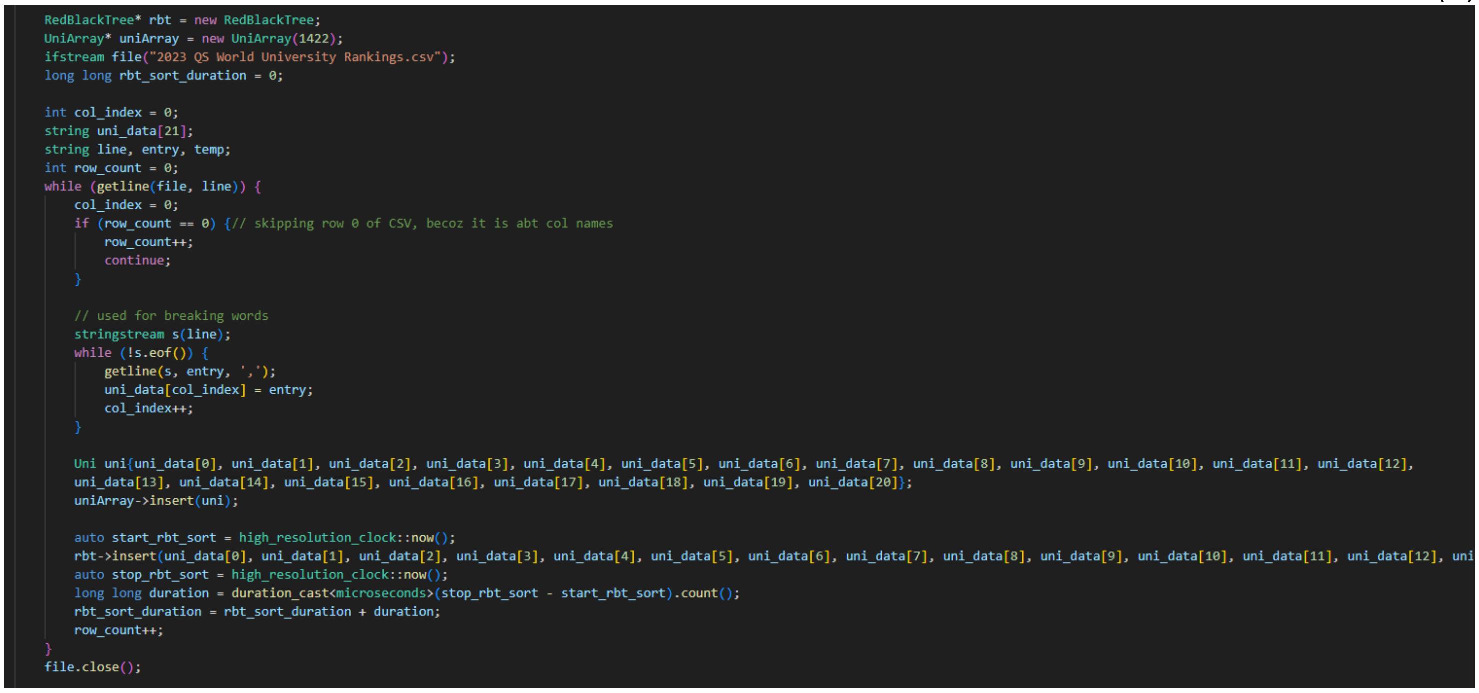
Inside **main.cpp**, this C++ file keeps functions of interfaces for normal users.



```cpp
int main() {
    default_time->tm_hour = 0;   default_time->tm_min = 0; default_time->tm_sec = 0;
    default_time->tm_year = 0; default_time->tm_mon = 0; default_time->tm_mday = 1;
    Hash_Table* cus_acc_hs = new Hash_Table(default_time);
    //Initializations of data (pretending we are loading data from files...)
    tm* date1 = new tm; tm* date2 = new tm; tm* date3 = new tm; tm* date_1 = new tm; tm* date_2 = new tm;
    date1->tm_hour = 0;   date1->tm_min = 0; date1->tm_sec = 0;
    date1->tm_year = 122; date1->tm_mon = 0; date1->tm_mday = 1;
    date2->tm_hour = 13;   date2->tm_min = 28; date2->tm_sec = 34;
    date2->tm_year = 123; date2->tm_mon = 2; date2->tm_mday = 5;
    date3->tm_hour = 15;   date3->tm_min = 10; date3->tm_sec = 4;
    date3->tm_year = 123; date3->tm_mon = 2; date3->tm_mday = 6;
    date_1->tm_hour = 0;   date_1->tm_min = 0; date_1->tm_sec = 0;
    date_1->tm_year = 122; date_1->tm_mon = 0; date_1->tm_mday = 1;
    date_2->tm_hour = 0;   date_2->tm_min = 13; date_2->tm_sec = 0;
    date_2->tm_year = 122; date_2->tm_mon = 2; date_2->tm_mday = 1;
    cus_acc_hs->AddUserAcc("Low", "Low123", date1);// inactive for more than 1 year
    cus_acc_hs->AddUserAcc("Woo", "Woo456", date2);
    cus_acc_hs->AddUserAcc("Jon", "Jon@123", date3);
    cus_acc_hs->AddUserAcc("Mustard", "Must123", date_1);// inactive for more than 1 year
    cus_acc_hs->AddUserAcc("woL", "woL123", date_2);// inactive for more than 1 year

    Hash_Table* admin_acc_hs = new Hash_Table(default_time);
    tm* date4 = new tm; tm* date5 = new tm; tm* date6 = new tm;
    date4->tm_hour = 0;   date4->tm_min = 0; date4->tm_sec = 0;
    date4->tm_year = 122; date4->tm_mon = 0; date4->tm_mday = 1;
    date5->tm_hour = 13;   date5->tm_min = 28; date5->tm_sec = 34;
    date5->tm_year = 123; date5->tm_mon = 2; date5->tm_mday = 5;
    date6->tm_hour = 15;   date6->tm_min = 10; date6->tm_sec = 4;
    date6->tm_year = 123; date6->tm_mon = 0; date6->tm_mday = 6;
    admin_acc_hs->AddUserAcc("admin01", "admin123", date4);
    admin_acc_hs->AddUserAcc("admin02", "admin456", date5);
    admin_acc_hs->AddUserAcc("admin03", "admin789", date6);
```

Figure 4.9.1: Initializations of Hash Tables in main()

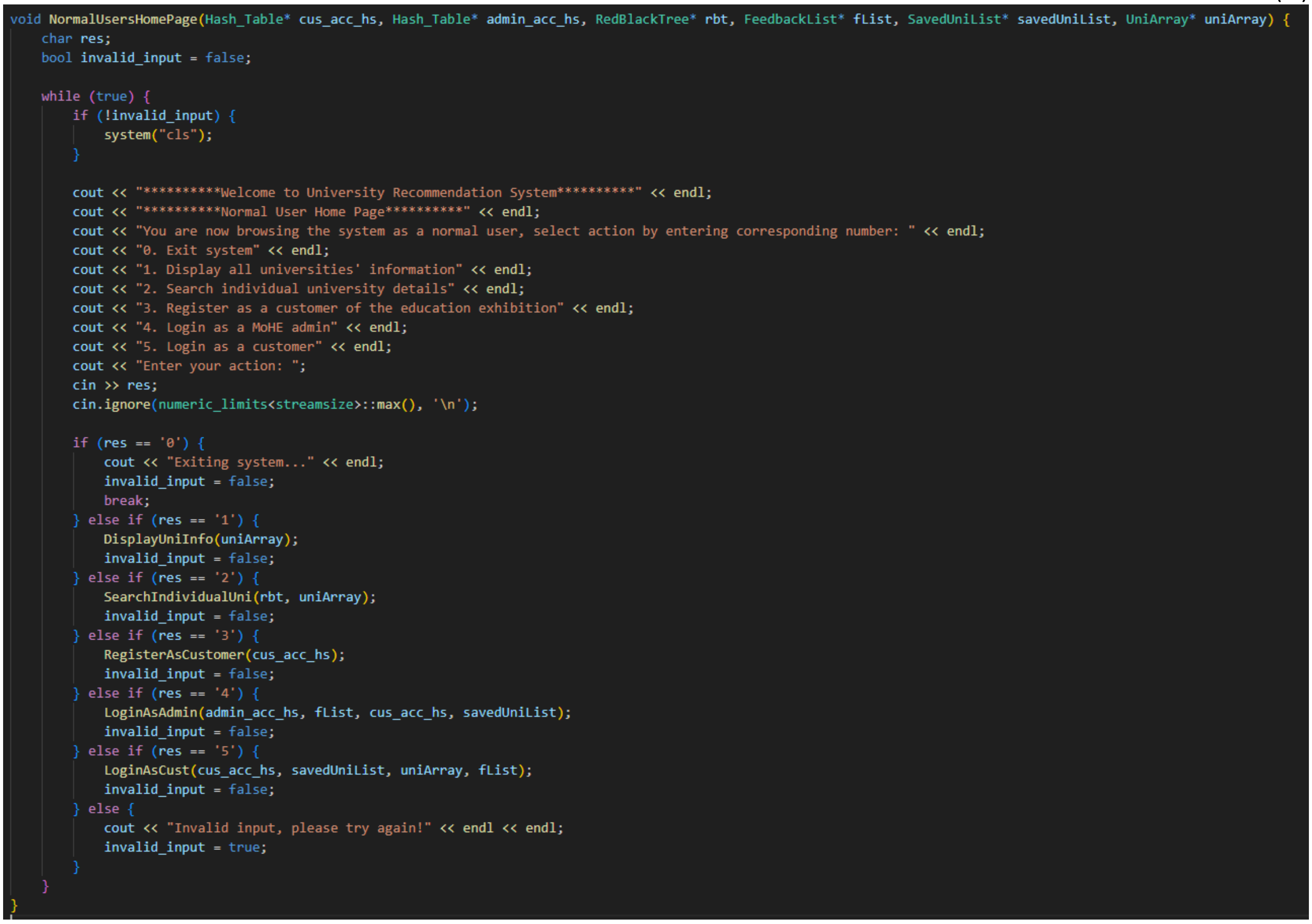
In the figure above, the main function in **main.cpp** will initialize the hash tables with some customers' and admins' credentials. Some of the last active times of the customers are controlled so that they are inactive for more than 1 year.



```
RedBlackTree* rbt = new RedBlackTree;
UniArray* uniArray = new UniArray(1422);
ifstream file("2023 QS World University Rankings.csv");
long long rbt_sort_duration = 0;

int col_index = 0;
string uni_data[21];
string line, entry, temp;
int row_count = 0;
while (getline(file, line)) {
    col_index = 0;
    if (row_count == 0) {// skipping row 0 of CSV, becoz it is abt col names
        row_count++;
        continue;
    }

    // used for breaking words
    stringstream s(line);
    while (!s.eof()) {
        getline(s, entry, ',');
        uni_data[col_index] = entry;
        col_index++;
    }

    Uni uni{uni_data[0], uni_data[1], uni_data[2], uni_data[3], uni_data[4], uni_data[5], uni_data[6], uni_data[7], uni_data[8], uni_data[9], uni_data[10], uni_data[11], uni_data[12],
    uni_data[13], uni_data[14], uni_data[15], uni_data[16], uni_data[17], uni_data[18], uni_data[19], uni_data[20]};
    uniArray->insert(uni);

    auto start_rbt_sort = high_resolution_clock::now();
    rbt->insert(uni_data[0], uni_data[1], uni_data[2], uni_data[3], uni_data[4], uni_data[5], uni_data[6], uni_data[7], uni_data[8], uni_data[9], uni_data[10], uni_data[11], uni_data[12], uni
    auto stop_rbt_sort = high_resolution_clock::now();
    long long duration = duration_cast<microseconds>(stop_rbt_sort - start_rbt_sort).count();
    rbt_sort_duration = rbt_sort_duration + duration;
    row_count++;
}
file.close();
```

Figure 4.9.2: Reading CSV file in main()

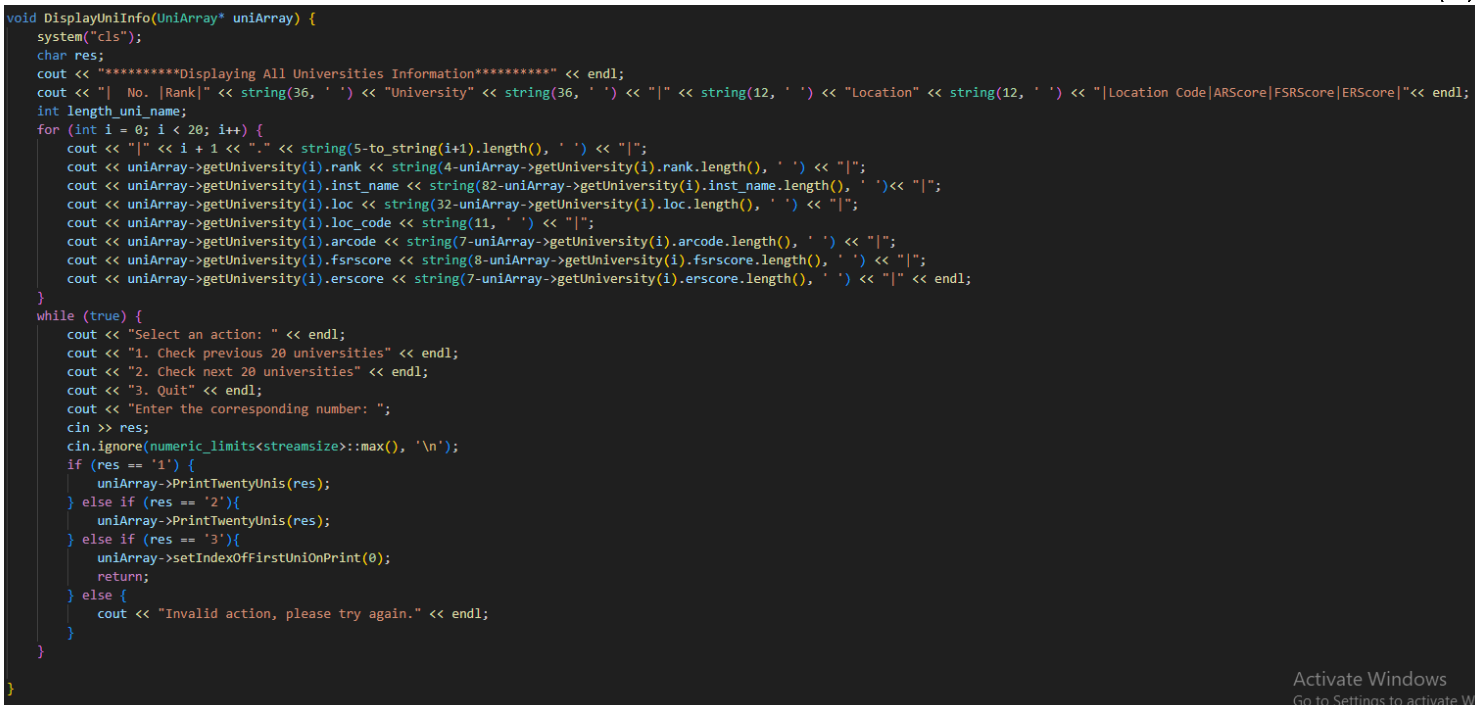
Figure 4.9.2 shows that *main()* function is also trying to load all the universities from the CSV file to initialize the red black tree and array of structures. This is for comparing the execution times for searching and sorting algorithms. There are still more initializations inside the *main()* function, they are inside the submitted zip file.



```cpp
void NormalUsersHomePage(Hash_Table* cus_acc_hs, Hash_Table* admin_acc_hs, RedBlackTree* rbt, FeedbackList* fList, SavedUniList* savedUniList, UniArray* uniArray) {
    char res;
    bool invalid_input = false;

    while (true) {
        if (!invalid_input) {
            system("cls");
        }

        cout << "**********Welcome to University Recommendation System**********" << endl;
        cout << "**********Normal User Home Page**********" << endl;
        cout << "You are now browsing the system as a normal user, select action by entering corresponding number: " << endl;
        cout << "0. Exit system" << endl;
        cout << "1. Display all universities' information" << endl;
        cout << "2. Search individual university details" << endl;
        cout << "3. Register as a customer of the education exhibition" << endl;
        cout << "4. Login as a MoHE admin" << endl;
        cout << "5. Login as a customer" << endl;
        cout << "Enter your action: ";
        cin >> res;
        cin.ignore(numeric_limits<streamsize>::max(), '\n');

        if (res == '0') {
            cout << "Exiting system..." << endl;
            invalid_input = false;
            break;
        } else if (res == '1') {
            DisplayUniInfo(uniArray);
            invalid_input = false;
        } else if (res == '2') {
            SearchIndividualUni(rbt, uniArray);
            invalid_input = false;
        } else if (res == '3') {
            RegisterAsCustomer(cus_acc_hs);
            invalid_input = false;
        } else if (res == '4') {
            LoginAsAdmin(admin_acc_hs, fList, cus_acc_hs, savedUniList);
            invalid_input = false;
        } else if (res == '5') {
            LoginAsCust(cus_acc_hs, savedUniList, uniArray, fList);
            invalid_input = false;
        } else {
            cout << "Invalid input, please try again!" << endl << endl;
            invalid_input = true;
        }
    }
}
```

Figure 4.9.3: Home Page for Normal Users

Figure 4.9.3 shows the function for generating a home page for normal users. This function is just some simple **if else** logics which are not worth mentioning. Entered numbers will redirect the normal user to corresponding pages by calling corresponding functions.



```cpp
void DisplayUniInfo(UniArray* uniArray) {
    system("cls");
    char res;
    cout << "**********Displaying All Universities Information**********" << endl;
    cout << "|  No. |Rank|" << string(36, ' ') << "University" << string(36, ' ') << "|" << string(12, ' ') << "Location" << string(12, ' ') << "|Location Code|ARScore|FSRScore|ERScore|"<< endl;
    int length_uni_name;
    for (int i = 0; i < 20; i++) {
        cout << "|" << i + 1 << "." << string(5-to_string(i+1).length(), ' ') << "|";
        cout << uniArray->getUniversity(i).rank << string(4-uniArray->getUniversity(i).rank.length(), ' ') << "|";
        cout << uniArray->getUniversity(i).inst_name << string(82-uniArray->getUniversity(i).inst_name.length(), ' ')<< "|";
        cout << uniArray->getUniversity(i).loc << string(32-uniArray->getUniversity(i).loc.length(), ' ') << "|";
        cout << uniArray->getUniversity(i).loc_code << string(11, ' ') << "|";
        cout << uniArray->getUniversity(i).arcode << string(7-uniArray->getUniversity(i).arcode.length(), ' ') << "|";
        cout << uniArray->getUniversity(i).fsrscore << string(8-uniArray->getUniversity(i).fsrscore.length(), ' ') << "|";
        cout << uniArray->getUniversity(i).erscore << string(7-uniArray->getUniversity(i).erscore.length(), ' ') << "|" << endl;
    }
    while (true) {
        cout << "Select an action: " << endl;
        cout << "1. Check previous 20 universities" << endl;
        cout << "2. Check next 20 universities" << endl;
        cout << "3. Quit" << endl;
        cout << "Enter the corresponding number: ";
        cin >> res;
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
        if (res == '1') {
            uniArray->PrintTwentyUnis(res);
        } else if (res == '2'){
            uniArray->PrintTwentyUnis(res);
        } else if (res == '3'){
            uniArray->setIndexOfFirstUniOnPrint(0);
            return;
        } else {
            cout << "Invalid action, please try again." << endl;
        }
    }
}
```

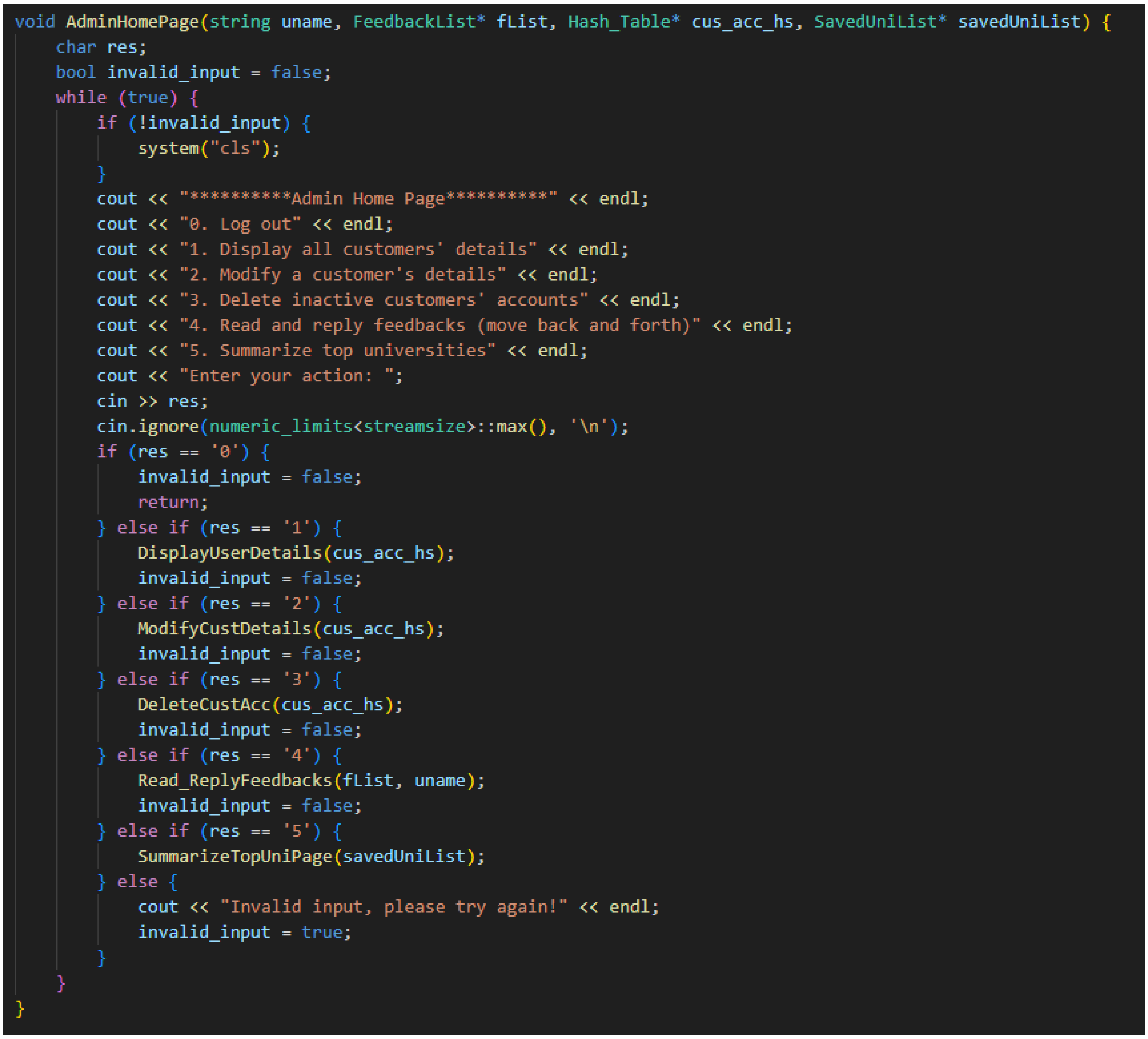
Figure 4.9.4: DisplayUniInfo() for Normal Users

Figure above shows the page allowing normal users to display all universities, which each page has a maximum of 20 universities. **UniArray->setIndexOfFirstOnPrint(0)** is executed before normal users exiting this page, this is to ensure every time the normal users are redirected to this page, this page will always show the first 20 universities by default.

For the other functions in **main.cpp**, they consist of some simple **while**, **if**, and **else** statements which are not worth explaining (due to the page number constraints), the core algorithms had been done in the classes of those data structure. For more information about those functions, please refer to the submitted zip file.

4.10 AdminPages.cpp

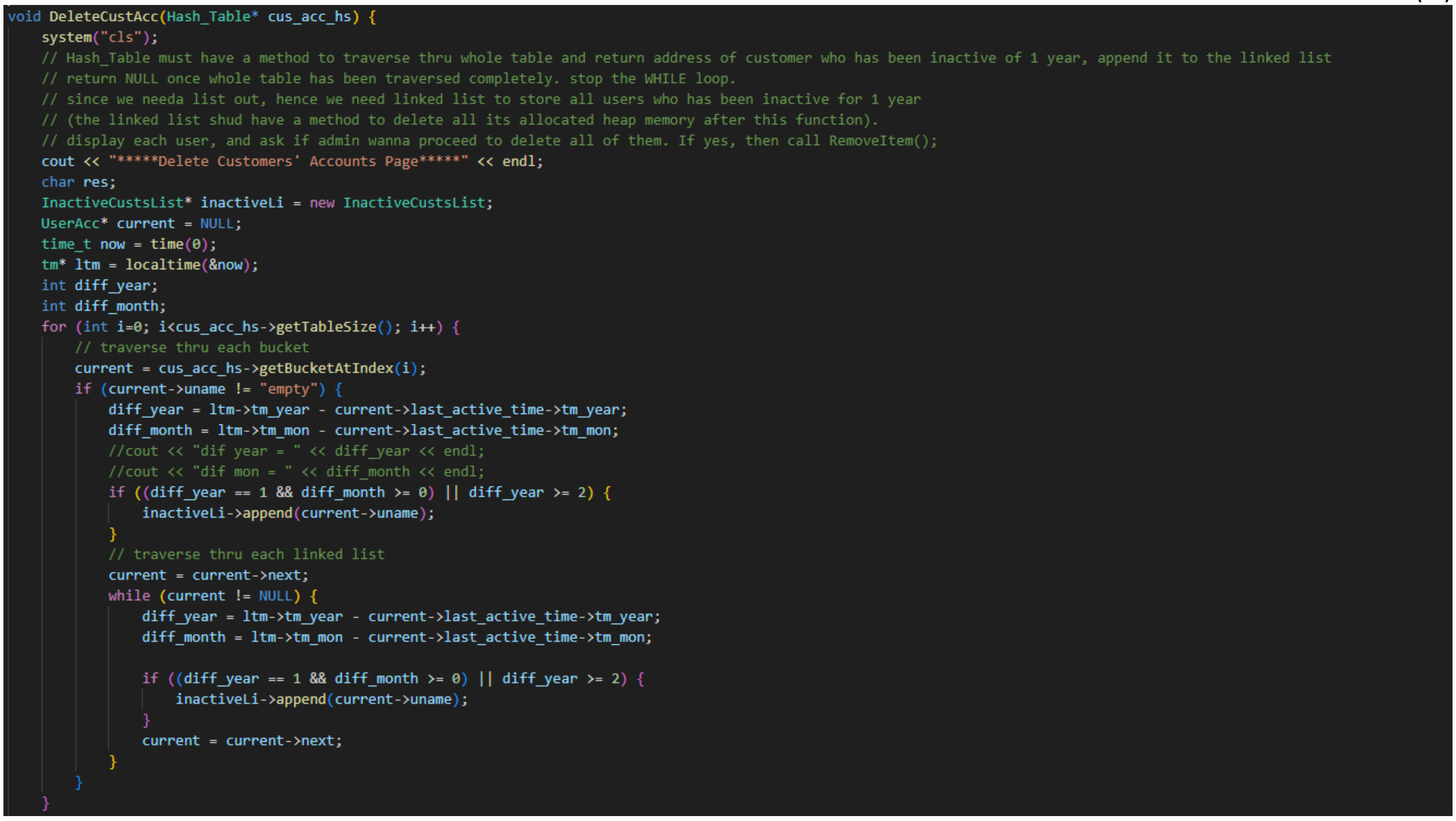
Inside **AdminPages.cpp**, this C++ file keeps functions of interfaces for admins.



```
void AdminHomePage(string uname, FeedbackList* fList, Hash_Table* cus_acc_hs, SavedUniList* savedUniList) {
    char res;
    bool invalid_input = false;
    while (true) {
        if (!invalid_input) {
            system("cls");
        }
        cout << "**********Admin Home Page**********" << endl;
        cout << "0. Log out" << endl;
        cout << "1. Display all customers' details" << endl;
        cout << "2. Modify a customer's details" << endl;
        cout << "3. Delete inactive customers' accounts" << endl;
        cout << "4. Read and reply feedbacks (move back and forth)" << endl;
        cout << "5. Summarize top universities" << endl;
        cout << "Enter your action: ";
        cin >> res;
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
        if (res == '0') {
            invalid_input = false;
            return;
        } else if (res == '1') {
            DisplayUserDetails(cus_acc_hs);
            invalid_input = false;
        } else if (res == '2') {
            ModifyCustDetails(cus_acc_hs);
            invalid_input = false;
        } else if (res == '3') {
            DeleteCustAcc(cus_acc_hs);
            invalid_input = false;
        } else if (res == '4') {
            Read_ReplyFeedbacks(fList, uname);
            invalid_input = false;
        } else if (res == '5') {
            SummarizeTopUniPage(savedUniList);
        } else {
            cout << "Invalid input, please try again!" << endl;
            invalid_input = true;
        }
    }
}
```

Figure 4.10.1: Home Page for Admins

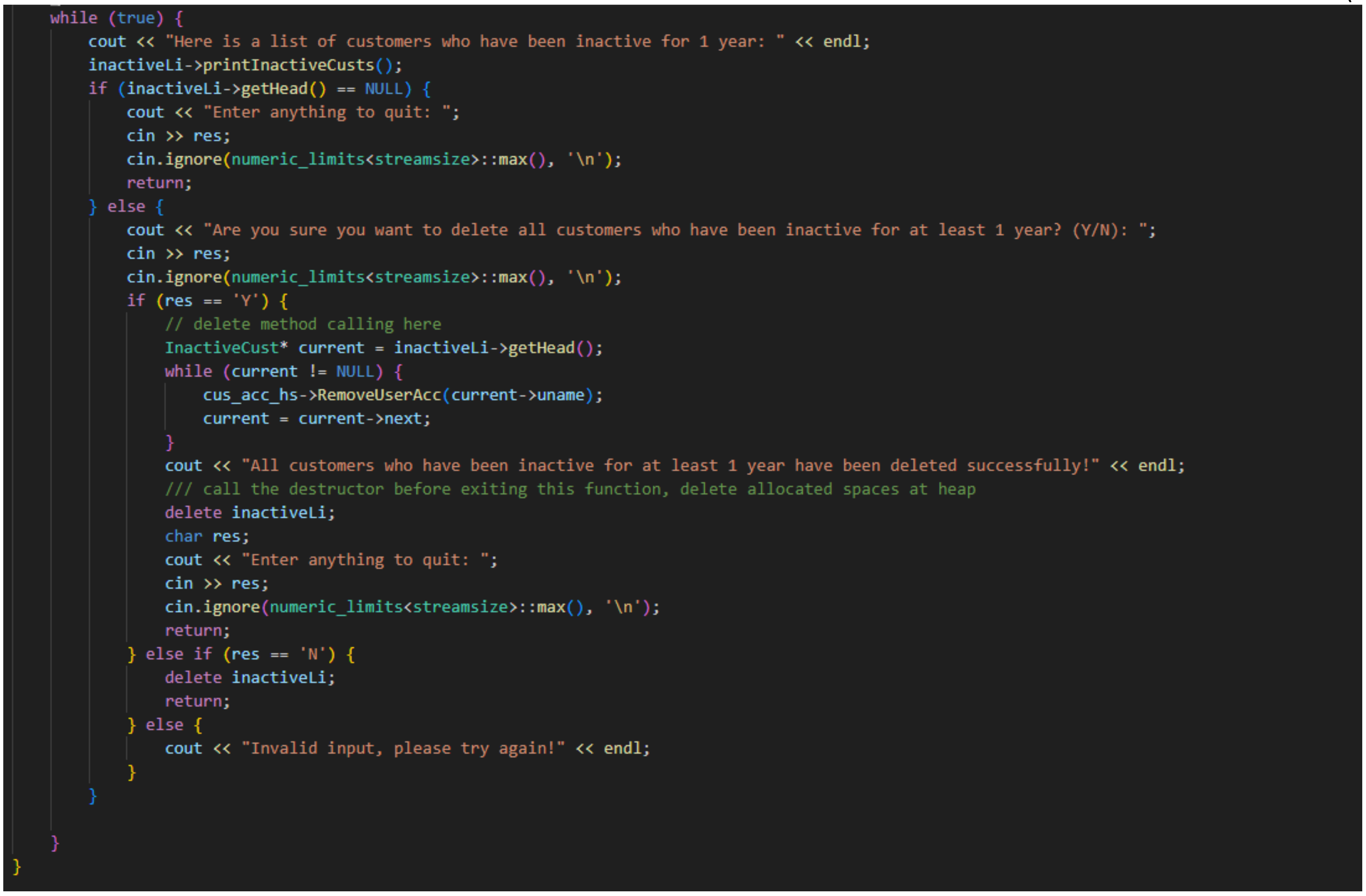
After the normal users enter the corrected admins' credentials, they are redirected to the home page of admins by calling the function as mentioned in *Figure 4.10.1*. Again, this home page is just about some simple **if else** statements handling the entered numbers, then redirect the admins by calling corresponding functions.



```
void DeleteCustAcc(Hash_Table* cus_acc_hs) {
    system("cls");
    // Hash_Table must have a method to traverse thru whole table and return address of customer who has been inactive of 1 year, append it to the linked list
    // return NULL once whole table has been traversed completely. stop the WHILE loop.
    // since we needa list out, hence we need linked list to store all users who has been inactive for 1 year
    // (the linked list shud have a method to delete all its allocated heap memory after this function).
    // display each user, and ask if admin wanna proceed to delete all of them. If yes, then call RemoveItem();
    cout << "*****Delete Customers' Accounts Page*****" << endl;
    char res;
    InactiveCustsList* inactiveLi = new InactiveCustsList;
    UserAcc* current = NULL;
    time_t now = time(0);
    tm* ltm = localtime(&now);
    int diff_year;
    int diff_month;
    for (int i=0; i<cus_acc_hs->getTableSize(); i++) {
        // traverse thru each bucket
        current = cus_acc_hs->getBucketAtIndex(i);
        if (current->uname != "empty") {
            diff_year = ltm->tm_year - current->last_active_time->tm_year;
            diff_month = ltm->tm_mon - current->last_active_time->tm_mon;
            //cout << "dif year = " << diff_year << endl;
            //cout << "dif mon = " << diff_month << endl;
            if ((diff_year == 1 && diff_month >= 0) || diff_year >= 2) {
                inactiveLi->append(current->uname);
            }
            // traverse thru each linked list
            current = current->next;
            while (current != NULL) {
                diff_year = ltm->tm_year - current->last_active_time->tm_year;
                diff_month = ltm->tm_mon - current->last_active_time->tm_mon;

                if ((diff_year == 1 && diff_month >= 0) || diff_year >= 2) {
                    inactiveLi->append(current->uname);
                }
                current = current->next;
            }
        }
    }
```

Figure 4.10.2: DeleteCustAcc() for Admins – Part 1

inactiveLi is a singly linked list containing all customers' names who have been inactive for more than 1 year. The **for** loop is to traverse through all buckets in the hash table. If the bucket contains some nodes (which means **current->uname != "empty"**), then the linked list will be traversed from front to end to see if there is any inactive customers. If there is, just append the customer's name into the *inactiveLi*.



```
    while (true) {
        cout << "Here is a list of customers who have been inactive for 1 year: " << endl;
        inactiveLi->printInactiveCusts();
        if (inactiveLi->getHead() == NULL) {
            cout << "Enter anything to quit: ";
            cin >> res;
            cin.ignore(numeric_limits<streamsize>::max(), '\n');
            return;
        } else {
            cout << "Are you sure you want to delete all customers who have been inactive for at least 1 year? (Y/N): ";
            cin >> res;
            cin.ignore(numeric_limits<streamsize>::max(), '\n');
            if (res == 'Y') {
                // delete method calling here
                InactiveCust* current = inactiveLi->getHead();
                while (current != NULL) {
                    cus_acc_hs->RemoveUserAcc(current->uname);
                    current = current->next;
                }
                cout << "All customers who have been inactive for at least 1 year have been deleted successfully!" << endl;
                /// call the destructor before exiting this function, delete allocated spaces at heap
                delete inactiveLi;
                char res;
                cout << "Enter anything to quit: ";
                cin >> res;
                cin.ignore(numeric_limits<streamsize>::max(), '\n');
                return;
            } else if (res == 'N') {
                delete inactiveLi;
                return;
            } else {
                cout << "Invalid input, please try again!" << endl;
            }
        }

    }
}
```

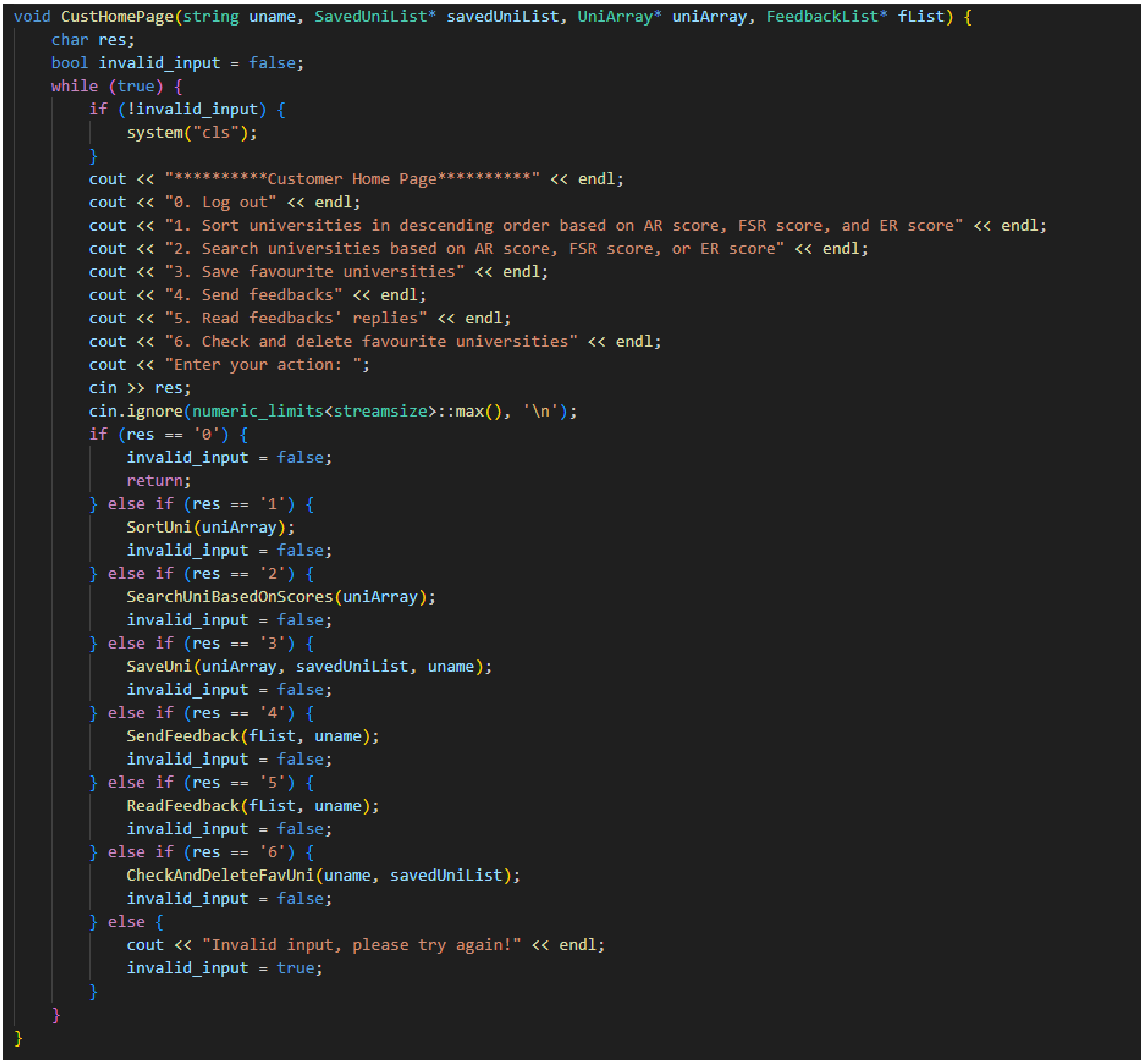
Figure 4.10.3: DeleteCustAcc() for Admins – Part 2

Then, the list of inactive customers' names will be displayed to the admins and ask for further confirmation if they want to delete all inactive customers. If they reply "Y", then the *inactiveList* will be traversed through from front to end and call *RemoveUserAcc()* to delete the inactive accounts at hash table. Before exiting the page, **delete inactiveList** should be executed to avoid memory leakage.

For the other functions in **AdminPages.cpp**, they consist of some simple **while**, **if**, and **else** statements which are not worth explaining (due to the page number constraints), the core algorithms had been done in the classes of those data structure. For more information about those functions, please refer to the submitted zip file.

4.11 CustomerPages.cpp

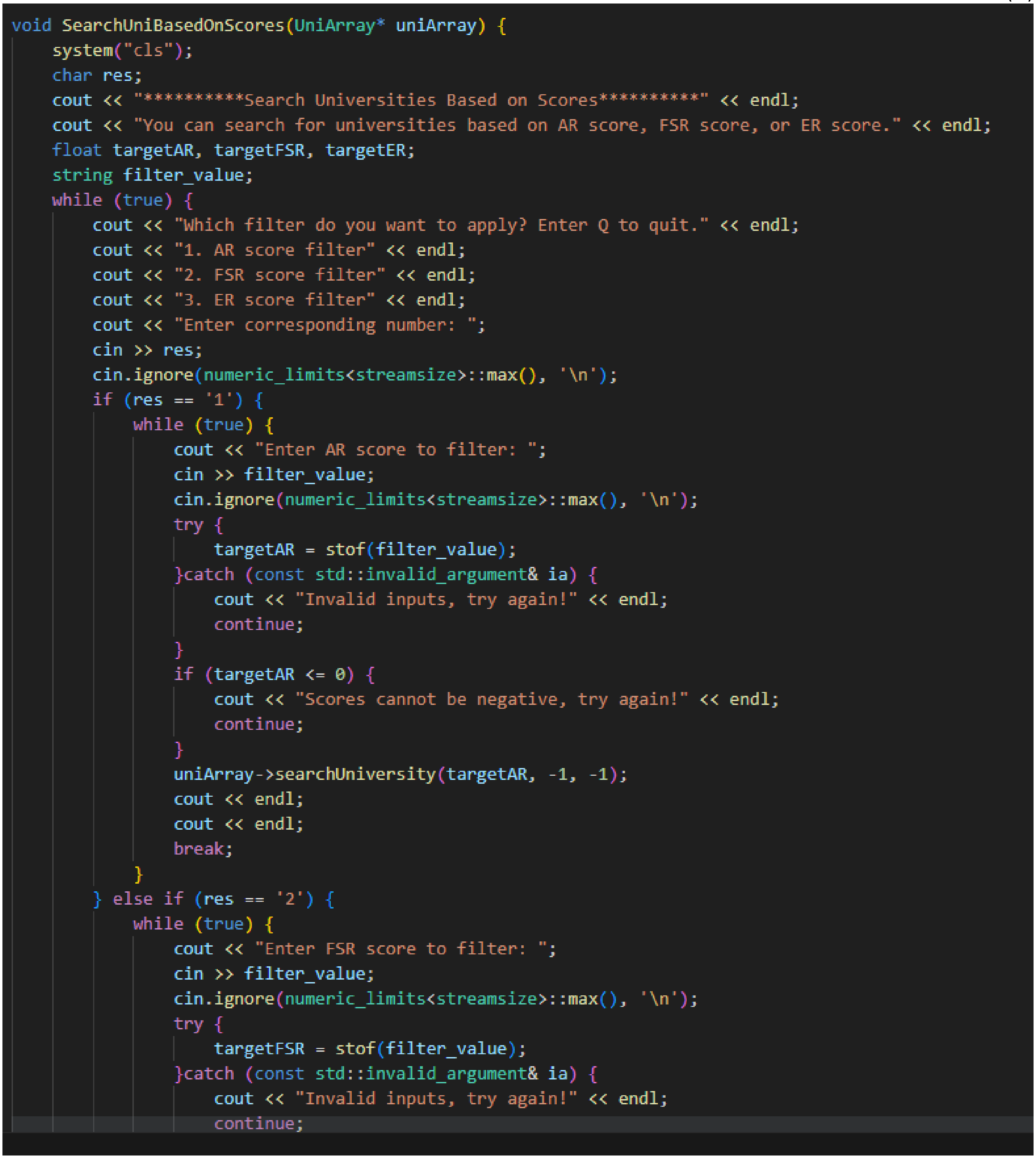
Inside **CustomerPages.cpp**, this C++ file keeps functions of interfaces for customers.



```cpp
void CustHomePage(string uname, SavedUniList* savedUniList, UniArray* uniArray, FeedbackList* fList) {
    char res;
    bool invalid_input = false;
    while (true) {
        if (!invalid_input) {
            system("cls");
        }
        cout << "**********Customer Home Page**********" << endl;
        cout << "0. Log out" << endl;
        cout << "1. Sort universities in descending order based on AR score, FSR score, and ER score" << endl;
        cout << "2. Search universities based on AR score, FSR score, or ER score" << endl;
        cout << "3. Save favourite universities" << endl;
        cout << "4. Send feedbacks" << endl;
        cout << "5. Read feedbacks' replies" << endl;
        cout << "6. Check and delete favourite universities" << endl;
        cout << "Enter your action: ";
        cin >> res;
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
        if (res == '0') {
            invalid_input = false;
            return;
        } else if (res == '1') {
            SortUni(uniArray);
            invalid_input = false;
        } else if (res == '2') {
            SearchUniBasedOnScores(uniArray);
            invalid_input = false;
        } else if (res == '3') {
            SaveUni(uniArray, savedUniList, uname);
            invalid_input = false;
        } else if (res == '4') {
            SendFeedback(fList, uname);
            invalid_input = false;
        } else if (res == '5') {
            ReadFeedback(fList, uname);
            invalid_input = false;
        } else if (res == '6') {
            CheckAndDeleteFavUni(uname, savedUniList);
            invalid_input = false;
        } else {
            cout << "Invalid input, please try again!" << endl;
            invalid_input = true;
        }
    }
}
```

Figure 4.11.1: Home Page for Customers

After the normal users enter the corrected customers' credentials, they are redirected to the home page of customers by calling the function as mentioned in *Figure 4.11.1*. Again, this home page is just about some simple **if else** statements handling the entered numbers, then redirect the admins by calling corresponding functions.



```
void SearchUniBasedOnScores(UniArray* uniArray) {
    system("cls");
    char res;
    cout << "**********Search Universities Based on Scores**********" << endl;
    cout << "You can search for universities based on AR score, FSR score, or ER score." << endl;
    float targetAR, targetFSR, targetER;
    string filter_value;
    while (true) {
        cout << "Which filter do you want to apply? Enter Q to quit." << endl;
        cout << "1. AR score filter" << endl;
        cout << "2. FSR score filter" << endl;
        cout << "3. ER score filter" << endl;
        cout << "Enter corresponding number: ";
        cin >> res;
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
        if (res == '1') {
            while (true) {
                cout << "Enter AR score to filter: ";
                cin >> filter_value;
                cin.ignore(numeric_limits<streamsize>::max(), '\n');
                try {
                    targetAR = stof(filter_value);
                }catch (const std::invalid_argument& ia) {
                    cout << "Invalid inputs, try again!" << endl;
                    continue;
                }
                if (targetAR <= 0) {
                    cout << "Scores cannot be negative, try again!" << endl;
                    continue;
                }
                uniArray->searchUniversity(targetAR, -1, -1);
                cout << endl;
                cout << endl;
                break;
            }
        } else if (res == '2') {
            while (true) {
                cout << "Enter FSR score to filter: ";
                cin >> filter_value;
                cin.ignore(numeric_limits<streamsize>::max(), '\n');
                try {
                    targetFSR = stof(filter_value);
                }catch (const std::invalid_argument& ia) {
                    cout << "Invalid inputs, try again!" << endl;
                    continue;
```

Figure 4.11.2: A Part of SearchUniBasedOnScores() for Customers

Figure above shows the page allowing customers to search for universities based on either AR score, or FSR score, or ER score. Some simple **if**, **else**, and **while** statements are used again to ensure proper flow of the system, the **try** and **catch** statements are used to ensure customers enter some valid inputs. If the inputs cannot be converted into floating type, the exception raised by *stof()* will be caught and raise warning to the customers.

For the other functions in **CustomerPages.cpp**, they consist of some simple **while**, **if**, and **else** statements which are not worth explaining (due to the page number constraints), the core algorithms had been done in the classes of those data structures. For more information about those functions, please refer to the submitted zip file.

5. Implementation and results of 2 search and 2 sort algorithms

5.1 Source Codes of Red Black Tree Sort

The Red-Black Tree algorithm is a significant component of the university recommendation system since it allows for the efficient and effective sorting of university data. The binary search tree served as inspiration for this sophisticated data structure, which provides a flexible and scalable architecture for storing and organising information.

The Red-Black Tree sorting technique ensures the tree stays height-balanced by introducing nodes into it while preserving the tree's balanced structure through a set of attributes, allowing for quick and efficient sorting of academic content. The Red-Black Tree sorting method ensures that the university recommendation system's performance is not degraded by the large size of the institutions to be processed by keeping the tree in a state of equilibrium.

University data in the university recommendation system may be sorted by a variety of factors using the Red-Black Tree algorithm, including the name of the institution, the academic reputation score, and the faculty-to-student ratio. Users can easily locate the relevant university data they need thanks to the algorithm's proficiency in processing and organising massive data sets, thereby bolstering the system's robustness and efficiency.

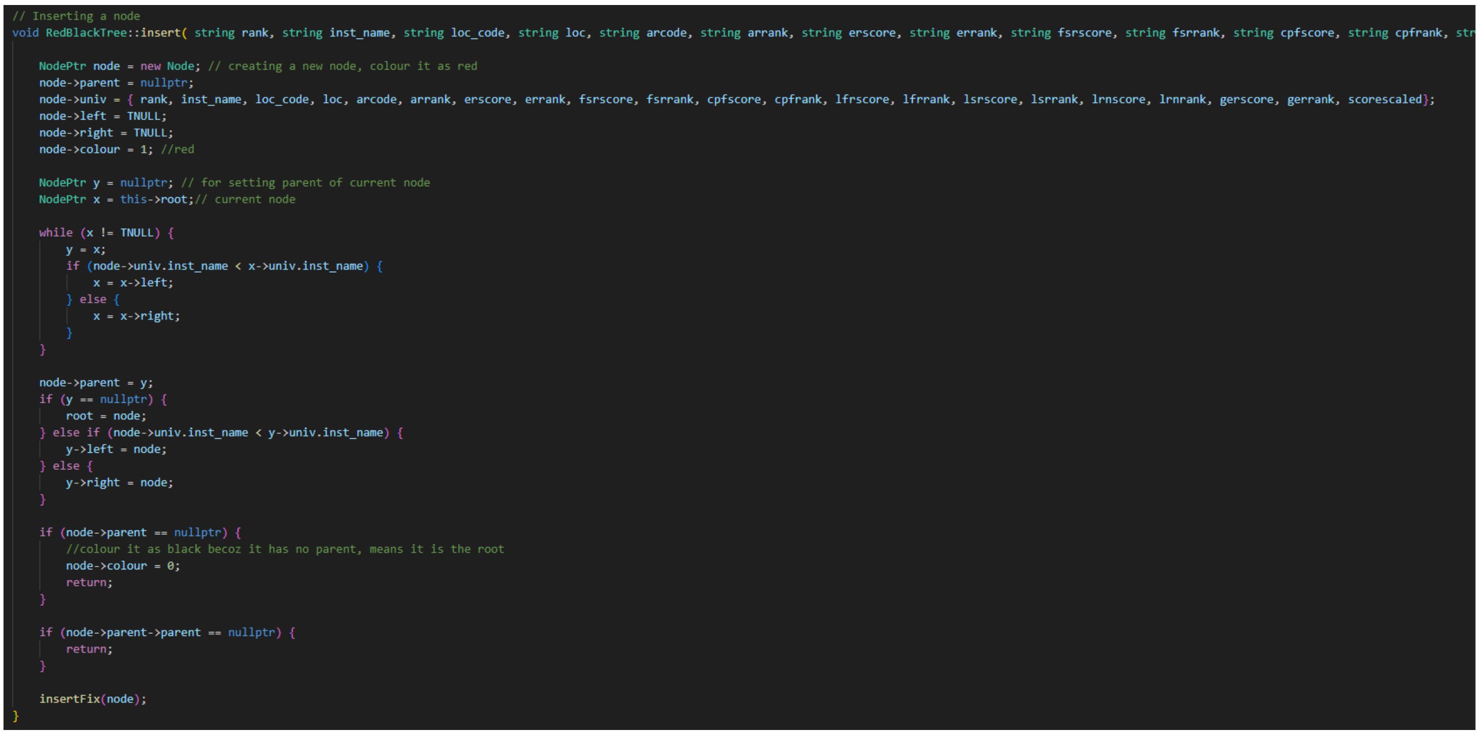
The incorporation of the Red-Black Tree sorting algorithm into the university recommendation system exemplifies how complex data structures may be put to use in practical situations like those faced while deciding where to enrol in college. When compared to similar university recommendation systems, ours stands out because to the innovative sorting approach it employs to improve the user experience and boost speed.

In conclusion, the Red-Black Tree sorting algorithm is an essential component of the university recommendation system, facilitating the organisation and accessibility of academic resources for students.

Using this state-of-the-art data format, we illustrate the potential use of sophisticated algorithms in settings as varied as policymaking in higher education. Innovatively using the Red-Black Tree sorting algorithm, our university recommendation system not only ensures the system's capacity to manage expanding data volumes but also gives an unrivalled user experience, setting it apart from the competition.

University recommendations must be flexible enough to accommodate an ever-increasing number of institutions, and the Red-Black Tree sorting method is essential for managing large datasets. The Red-Black Tree algorithm is a key part of the system's adaptability and long-term value since it provides consistently high sorting performance even as the number of colleges grows.

Ultimately, the Red-Black Tree sorting algorithm is crucial to the success of the university recommendation system since it provides an effective way of organising and gaining access to relevant university data. The implementation of this cutting-edge sorting method exemplifies how complicated algorithms may be put to use in everyday settings, such as the strategic planning and decision-making involved in tertiary education. The Red-Black Tree sorting algorithm was creatively incorporated into our university recommendation system in a way that sets it apart from competing solutions, resulting in a first-rate user experience, and futureproofing for the system.



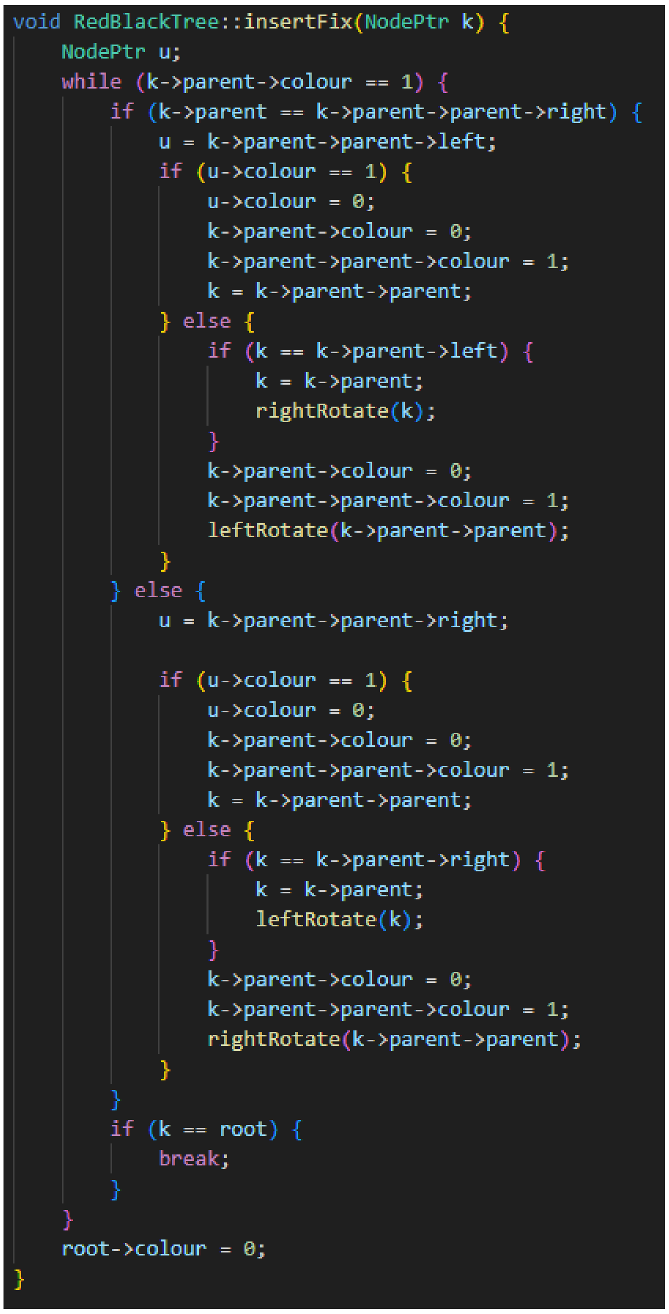
```
// Inserting a node
void RedBlackTree::insert( string rank, string inst_name, string loc_code, string loc, string arcode, string arrank, string erscore, string errank, string fsrscore, string fsrrank, string cpfscore, string cpfrank, str

    NodePtr node = new Node; // creating a new node, colour it as red
    node->parent = nullptr;
    node->univ = { rank, inst_name, loc_code, loc, arcode, arrank, erscore, errank, fsrscore, fsrrank, cpfscore, cpfrank, lfrscore, lfrrank, lsrscore, lsrrank, lrnscore, lrnrank, gerscore, gerrank, scorescaled};
    node->left = TNULL;
    node->right = TNULL;
    node->colour = 1; //red

    NodePtr y = nullptr; // for setting parent of current node
    NodePtr x = this->root;// current node

    while (x != TNULL) {
        y = x;
        if (node->univ.inst_name < x->univ.inst_name) {
            x = x->left;
        } else {
            x = x->right;
        }
    }

    node->parent = y;
    if (y == nullptr) {
        root = node;
    } else if (node->univ.inst_name < y->univ.inst_name) {
        y->left = node;
    } else {
        y->right = node;
    }

    if (node->parent == nullptr) {
        //colour it as black becoz it has no parent, means it is the root
        node->colour = 0;
        return;
    }

    if (node->parent->parent == nullptr) {
        return;
    }

    insertFix(node);
}
```

Figure 5.1.1: insert() method in RedBlackTree class

insert() method takes in all the fields of a university record from the CSV file. A *node* is created at heap and its *univ* will be initialized with all the fields taken in. Its colour will be set to red (indicated by integer 1), parent will be NULL, left child and right child will be TNULL.

Two additional pointers, *y* and *x*, are initialized. *y* is used to keep track of the parent of the new node being inserted, while *x* will travers the tree to find the appropriate position for the new node. A while loop is executed to find the appropriate position for insertion of the new node, the university name of the new node is compared with the university name of the *x*. If the university name of new node is smaller than of *x*, visit its left child; Else visit its right child.

The parent of the new node will be set to *y*. If the *y* is NULL, it means there is no node at the tree, hence the root node is set to the new node. If the new node has no parent, it means it is the root, hence it must be coloured black to maintain the first property of a red black tree. *insertFix()* method is called for rebalancing the tree to ensure the properties of red black tree are still fulfilled.



```
void RedBlackTree::insertFix(NodePtr k) {
    NodePtr u;
    while (k->parent->colour == 1) {
        if (k->parent == k->parent->parent->right) {
            u = k->parent->parent->left;
            if (u->colour == 1) {
                u->colour = 0;
                k->parent->colour = 0;
                k->parent->parent->colour = 1;
                k = k->parent->parent;
            } else {
                if (k == k->parent->left) {
                    k = k->parent;
                    rightRotate(k);
                }
                k->parent->colour = 0;
                k->parent->parent->colour = 1;
                leftRotate(k->parent->parent);
            }
        } else {
            u = k->parent->parent->right;

            if (u->colour == 1) {
                u->colour = 0;
                k->parent->colour = 0;
                k->parent->parent->colour = 1;
                k = k->parent->parent;
            } else {
                if (k == k->parent->right) {
                    k = k->parent;
                    leftRotate(k);
                }
                k->parent->colour = 0;
                k->parent->parent->colour = 1;
                rightRotate(k->parent->parent);
            }
        }
        if (k == root) {
            break;
        }
    }
    root->colour = 0;
}
```

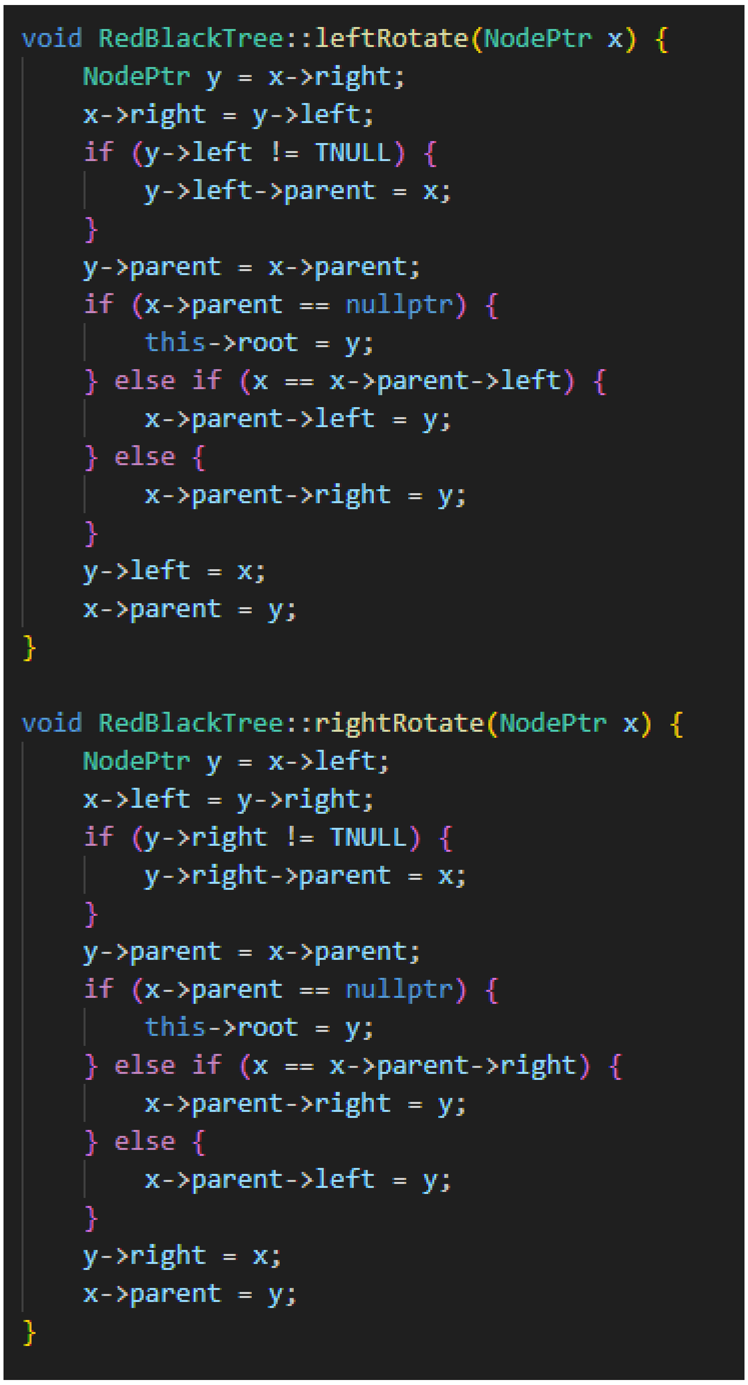
Figure 5.1.2: insertFix() method in RedBlackTree class

insertFix() method involves rotations and recolouring of nodes to restore balance, it takes in the *k* which represents the newly inserted node.

The while loop is executed when the colour of the parent of the new node is red. Inside the while loop, the new node is checked if it is the right child of the grandparent.

If the new node is the right child of the grandparent: The *u* means the uncle node of the new node (sibling of the parent node), if the colour of uncle node is red, recolouration of few nodes is carried out; otherwise recolouration, right rotation and left rotation are carried out.

If the new node is the left child of the grandparent: The *u* means the uncle node of the new node (sibling of the parent node). Recolouration of a few nodes and rotations are also carried out based on different cases as shown in *Figure 5.1.2*.



```
void RedBlackTree::leftRotate(NodePtr x) {
    NodePtr y = x->right;
    x->right = y->left;
    if (y->left != TNULL) {
        y->left->parent = x;
    }
    y->parent = x->parent;
    if (x->parent == nullptr) {
        this->root = y;
    } else if (x == x->parent->left) {
        x->parent->left = y;
    } else {
        x->parent->right = y;
    }
    y->left = x;
    x->parent = y;
}

void RedBlackTree::rightRotate(NodePtr x) {
    NodePtr y = x->left;
    x->left = y->right;
    if (y->right != TNULL) {
        y->right->parent = x;
    }
    y->parent = x->parent;
    if (x->parent == nullptr) {
        this->root = y;
    } else if (x == x->parent->right) {
        x->parent->right = y;
    } else {
        x->parent->left = y;
    }
    y->right = x;
    x->parent = y;
}
```

Figure 5.1.3: leftRotate() and rightRotate() in RedBlackTree class

5.2 Source Codes of Quick Sort

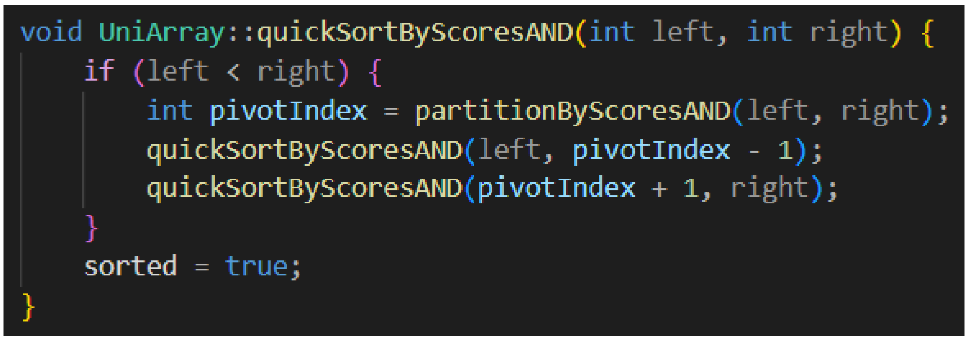
The Quick Sort algorithm is a vital component of the university recommendation system project since it offers a very efficient technique of classifying and structuring scholarly materials. Quick Sort, a divide-and-conquer sorting strategy, is often praised for its speed and efficiency; as a consequence, it is a wonderful match for handling the data related to universities.

To swiftly sort data, Quick Sort selects a single 'pivot' item from the dataset and separates the remainder into two groups, smaller and larger, depending on whether or not they are less than or more than the pivot. The same operation is then performed recursively on both subsets, leading to a completely sorted dataset.

The Quick Sort algorithm is used inside the university recommendation system to sort university data in accordance with several criteria, such as the institution's name, academic reputation score, and faculty-to-student ratio score, among others. Users are able to rapidly get the information they need from the university because to the algorithm's ability to process and organise large datasets, which also improves the system's efficiency and effectiveness.

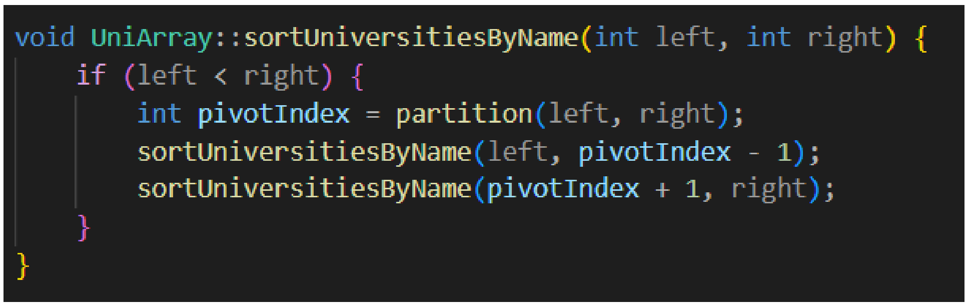
The incorporation of a state-of-the-art sorting algorithm like Quick Sort into a college and university recommendation system shows the value of such methods in administrative settings. The university's recommendation system distinguishes out from the crowd because to its improved usability and performance, both of which are made possible by the use of this effective categorization approach.

Last but not least, the Quick Sort algorithm is a crucial component of the university recommendation system as it facilitates the quick and effective structuring of university data. The university recommendation system stands out from the crowd because of the superior user experience it provides as a result of the sophisticated sorting approach it employs.



```
void UniArray::quickSortByScoresAND(int left, int right) {
    if (left < right) {
        int pivotIndex = partitionByScoresAND(left, right);
        quickSortByScoresAND(left, pivotIndex - 1);
        quickSortByScoresAND(pivotIndex + 1, right);
    }
    sorted = true;
}
```

Figure 5.2.1: Sort algorithm



```cpp
void UniArray::sortUniversitiesByName(int left, int right) {
    if (left < right) {
        int pivotIndex = partition(left, right);
        sortUniversitiesByName(left, pivotIndex - 1);
        sortUniversitiesByName(pivotIndex + 1, right);
    }
}
```

Figure 5.2.2: partition() and sortUniversitiesByName() in 'UniArray' class

Our university administration system benefits greatly from quicksort. So, their implementation improves system functioning.

For sorting institutions by name or score, we use quicksort. Quicksort's average-case time complexity of O(n log n) is ideal for datasets that may include several colleges. Our system orders colleges by AR, FSR, and ER using quicksort in the quickSortByScoresAND() method. The function uses 'Divide and Conquer' to divide the array and sort its subarrays. The partitioning procedure entails choosing a pivot and reordering the institutions such that those with better scores come first and those with lower scores come last. Sorted arrays provide efficient binary search.

In conclusion, our project's data retrieval and display skills are substantially improved by quicksort. This algorithm makes sorting colleges in enormous databases faster and more efficient. Our university management system relies on this practical approach.

5.3 Source Codes of Red Black Tree Search

University recommendation system rely heavily on the Red-Black Tree search algorithm, which plays a critical role in quickly discovering specific student information. This complex data structure is a subtype of a binary search tree, distinguished by its unique balancing algorithm that ensures the tree maintains its vertical symmetry even after nodes are added or removed. Red-Black Trees are well-suited to our recommendation system because of their remarkable searching, insertion, and deletion capabilities while still maintaining this equilibrium.

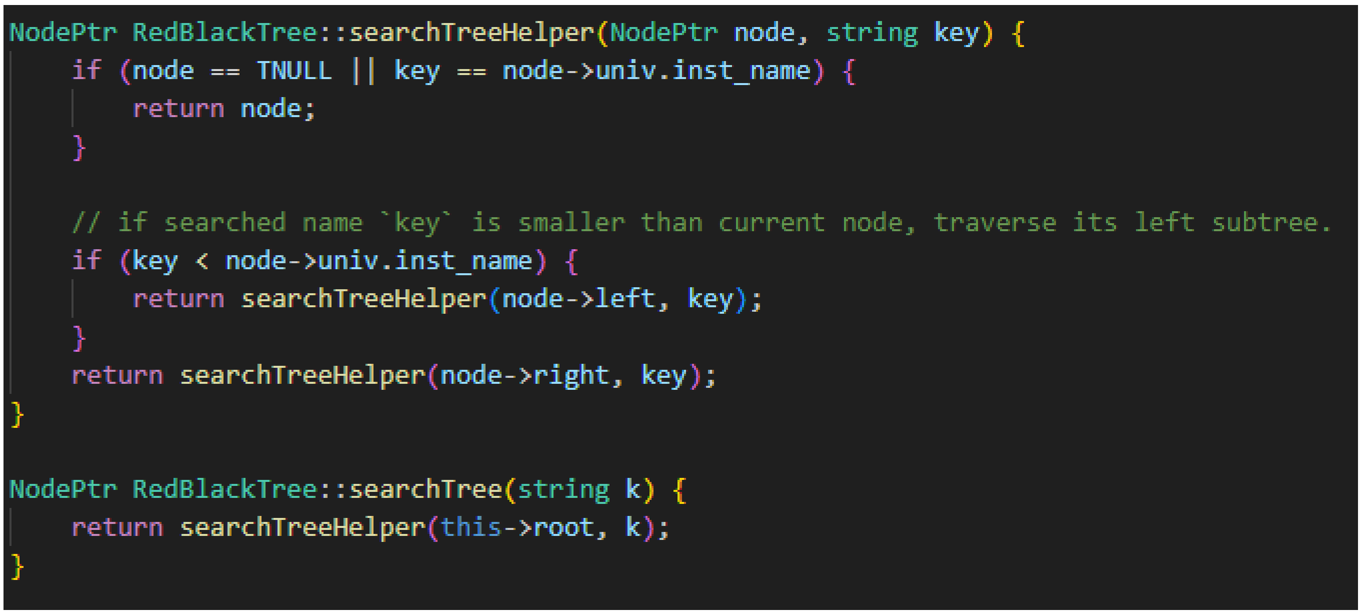
The Red-Black Tree search technique is used inside the university recommendation system to help users quickly obtain particular university data by using the tree's inherent organisation. Each institution is represented by a node in the tree, and each node has a key based on some distinguishing feature of the school, such as its location or name. In order to maintain harmony and provide the highest possible search efficiency, nodes in the tree are assigned one of two colours: red or black.

Finding anything requires starting at the bottom of a Red-Black Tree and working your way up, comparing the keys of each node with the target key as you go. The method moves to the left or right child node of the current node depending on the results of the comparison, exploring the tree until it finds the target university or a null node.

Our university recommendation system guarantees customers quick and efficient access to specific institution information by including the Red-Black Tree search algorithm. The main reason the data structure is efficient is because it can improve the system’s functionality. However, when it comes to the limitations, it also highlights how sophisticated data structures may be put to use in everyday circumstances like strategic planning and policymaking in higher education. With this meticulously designed phrase, we want to convey our dedication to providing a one-of-a-kind, high-quality service to our consumers using the Red-Black Tree search algorithm we developed.

The Red-Black Tree search technique guarantees that the university recommendation system will scale indefinitely as the number of schools in the dataset increases. Our recommendation system is able to expand to a growing dataset without compromising speed because to the Red-Black Tree's intrinsic capacity to self-balance.

In conclusion, the Red-Black Tree search algorithm is an integral part of the university recommendation system, providing consumers with easy and quick access to relevant data about certain universities. Applying theoretical concepts in the context of planning and decision-making in higher education is made possible through the use of this cutting-edge data structure. Our university recommendation system stands out from the competition because to its inventive use of the Red-Black Tree search algorithm, which not only provides users with an unrivalled experience but also guarantees the system's scalability for future growth.



```cpp
NodePtr RedBlackTree::searchTreeHelper(NodePtr node, string key) {
    if (node == TNULL || key == node->univ.inst_name) {
        return node;
    }

    // if searched name `key` is smaller than current node, traverse its left subtree.
    if (key < node->univ.inst_name) {
        return searchTreeHelper(node->left, key);
    }
    return searchTreeHelper(node->right, key);
}

NodePtr RedBlackTree::searchTree(string k) {
    return searchTreeHelper(this->root, k);
}
```

Figure 5.3.1: searchTree() and searchTreeHelper() in RedBlackTree class

To search in the red black tree, *searchTree()* method is called and it takes in university name. Inside it, *searchTreeHelper()* method is called and it takes in a node and university name, *key*, as arguments. The *node* will be further updated by either traversing left or right.

If *node* equals TNULL or there is a matching node, then the *node* is returned. In this case, *node* equals TNULL means the university searched by users does not exist, warning message should be printed to the users.

If the searched name *key* is smaller than of *node*, the left subtree of the *node* is traversed; Else, the right subtree of the *node* is traversed.

5.4 Source Codes of Binary Search

The binary search algorithm plays an essential role in the university recommendation system by making specific university data considerably more accessible. This tried-and-true method of searching, which performs best on ordered collections, improves search efficiency by repeatedly bisecting the search interval. Quickly retrieving scholarly information is made possible by the binary search algorithm's logarithmic search method, which also increases the system's robustness.

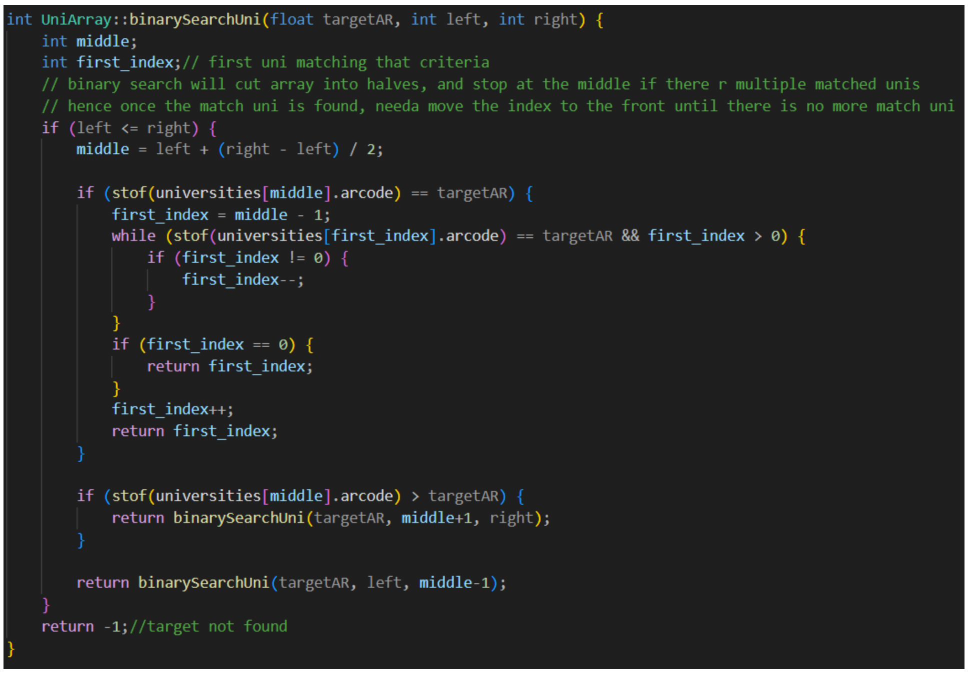
The binary search method works in concert with sorted information, such as a list of institutions arranged alphabetically or by ranking, to provide an efficient and effective university recommendation system. To begin its search, the algorithm compares the sought-after value to the item in the list's geometric centre. As a result of this comparison, the algorithm either zeroes in on the correct school or discards half of the candidates while continuing the search in the remaining half. Until either the desired school is found, or the search interval is exhausted, the procedure will continue.

User access to relevant university information is maximised while computing costs are kept to a minimum by integrating the binary search algorithm into the university recommendation system. In addition to enhancing the system's overall functioning, this updated search approach exemplifies how fundamental search algorithms may be put to use in the real world, in this instance, in the context of strategic planning and policymaking in higher education. Our commitment to bringing you a one-of-a-kind, best-in-class solution is shown in the clarity, concision, formality, inclusion, and advanced terminology we put into building the binary search algorithm.

In conclusion, the binary search algorithm is crucial to the success of the university's advising system since it facilitates the rapid retrieval of desired student information. To show how even the most basic algorithms may be put to use in the realm of policy and programme creation in higher education, we use this basic search method. By creatively implementing the binary search algorithm, our university recommendation system not only provides users with an unmatched experience, but also exemplifies the significance of efficient search algorithms in data management and decision-making.

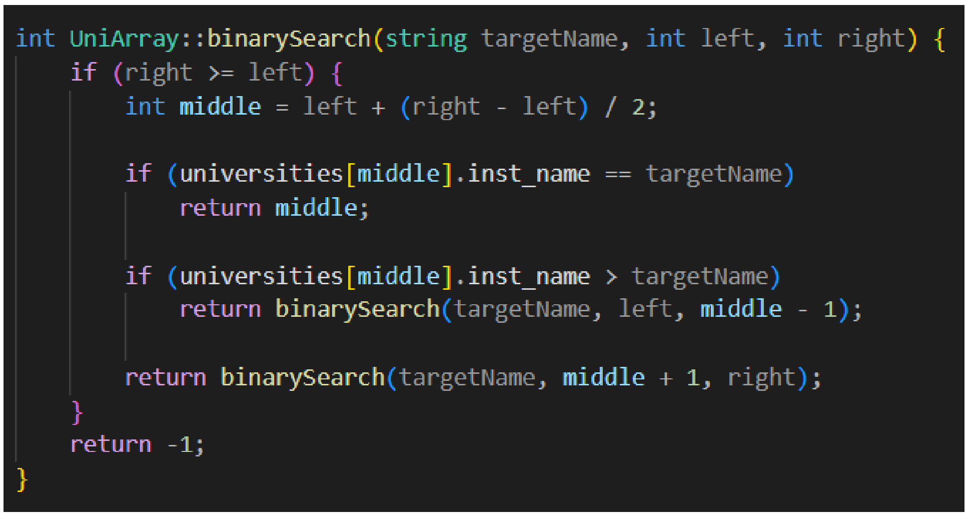
To ensure the university recommendation system can keep growing as more institutions are added, the effectiveness of the binary search algorithm is particularly crucial for processing enormous datasets. The ability of the binary search algorithm to scale with a growing university database without compromising search quality is a major factor in the system's adaptability and endurance.

In conclusion, the binary search algorithm is essential to the success of the university recommendation system since it provides consumers with a quick and reliable way to acquire specific university data. As an example of how fundamental algorithms may be put to use in the real world, the use of this time-tested search strategy in contexts like strategic planning in higher education is provided. Our university recommendation system stands out from the competition because to the novel and inventive way in which the binary search algorithm is incorporated into it, resulting in a greater user experience, and making the system future-proof.



```cpp
int UniArray::binarySearchUni(float targetAR, int left, int right) {
    int middle;
    int first_index;// first uni matching that criteria
    // binary search will cut array into halves, and stop at the middle if there r multiple matched unis
    // hence once the match uni is found, needa move the index to the front until there is no more match uni
    if (left <= right) {
        middle = left + (right - left) / 2;

        if (stof(universities[middle].arcode) == targetAR) {
            first_index = middle - 1;
            while (stof(universities[first_index].arcode) == targetAR && first_index > 0) {
                if (first_index != 0) {
                    first_index--;
                }
            }
            if (first_index == 0) {
                return first_index;
            }
            first_index++;
            return first_index;
        }

        if (stof(universities[middle].arcode) > targetAR) {
            return binarySearchUni(targetAR, middle+1, right);
        }

        return binarySearchUni(targetAR, left, middle-1);
    }
    return -1;//target not found
}
```

Figure 5.4.1: Search algorithm



```cpp
int UniArray::binarySearch(string targetName, int left, int right) {
    if (right >= left) {
        int middle = left + (right - left) / 2;

        if (universities[middle].inst_name == targetName)
            return middle;

        if (universities[middle].inst_name > targetName)
            return binarySearch(targetName, left, middle - 1);

        return binarySearch(targetName, middle + 1, right);
    }
    return -1;
}
```

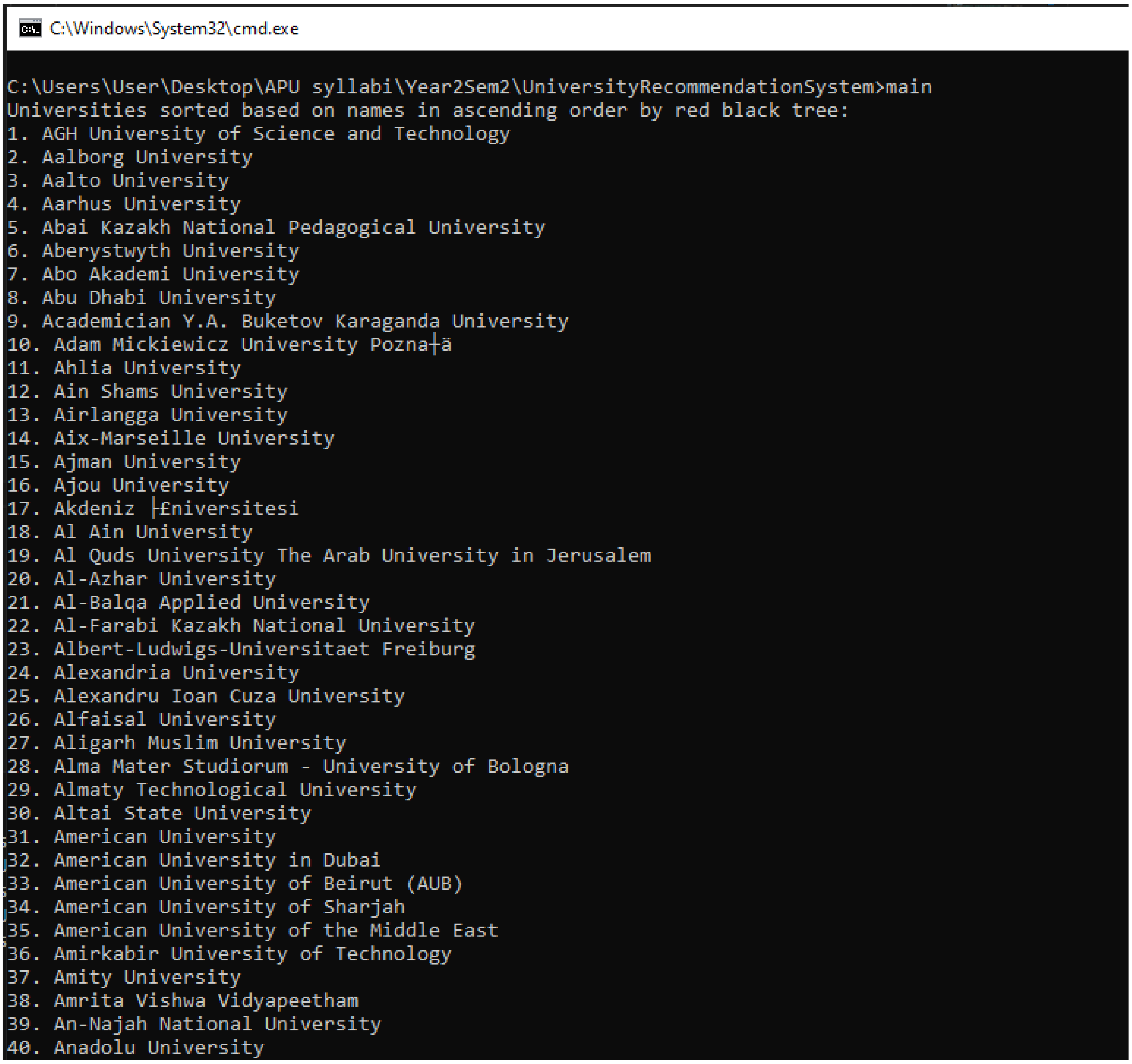
Figure 5.4.2: binarySearch() in 'UniArray' class

Our university administration system benefits greatly from binary search. So, their implementation improves system functioning.

Our method searches by Academic Reputation (AR) score using binary search. Binary search's O(log n) time complexity outperforms linear search's O(n), especially when processing large datasets. After sorting universities, our system uses this efficient search approach. Binary search keeps halves the search space until it finds the right value or runs out of options. In the given code, the binarySearchUni() function partitions the dataset and compares the center element with the target AR. This method continues until it finds the institution with the desired AR score or can no longer split the dataset.

In conclusion, our project's data retrieval and display skills are substantially improved by binary search. This algorithm makes searching colleges in enormous databases faster and more efficient. Our university management system relies on this practical approach.

5.5 Comparing 2 Sort Algorithms



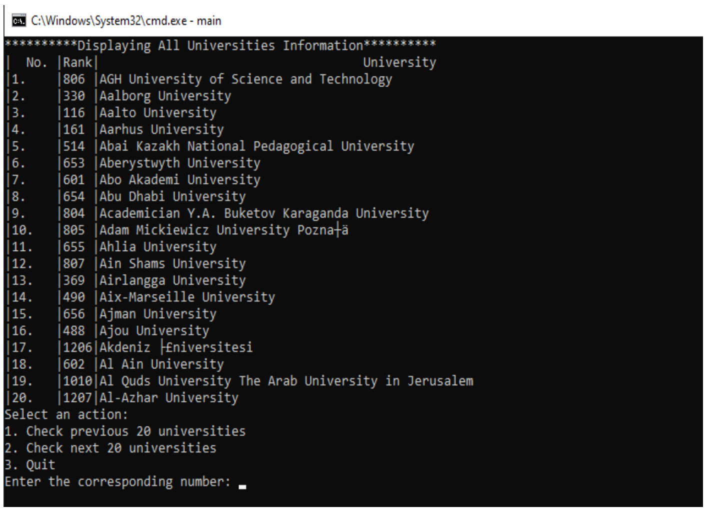
```
C:\Windows\System32\cmd.exe

C:\Users\User\Desktop\APU syllabi\Year2Sem2\UniversityRecommendationSystem>main
Universities sorted based on names in ascending order by red black tree:
1. AGH University of Science and Technology
2. Aalborg University
3. Aalto University
4. Aarhus University
5. Abai Kazakh National Pedagogical University
6. Aberystwyth University
7. Abo Akademi University
8. Abu Dhabi University
9. Academician Y.A. Buketov Karaganda University
10. Adam Mickiewicz University Pozna┼ä
11. Ahlia University
12. Ain Shams University
13. Airlangga University
14. Aix-Marseille University
15. Ajman University
16. Ajou University
17. Akdeniz ├£niversitesi
18. Al Ain University
19. Al Quds University The Arab University in Jerusalem
20. Al-Azhar University
21. Al-Balqa Applied University
22. Al-Farabi Kazakh National University
23. Albert-Ludwigs-Universitaet Freiburg
24. Alexandria University
25. Alexandru Ioan Cuza University
26. Alfaisal University
27. Aligarh Muslim University
28. Alma Mater Studiorum - University of Bologna
29. Almaty Technological University
30. Altai State University
31. American University
32. American University in Dubai
33. American University of Beirut (AUB)
34. American University of Sharjah
35. American University of the Middle East
36. Amirkabir University of Technology
37. Amity University
38. Amrita Vishwa Vidyapeetham
39. An-Najah National University
40. Anadolu University
```

Figure 5.5.1: Sorted Universities Based on Names in Ascending Order by Red Black Tree

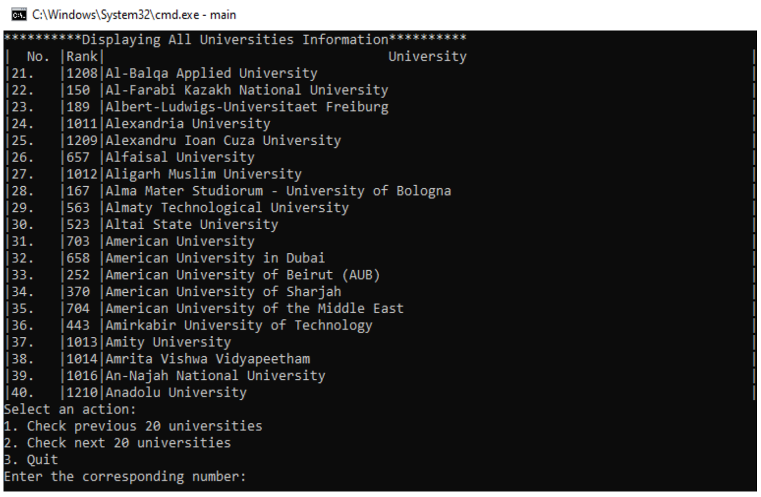
Figure 5.5.1 shows the ascending order of universities based on names (only 40 universities are shown, otherwise it is too much), and the sorting is happening inside red black tree. To get such outputs, the following method is called as shown in *Figure 5.5.2*.

```
cout << "Universities sorted based on names in ascending order by red black tree:" << endl;
rbt->inOrderPrint();
```

Figure 5.5.2: Calling inOrderPrint() method



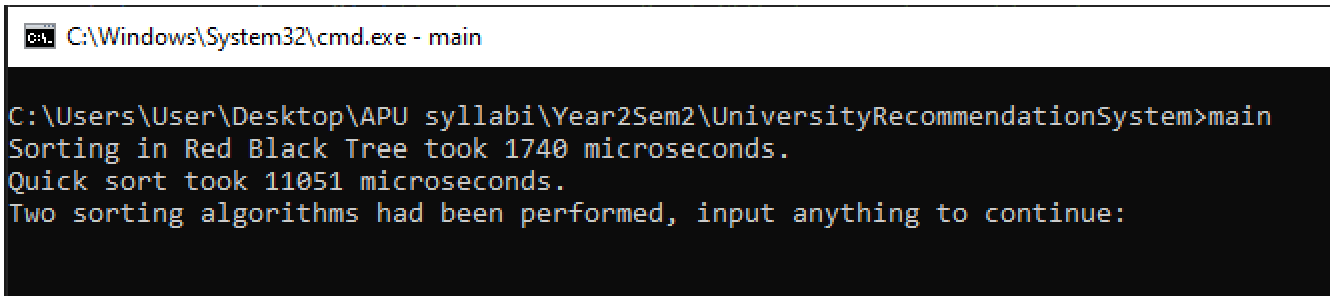
```
C:\Windows\System32\cmd.exe - main
**********Displaying All Universities Information**********
|  No. |Rank|                                    University
|1.    |806 |AGH University of Science and Technology
|2.    |330 |Aalborg University
|3.    |116 |Aalto University
|4.    |161 |Aarhus University
|5.    |514 |Abai Kazakh National Pedagogical University
|6.    |653 |Aberystwyth University
|7.    |601 |Abo Akademi University
|8.    |654 |Abu Dhabi University
|9.    |804 |Academician Y.A. Buketov Karaganda University
|10.   |805 |Adam Mickiewicz University Pozna┼ä
|11.   |655 |Ahlia University
|12.   |807 |Ain Shams University
|13.   |369 |Airlangga University
|14.   |490 |Aix-Marseille University
|15.   |656 |Ajman University
|16.   |488 |Ajou University
|17.   |1206|Akdeniz ├£niversitesi
|18.   |602 |Al Ain University
|19.   |1010|Al Quds University The Arab University in Jerusalem
|20.   |1207|Al-Azhar University
Select an action:
1. Check previous 20 universities
2. Check next 20 universities
3. Quit
Enter the corresponding number:
```



```
C:\Windows\System32\cmd.exe - main
**********Displaying All Universities Information**********
|  No. |Rank|                                    University                                |
|21.   |1208|Al-Balqa Applied University                                                 |
|22.   |150 |Al-Farabi Kazakh National University                                        |
|23.   |189 |Albert-Ludwigs-Universitaet Freiburg                                        |
|24.   |1011|Alexandria University                                                       |
|25.   |1209|Alexandru Ioan Cuza University                                              |
|26.   |657 |Alfaisal University                                                         |
|27.   |1012|Aligarh Muslim University                                                   |
|28.   |167 |Alma Mater Studiorum - University of Bologna                                |
|29.   |563 |Almaty Technological University                                              |
|30.   |523 |Altai State University                                                      |
|31.   |703 |American University                                                         |
|32.   |658 |American University in Dubai                                                |
|33.   |252 |American University of Beirut (AUB)                                         |
|34.   |370 |American University of Sharjah                                              |
|35.   |704 |American University of the Middle East                                      |
|36.   |443 |Amirkabir University of Technology                                          |
|37.   |1013|Amity University                                                            |
|38.   |1014|Amrita Vishwa Vidyapeetham                                                  |
|39.   |1016|An-Najah National University                                                |
|40.   |1210|Anadolu University                                                          |
Select an action:
1. Check previous 20 universities
2. Check next 20 universities
3. Quit
Enter the corresponding number:
```

Figure 5.5.3: Sorted Universities Based on Names in Ascending Order by Quick Sort

Figure 5.5.3 shows the sorted universities by using quick sort, the results are the same as *Figure 5.5.1*.



```
C:\Windows\System32\cmd.exe - main

C:\Users\User\Desktop\APU syllabi\Year2Sem2\UniversityRecommendationSystem>main
Sorting in Red Black Tree took 1740 microseconds.
Quick sort took 11051 microseconds.
Two sorting algorithms had been performed, input anything to continue:
```

Figure 5.5.4: Execution Times of 2 Sort Algorithms

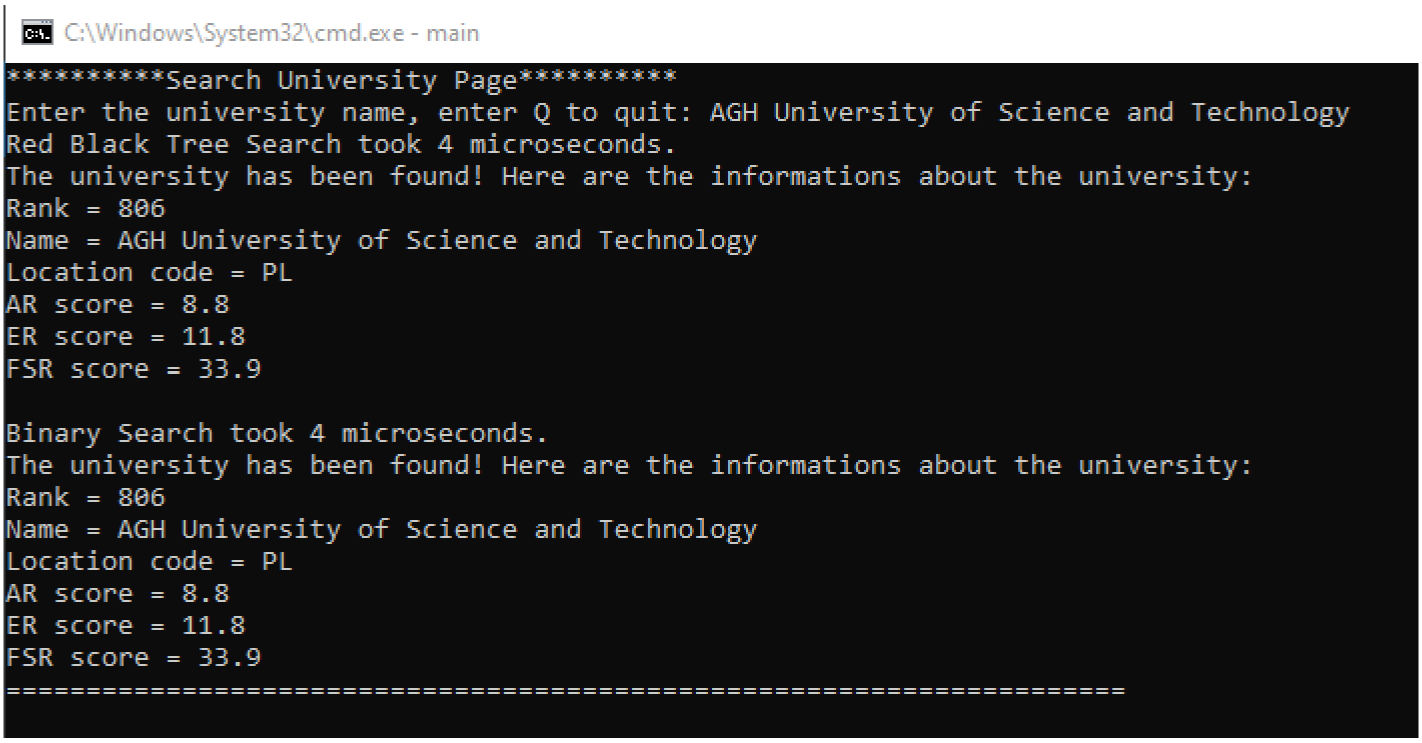
Considerations like update frequency, memory use, and system constraints must be made while comparing these two algorithms for the university recommendation system. Since it allows for fast inserts, deletes, and searches while keeping the tree balanced, Red Black Tree sort is useful in situations that call for frequent updates. However, the colour attribute of each node causes it to use more memory than other approaches.

Quick Sort, on the other hand, shines when dealing with static data or single instance sorting tasks. The elimination of redundant data structures is another advantage of in-place sorting over Red Black Trees. In conclusion, the Red Black Tree sort or Quick Sort algorithms will be utilised depending on the needs and features of the university recommendation system.

From the results in *Figure 5.5.4*, it is obvious that sorting in red black tree was way faster than quick sort. Sorting in red black tree is just insertion of new node into tree while maintaining the properties of a red black tree. Hence, theoretically, time complexity of insertion in red black tree is *O(log N)* while space complexity is *O(N)* (Gupta, n.d.). For quick sort, its average time complexity is *O(N log N)*. Theoretically and practically, sorting in red black tree is faster than quick sort.

Although red black tree is way faster than quick sort, red black tree requires extra memory while retrieving the data. For the quick sort, the elements are arranged in ascending order, and the data can be retrieved from the front to end with time complexity *O(N)* and no memory required for that. For retrieval of data of red black tree in ascending order, in order traversal is carried out with time complexity of *O(N)* (*N* stands for number of nodes) and space complexity of *O(h)* (*h* stands for height of tree) (Pandey, 2022). This is because in order traversal is done with the concept of recursion, the stack frames of each call are stacked in the call stack until the *return* keyword is met, then the stack frame will be popped from the call stack.

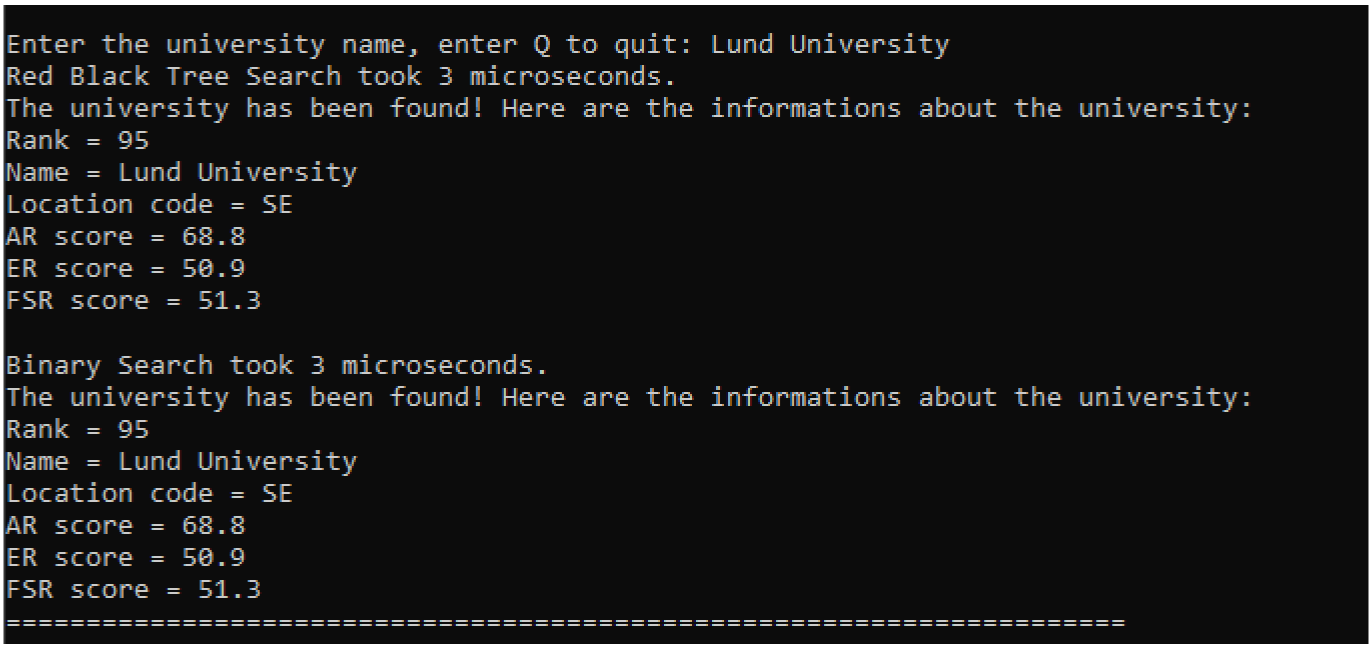
5.6 Comparing 2 Search Algorithms



C:\Windows\System32\cmd.exe - main

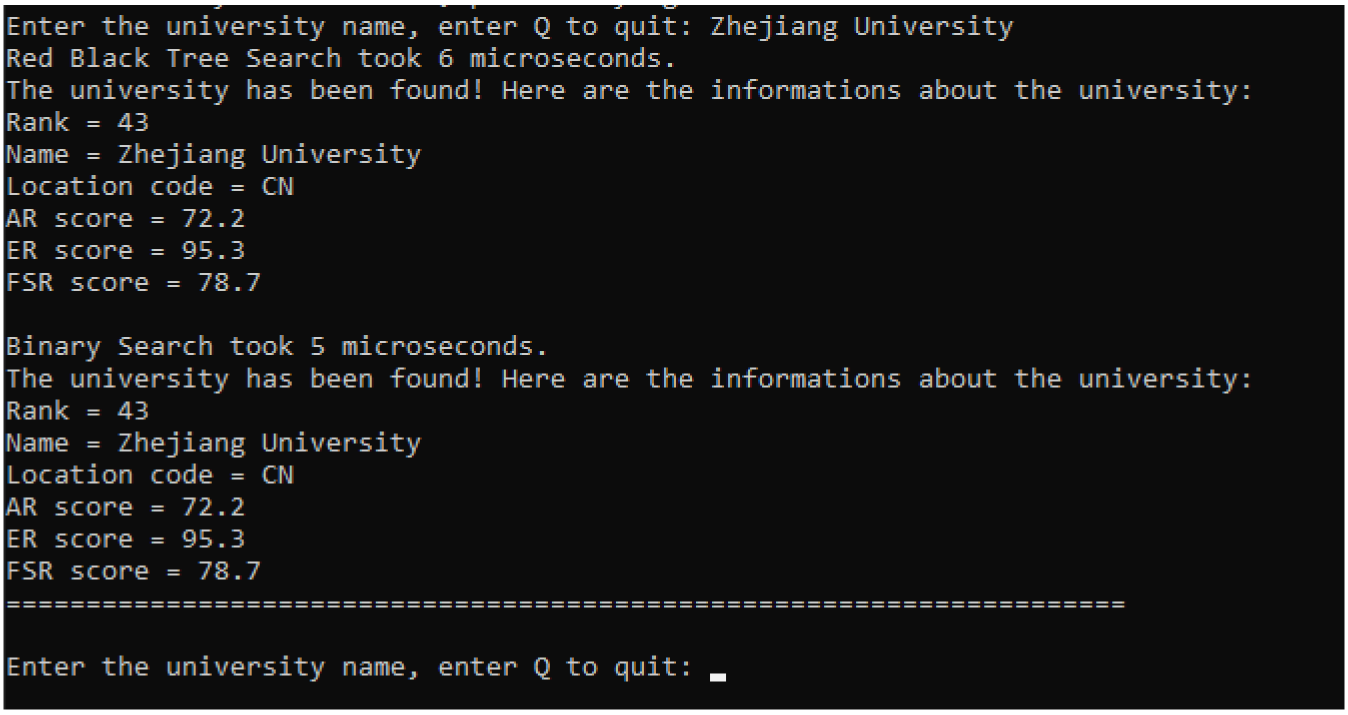
```
**********Search University Page***********
Enter the university name, enter Q to quit: AGH University of Science and Technology
Red Black Tree Search took 4 microseconds.
The university has been found! Here are the informations about the university:
Rank = 806
Name = AGH University of Science and Technology
Location code = PL
AR score = 8.8
ER score = 11.8
FSR score = 33.9

Binary Search took 4 microseconds.
The university has been found! Here are the informations about the university:
Rank = 806
Name = AGH University of Science and Technology
Location code = PL
AR score = 8.8
ER score = 11.8
FSR score = 33.9
======================================================================
```

Figure 5.6.1: Searching for university starting with "A"



```
Enter the university name, enter Q to quit: Lund University
Red Black Tree Search took 3 microseconds.
The university has been found! Here are the informations about the university:
Rank = 95
Name = Lund University
Location code = SE
AR score = 68.8
ER score = 50.9
FSR score = 51.3

Binary Search took 3 microseconds.
The university has been found! Here are the informations about the university:
Rank = 95
Name = Lund University
Location code = SE
AR score = 68.8
ER score = 50.9
FSR score = 51.3
======================================================================
```

Figure 5.6.2: Searching for university starting with "L"



```
Enter the university name, enter Q to quit: Zhejiang University
Red Black Tree Search took 6 microseconds.
The university has been found! Here are the informations about the university:
Rank = 43
Name = Zhejiang University
Location code = CN
AR score = 72.2
ER score = 95.3
FSR score = 78.7

Binary Search took 5 microseconds.
The university has been found! Here are the informations about the university:
Rank = 43
Name = Zhejiang University
Location code = CN
AR score = 72.2
ER score = 95.3
FSR score = 78.7
=======================================================================

Enter the university name, enter Q to quit: _
```

Figure 5.6.3: Searching for university starting with "Z"

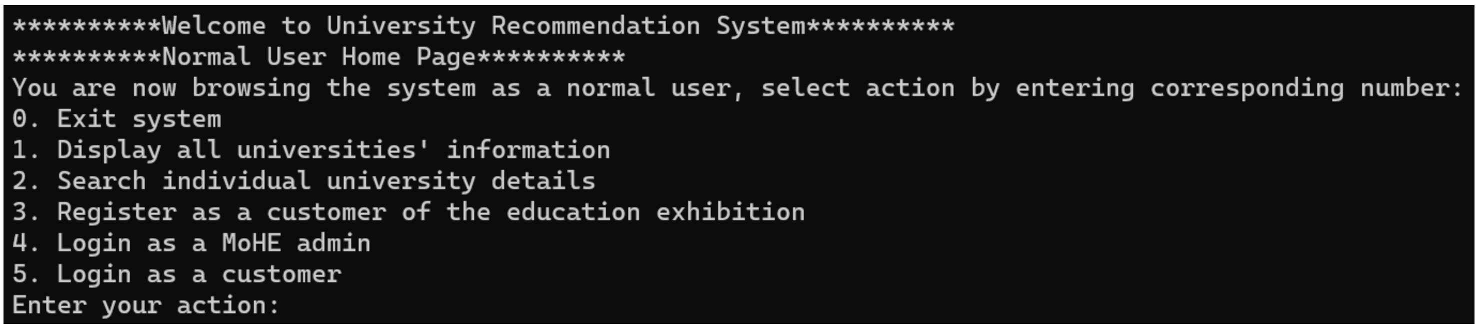
The three figures above show the execution times for binary search and red black tree search, and the execution times were almost the same. This is expected because theoretically binary search has the time complexity of *O(log N)* in average (Sharma, 2022), while search operation in red black tree is *O(log N)* as well (Gupta, n.d.).

It is difficult to provide a fair comparison between the two methods without considering system-specific factors like update rate and memory availability. Binary Search may provide somewhat better search performance in situations with static or seldom changed databases. Red Black Tree search, on the other hand, guarantees effective insertion and deletion operations while maintaining a balanced tree structure, making it a better fit for datasets that need frequent updates. As a result of the added complexity introduced by the colour property of each node, Red Black Trees also have a higher memory need. In conclusion, the unique needs of the university recommendation system will determine whether the Red Black Tree search or Binary Search algorithms are used.

6. System Input/Output Screenshot

6.1 Normal Users

Normal User Home Page

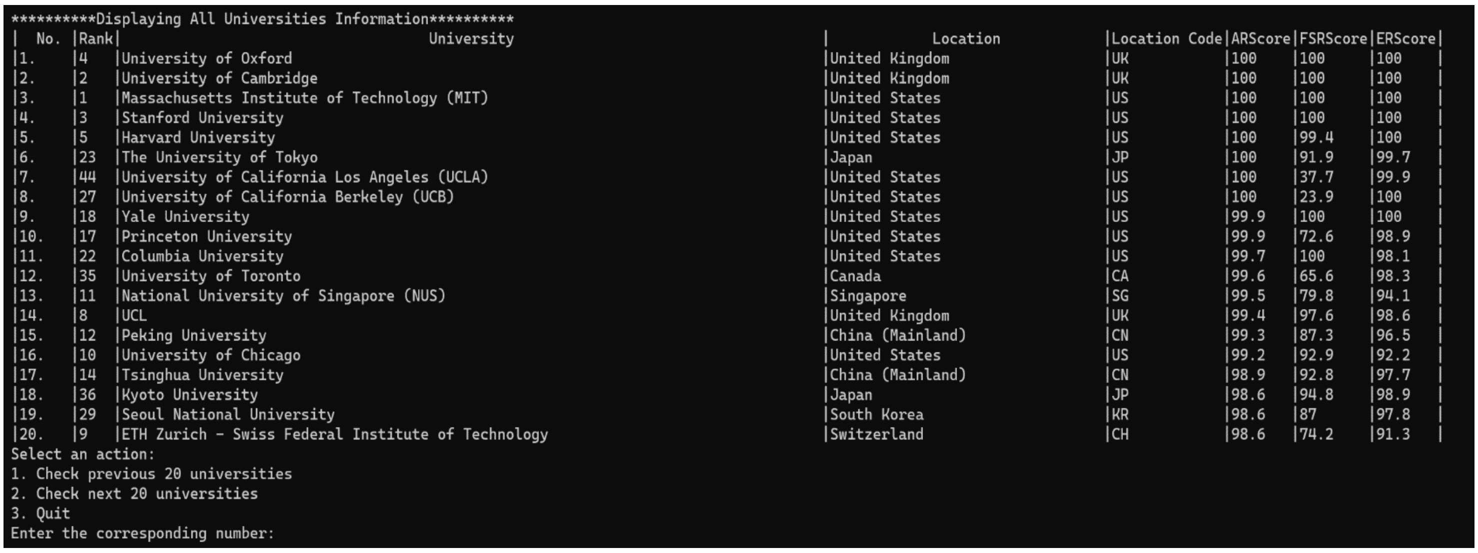


```
**********Welcome to University Recommendation System**********
**********Normal User Home Page**********
You are now browsing the system as a normal user, select action by entering corresponding number:
0. Exit system
1. Display all universities' information
2. Search individual university details
3. Register as a customer of the education exhibition
4. Login as a MoHE admin
5. Login as a customer
Enter your action:
```

Figure 6.1.1: Normal User Home Page

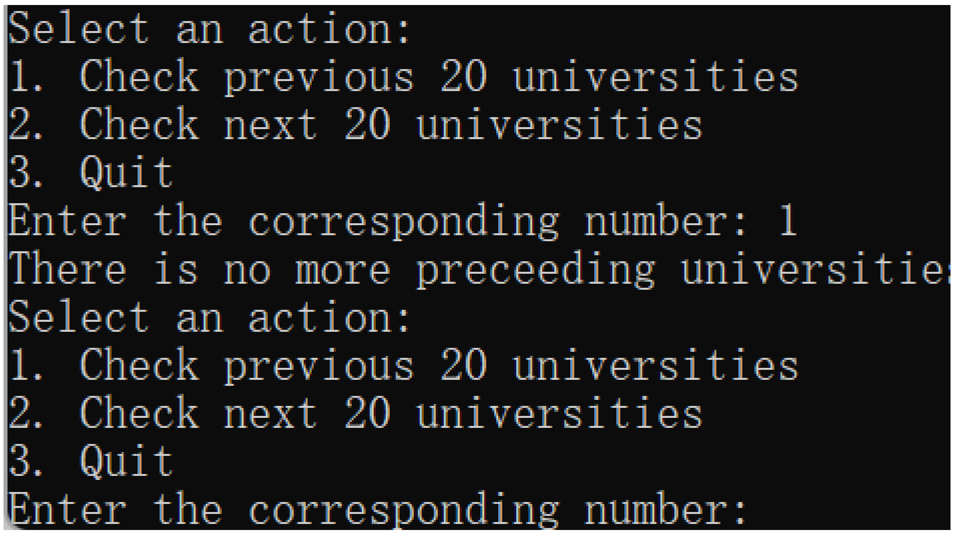
The figure above shows the initial home page for every user as normal user. Normal users are accessible to exit the system, display all universities' information, search a individual university's details, register as a customer of the system, login as admin of MoHE, and Login as a customer. The last two functions are only accessible if the user is an admin or existing customer. Users can access each function by entering the index number shown beside the function name.

Display all University Information Page



```
**********Displaying All Universities Information**********
|  No. |Rank|                                    University                                     |           Location             |Location Code|ARScore|FSRScore|ERScore|
|1.    |4   |University of Oxford                                                               |United Kingdom                  |UK           |100    |100     |100    |
|2.    |2   |University of Cambridge                                                            |United Kingdom                  |UK           |100    |100     |100    |
|3.    |1   |Massachusetts Institute of Technology (MIT)                                        |United States                   |US           |100    |100     |100    |
|4.    |3   |Stanford University                                                                |United States                   |US           |100    |100     |100    |
|5.    |5   |Harvard University                                                                 |United States                   |US           |100    |99.4    |100    |
|6.    |23  |The University of Tokyo                                                            |Japan                           |JP           |100    |91.9    |99.7   |
|7.    |44  |University of California Los Angeles (UCLA)                                        |United States                   |US           |100    |37.7    |99.9   |
|8.    |27  |University of California Berkeley (UCB)                                            |United States                   |US           |100    |23.9    |100    |
|9.    |18  |Yale University                                                                    |United States                   |US           |99.9   |100     |100    |
|10.   |17  |Princeton University                                                               |United States                   |US           |99.9   |72.6    |98.9   |
|11.   |22  |Columbia University                                                                |United States                   |US           |99.7   |100     |98.1   |
|12.   |35  |University of Toronto                                                              |Canada                          |CA           |99.6   |65.6    |98.3   |
|13.   |11  |National University of Singapore (NUS)                                             |Singapore                       |SG           |99.5   |79.8    |94.1   |
|14.   |8   |UCL                                                                                |United Kingdom                  |UK           |99.4   |97.6    |98.6   |
|15.   |12  |Peking University                                                                  |China (Mainland)                |CN           |99.3   |87.3    |96.5   |
|16.   |10  |University of Chicago                                                              |United States                   |US           |99.2   |92.9    |92.2   |
|17.   |14  |Tsinghua University                                                                |China (Mainland)                |CN           |98.9   |92.8    |97.7   |
|18.   |36  |Kyoto University                                                                   |Japan                           |JP           |98.6   |94.8    |98.9   |
|19.   |29  |Seoul National University                                                          |South Korea                     |KR           |98.6   |87      |97.8   |
|20.   |9   |ETH Zurich - Swiss Federal Institute of Technology                                 |Switzerland                     |CH           |98.6   |74.2    |91.3   |
Select an action:
1. Check previous 20 universities
2. Check next 20 universities
3. Quit
Enter the corresponding number:
```

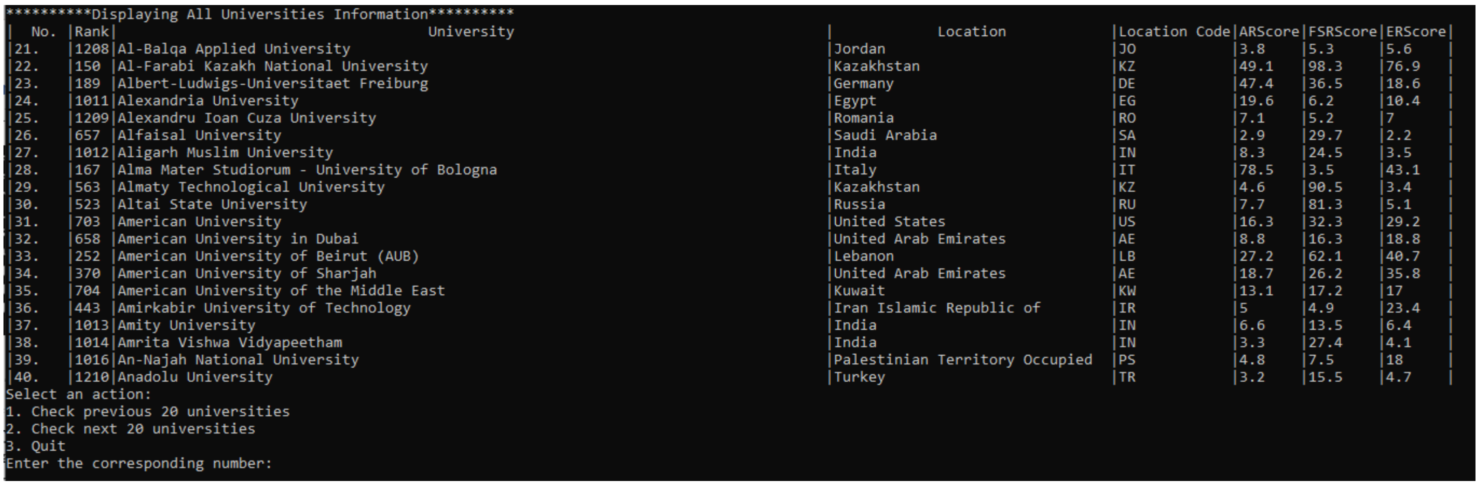
Figure 6.1.2: All universities's information page



```
Select an action:
1. Check previous 20 universities
2. Check next 20 universities
3. Quit
Enter the corresponding number: 1
There is no more preceeding universitie
Select an action:
1. Check previous 20 universities
2. Check next 20 universities
3. Quit
Enter the corresponding number:
```

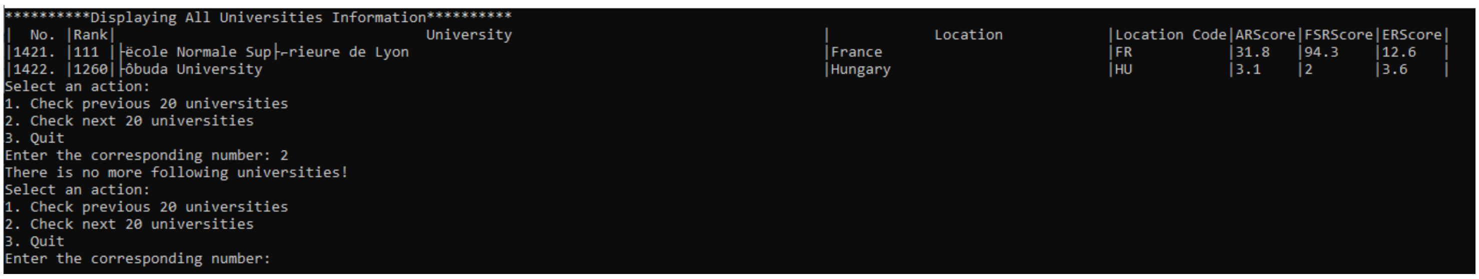
Figure 6.1.3: Warning message for invalid option

This system will direct to the page as shown in *Figure 6.1.2* when user enters option 1. This page will show the first 20 top universities with their rank, name, location, location code, AR score, FSR score, and ER score. Users will have 3 options in this page. Number 1 will show previous 20 universities, number 2 will show next 20 universities, and number 3 will exit the current page and redirect user to Normal User Home Page. However, if the user chooses option 1 while the system is already showing the first 20 universities, a warning message will be printed out for the user and the user needs to choose from the three options again as shown in *Figure 6.1.3*.



```
**********Displaying All Universities Information**********
|  No. |Rank|                  University                  |           Location            |Location Code|ARScore|FSRScore|ERScore|
|21.   |1208|Al-Balqa Applied University                   |Jordan                         |JO           |3.8    |5.3     |5.6    |
|22.   |150 |Al-Farabi Kazakh National University          |Kazakhstan                     |KZ           |49.1   |98.3    |76.9   |
|23.   |189 |Albert-Ludwigs-Universitaet Freiburg          |Germany                        |DE           |47.4   |36.5    |18.6   |
|24.   |1011|Alexandria University                         |Egypt                          |EG           |19.6   |6.2     |10.4   |
|25.   |1209|Alexandru Ioan Cuza University                |Romania                        |RO           |7.1    |5.2     |7      |
|26.   |657 |Alfaisal University                           |Saudi Arabia                   |SA           |2.9    |29.7    |2.2    |
|27.   |1012|Aligarh Muslim University                     |India                          |IN           |8.3    |24.5    |3.5    |
|28.   |167 |Alma Mater Studiorum - University of Bologna  |Italy                          |IT           |78.5   |3.5     |43.1   |
|29.   |563 |Almaty Technological University               |Kazakhstan                     |KZ           |4.6    |90.5    |3.4    |
|30.   |523 |Altai State University                        |Russia                         |RU           |7.7    |81.3    |5.1    |
|31.   |703 |American University                           |United States                  |US           |16.3   |32.3    |29.2   |
|32.   |658 |American University in Dubai                  |United Arab Emirates           |AE           |8.8    |16.3    |18.8   |
|33.   |252 |American University of Beirut (AUB)           |Lebanon                        |LB           |27.2   |62.1    |40.7   |
|34.   |370 |American University of Sharjah                |United Arab Emirates           |AE           |18.7   |26.2    |35.8   |
|35.   |704 |American University of the Middle East        |Kuwait                         |KW           |13.1   |17.2    |17     |
|36.   |443 |Amirkabir University of Technology            |Iran Islamic Republic of       |IR           |5      |4.9     |23.4   |
|37.   |1013|Amity University                              |India                          |IN           |6.6    |13.5    |6.4    |
|38.   |1014|Amrita Vishwa Vidyapeetham                    |India                          |IN           |3.3    |27.4    |4.1    |
|39.   |1016|An-Najah National University                  |Palestinian Territory Occupied |PS           |4.8    |7.5     |18     |
|40.   |1210|Anadolu University                            |Turkey                         |TR           |3.2    |15.5    |4.7    |
Select an action:
1. Check previous 20 universities
2. Check next 20 universities
3. Quit
Enter the corresponding number:
```

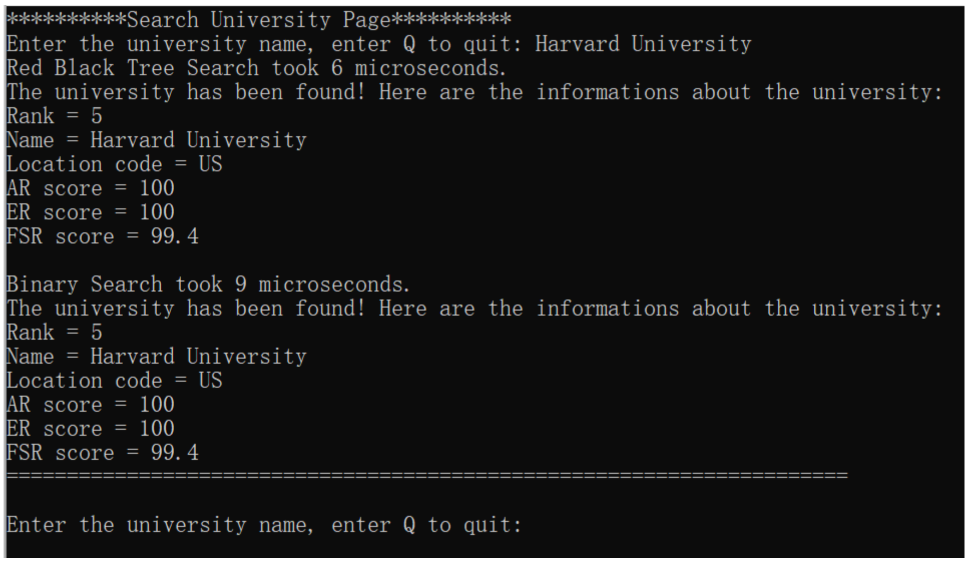
Figure 6.1.4: Showing next 20 universities after entering "2"



```
**********Displaying All Universities Information**********
|  No. |Rank|                  University                  |           Location            |Location Code|ARScore|FSRScore|ERScore|
|1421. |111 |├ëcole Normale Sup├-rieure de Lyon             |France                         |FR           |31.8   |94.3    |12.6   |
|1422. |1260|├ôbuda University                             |Hungary                        |HU           |3.1    |2       |3.6    |
Select an action:
1. Check previous 20 universities
2. Check next 20 universities
3. Quit
Enter the corresponding number: 2
There is no more following universities!
Select an action:
1. Check previous 20 universities
2. Check next 20 universities
3. Quit
Enter the corresponding number:
```

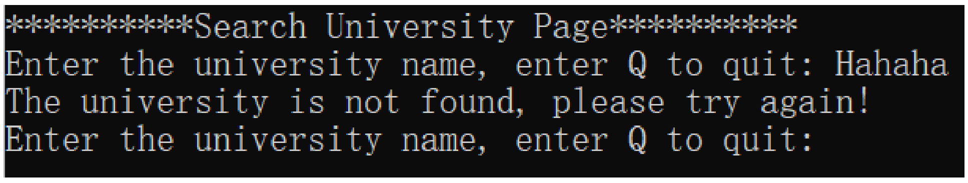
Figure 6.1.5: Showing warning if user persists entering "2" when there is no more university

Search University Page



```
**********Search University Page**********
Enter the university name, enter Q to quit: Harvard University
Red Black Tree Search took 6 microseconds.
The university has been found! Here are the informations about the university:
Rank = 5
Name = Harvard University
Location code = US
AR score = 100
ER score = 100
FSR score = 99.4

Binary Search took 9 microseconds.
The university has been found! Here are the informations about the university:
Rank = 5
Name = Harvard University
Location code = US
AR score = 100
ER score = 100
FSR score = 99.4
======================================================================

Enter the university name, enter Q to quit:
```

Figure 6.1.6: Search University Page

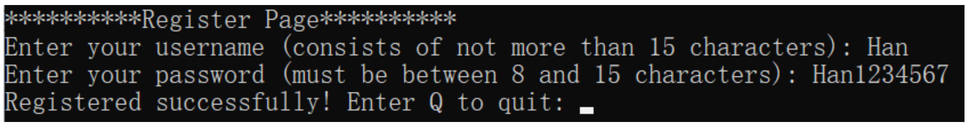


```
**********Search University Page**********
Enter the university name, enter Q to quit: Hahaha
The university is not found, please try again!
Enter the university name, enter Q to quit:
```

Figure 6.1.7: Warning message for unfound university

When the user chooses option 2, the user will be directed to this page as shown in *Figure 6.1.6*. At this page, users can enter the university name of their desired university. Two results will be generated and shown as two sorting methods were used. The time taken for each method will be shown in microseconds. University rank, name, AR score, ER score, and FSR score will be displayed in order. User can continue to search another university or enter capital letter "Q" to exit the current page. When the user did not enter the correct university name, or a name that is not found in the list. The system will prompt a warning message and ask the user to enter again as shown in *Figure 6.1.7*.

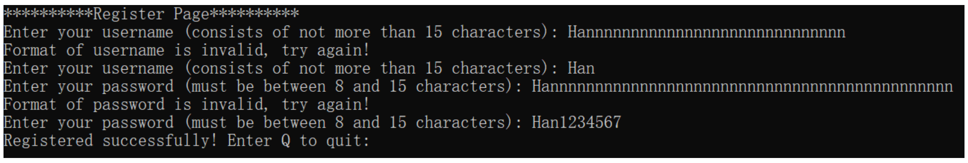
Register as Customer



```
**********Register Page**********
Enter your username (consists of not more than 15 characters): Han
Enter your password (must be between 8 and 15 characters): Han1234567
Registered successfully! Enter Q to quit:
```

Figure 6.1.8: User Registration Page

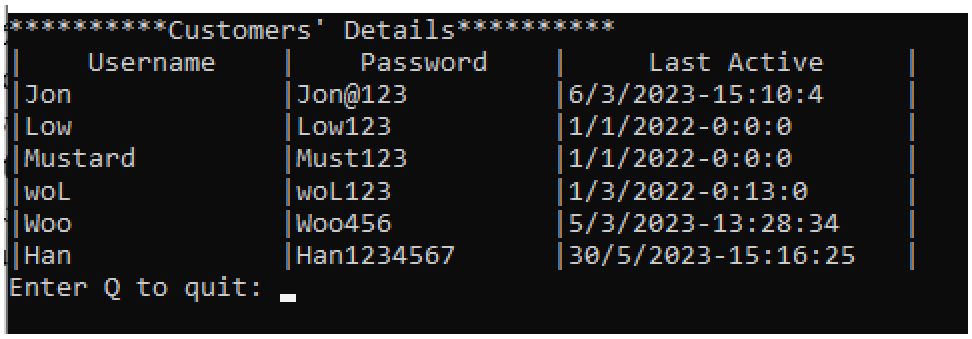
This page will be directed to the user when he chooses option 3 as shown in figure above. The system will ask the user to enter a username that may not contain more than 15 characters and a password that must be between 8 and 15 characters. Once requirements are met, the user can login as a customer and access all customers' functions. The newly created account for customer "Han" can be proven by *Figure 6.1.10*, and it is the screenshot when an admin uses his/her rights to print out the details of all current customers.



```
**********Register Page**********
Enter your username (consists of not more than 15 characters): Hannnnnnnnnnnnnnnnnnnnnnnnnnnnn
Format of username is invalid, try again!
Enter your username (consists of not more than 15 characters): Han
Enter your password (must be between 8 and 15 characters): Hannnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnn
Format of password is invalid, try again!
Enter your password (must be between 8 and 15 characters): Han1234567
Registered successfully! Enter Q to quit:
```

Figure 6.1.9: Warning messages for invalid username and password

Figure above shows the warning messages prompted when the user enters invalid username and password that did not reach the requirements set. Also, it shows how the system carries on when the user enters a valid username and password.

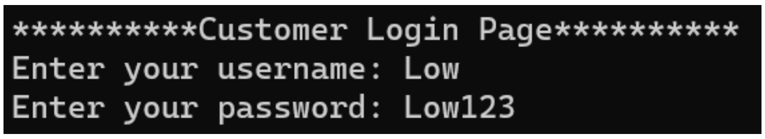


```
**********Customers' Details**********
|    Username      |     Password     |      Last Active      |
|Jon               |Jon@123           |6/3/2023-15:10:4       |
|Low               |Low123            |1/1/2022-0:0:0         |
|Mustard           |Must123           |1/1/2022-0:0:0         |
|woL               |woL123            |1/3/2022-0:13:0        |
|Woo               |Woo456            |5/3/2023-13:28:34      |
|Han               |Han1234567        |30/5/2023-15:16:25     |
Enter Q to quit:
```

Figure 6.1.10: New customer's account "Han" had been added

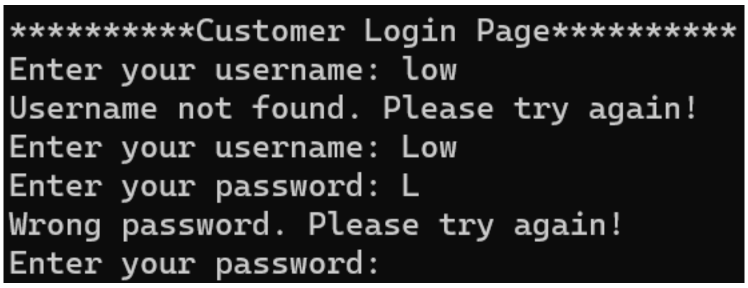
6.2 Customers

Customer login page:



```
**********Customer Login Page**********
Enter your username: Low
Enter your password: Low123
```

Figure 6.2.1: Customer Login Page with correct credentials

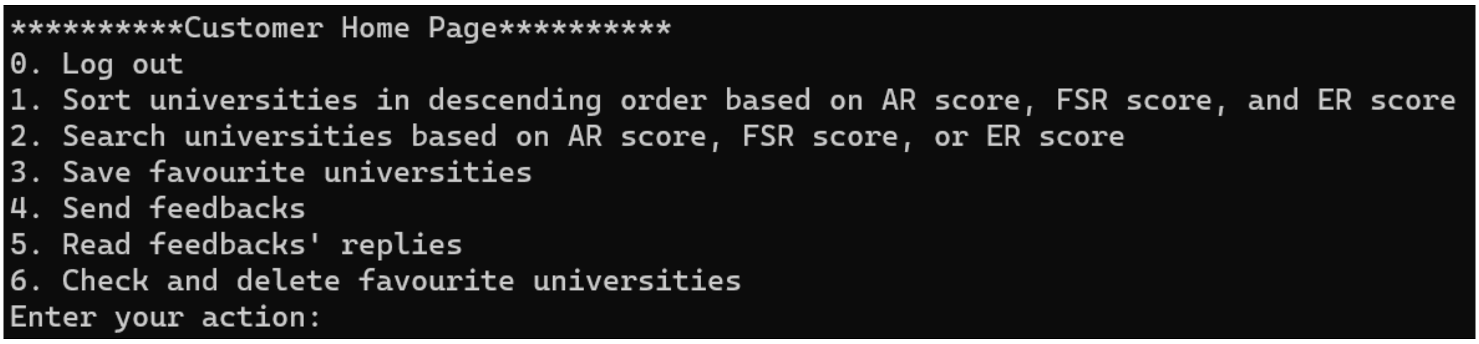


```
**********Customer Login Page**********
Enter your username: low
Username not found. Please try again!
Enter your username: Low
Enter your password: L
Wrong password. Please try again!
Enter your password:
```

Figure 6.2.2: Customer Login Page with wrong credentials

In *Figure 6.2.1*, we can observe the login page designed for customer access. To access the customer portal successfully, the username should be set as "Low," while the password should be "Low123." When the user enters these credentials correctly, they will be redirected to the customer home page. However, if the provided credentials are incorrect, an error message will be displayed, prompting the user to re-enter the correct username and password. This behavior is shown in *Figure 6.2.2*.

Customer home page:

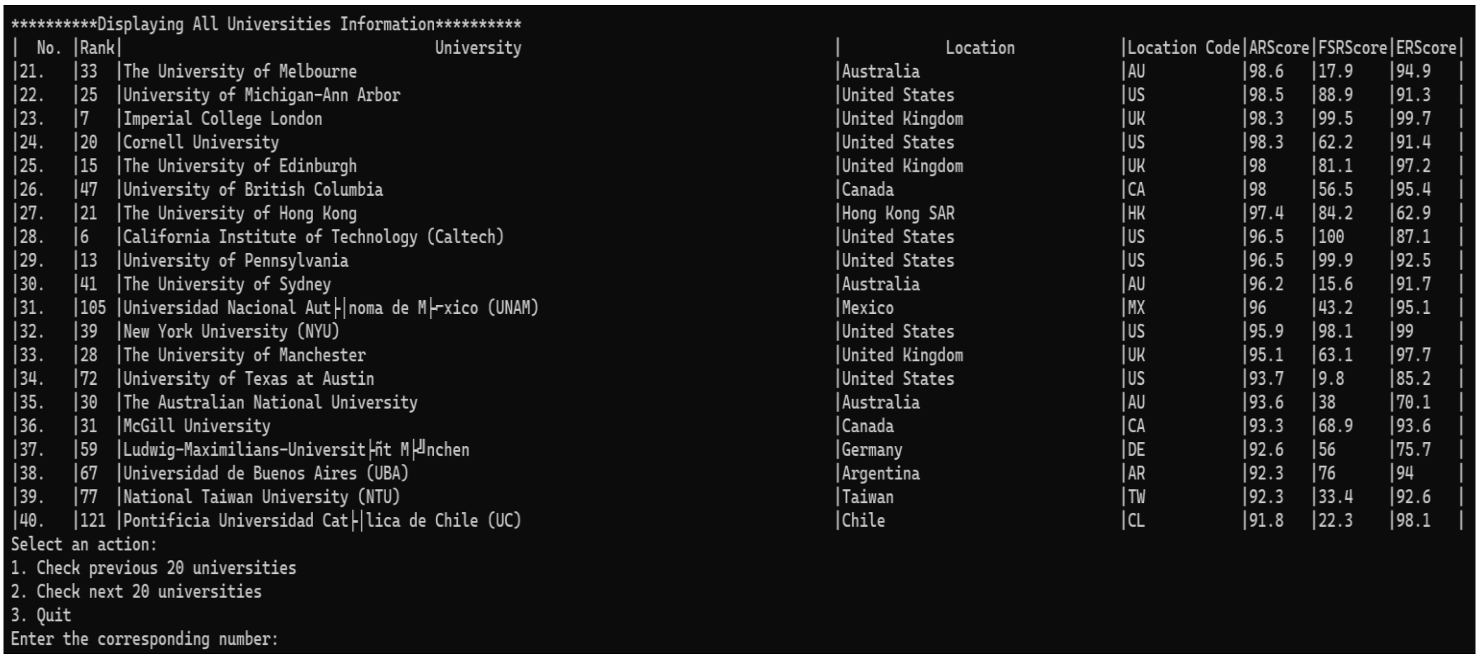


```
**********Customer Home Page**********
0. Log out
1. Sort universities in descending order based on AR score, FSR score, and ER score
2. Search universities based on AR score, FSR score, or ER score
3. Save favourite universities
4. Send feedbacks
5. Read feedbacks' replies
6. Check and delete favourite universities
Enter your action:
```

Figure 6.2.3: Customer Home Page

Figure 6.2.3 shows the customer home page once the user has entered the correct customer credentials. In this page, the customer has access to seven functions represented by numbers 0 to 6. These functions include logging out, sorting universities in descending order based on AR score, FSR score, and ER score, searching for universities based on AR score, FSR score, or ER score, saving favorite universities, sending feedback, reading feedback replies, and checking and deleting favorite universities. Upon entering the corresponding number that represents a specific function, the user will be directed to the respective page for that function. This provides the customer with a range of options and functionalities to interact with the system and manage their university-related activities.

Displaying universities in descending order base on AR, FSR and ER score:



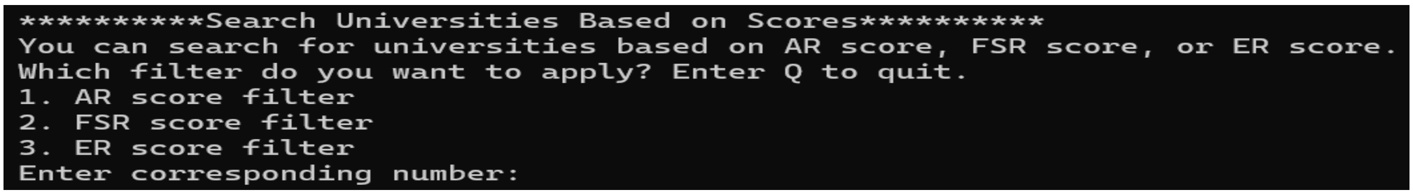
```
**********Displaying All Universities Information**********
|  No. |Rank|                                    University                                    |             Location            |Location Code|ARScore|FSRScore|ERScore|
|21.   |33  |The University of Melbourne                                                       |Australia                        |AU           |98.6   |17.9    |94.9   |
|22.   |25  |University of Michigan-Ann Arbor                                                  |United States                    |US           |98.5   |88.9    |91.3   |
|23.   |7   |Imperial College London                                                           |United Kingdom                   |UK           |98.3   |99.5    |99.7   |
|24.   |20  |Cornell University                                                                |United States                    |US           |98.3   |62.2    |91.4   |
|25.   |15  |The University of Edinburgh                                                       |United Kingdom                   |UK           |98     |81.1    |97.2   |
|26.   |47  |University of British Columbia                                                    |Canada                           |CA           |98     |56.5    |95.4   |
|27.   |21  |The University of Hong Kong                                                       |Hong Kong SAR                    |HK           |97.4   |84.2    |62.9   |
|28.   |6   |California Institute of Technology (Caltech)                                      |United States                    |US           |96.5   |100     |87.1   |
|29.   |13  |University of Pennsylvania                                                        |United States                    |US           |96.5   |99.9    |92.5   |
|30.   |41  |The University of Sydney                                                          |Australia                        |AU           |96.2   |15.6    |91.7   |
|31.   |105 |Universidad Nacional Aut├│noma de M├⌐xico (UNAM)                                    |Mexico                           |MX           |96     |43.2    |95.1   |
|32.   |39  |New York University (NYU)                                                         |United States                    |US           |95.9   |98.1    |99     |
|33.   |28  |The University of Manchester                                                      |United Kingdom                   |UK           |95.1   |63.1    |97.7   |
|34.   |72  |University of Texas at Austin                                                     |United States                    |US           |93.7   |9.8     |85.2   |
|35.   |30  |The Australian National University                                                |Australia                        |AU           |93.6   |38      |70.1   |
|36.   |31  |McGill University                                                                 |Canada                           |CA           |93.3   |68.9    |93.6   |
|37.   |59  |Ludwig-Maximilians-Universit├ñt M├╝nchen                                           |Germany                          |DE           |92.6   |56      |75.7   |
|38.   |67  |Universidad de Buenos Aires (UBA)                                                 |Argentina                        |AR           |92.3   |76      |94     |
|39.   |77  |National Taiwan University (NTU)                                                  |Taiwan                           |TW           |92.3   |33.4    |92.6   |
|40.   |121 |Pontificia Universidad Cat├│lica de Chile (UC)                                     |Chile                            |CL           |91.8   |22.3    |98.1   |
Select an action:
1. Check previous 20 universities
2. Check next 20 universities
3. Quit
Enter the corresponding number:
```

Figure 6.2.4: output of sorting universities in descending order base on AR, FSR, ER score

When the customer enters "1" on the customer home page, the system will redirect them to the "Sort Universities" page. In this page, the universities will be sorted in descending order based on three scores: AR score, FSR score, and ER score. The university details, including relevant information such as rank, university name, location, location code will be displayed for the customer to view. On the "Sort Universities" page, the customer will have the option to perform three actions:

1. Check previous 20 universities: This action allows the customer to view the details of the previous 20 universities in the sorted list. By clicking this option, the customer can navigate through the list and explore the universities ranked higher based on the scores mentioned earlier.
2. Check next 20 universities: This action enables the customer to view the details of the next 20 universities in the sorted list. Clicking this option allows the customer to browse through the list and discover universities ranked lower based on the scores provided.
3. Quit and go back to the customer home page: This action allows the customer to exit the sorting function and return to the customer home page, providing them with the opportunity to explore other functionalities and features available in the customer portal.

Searching universities base on AR, FSR and ER score:



```
**********Search Universities Based on Scores**********
You can search for universities based on AR score, FSR score, or ER score.
Which filter do you want to apply? Enter Q to quit.
1. AR score filter
2. FSR score filter
3. ER score filter
Enter corresponding number:
```

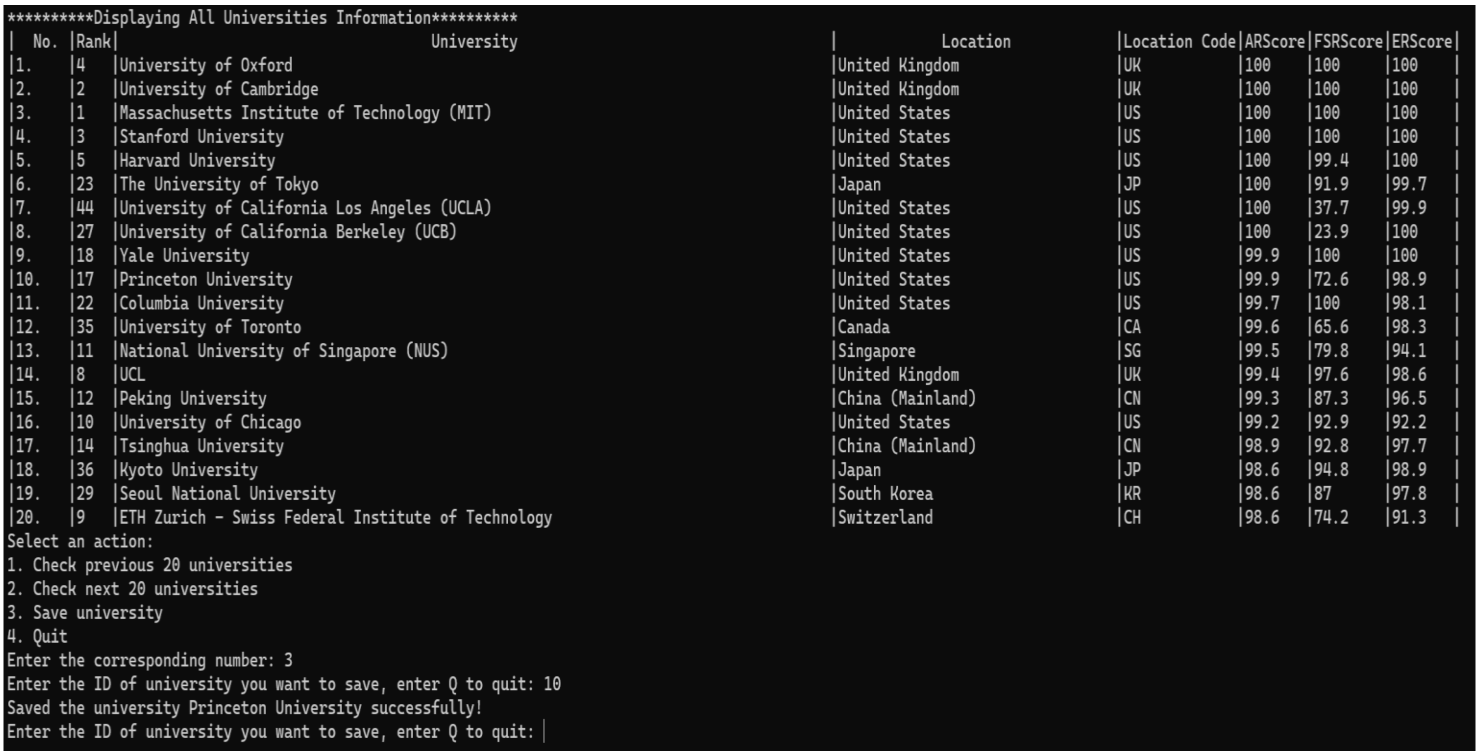
Figure 6.2.5: output of searching universities base on AR, FSR, or ER score

When the customer enters "2" on the home page, they will be directed to the search universities function based on scores. A message prompt, as shown in *Figure 6.2.5*, will be displayed, guiding the customer through the process. In the search universities function, the customer can perform three actions:

1. AR score filter: This action allows the customer to filter universities based on their AR score. By selecting this option, the customer can narrow down the search results to universities with a specific AR score range.
2. FSR score filter: This action enables the customer to filter universities based on their FSR score. By choosing this option, the customer can refine the search results to universities within a particular FSR score range.
3. ER score filter: This action allows the customer to filter universities based on their ER score. By opting for this option, the customer can focus the search results on universities falling within a specific ER score range.

To quit from the search university's function and return to the customer home page, the customer needs to type capital “Q”.

Save favorite universities:

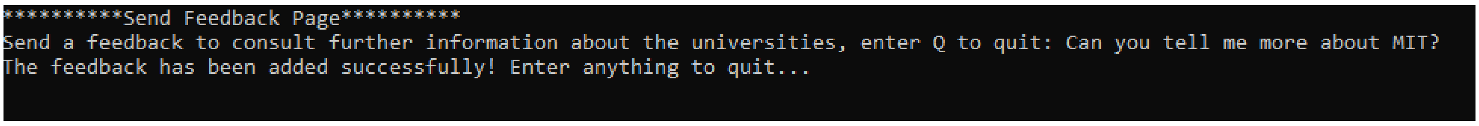


```
**********Displaying All Universities Information**********
|  No. |Rank|                      University                              |          Location          |Location Code|ARScore|FSRScore|ERScore|
|1.    |4   |University of Oxford                                          |United Kingdom              |UK           |100    |100     |100    |
|2.    |2   |University of Cambridge                                       |United Kingdom              |UK           |100    |100     |100    |
|3.    |1   |Massachusetts Institute of Technology (MIT)                   |United States               |US           |100    |100     |100    |
|4.    |3   |Stanford University                                           |United States               |US           |100    |100     |100    |
|5.    |5   |Harvard University                                            |United States               |US           |100    |99.4    |100    |
|6.    |23  |The University of Tokyo                                       |Japan                       |JP           |100    |91.9    |99.7   |
|7.    |44  |University of California Los Angeles (UCLA)                   |United States               |US           |100    |37.7    |99.9   |
|8.    |27  |University of California Berkeley (UCB)                       |United States               |US           |100    |23.9    |100    |
|9.    |18  |Yale University                                               |United States               |US           |99.9   |100     |100    |
|10.   |17  |Princeton University                                          |United States               |US           |99.9   |72.6    |98.9   |
|11.   |22  |Columbia University                                           |United States               |US           |99.7   |100     |98.1   |
|12.   |35  |University of Toronto                                         |Canada                      |CA           |99.6   |65.6    |98.3   |
|13.   |11  |National University of Singapore (NUS)                        |Singapore                   |SG           |99.5   |79.8    |94.1   |
|14.   |8   |UCL                                                           |United Kingdom              |UK           |99.4   |97.6    |98.6   |
|15.   |12  |Peking University                                             |China (Mainland)            |CN           |99.3   |87.3    |96.5   |
|16.   |10  |University of Chicago                                         |United States               |US           |99.2   |92.9    |92.2   |
|17.   |14  |Tsinghua University                                           |China (Mainland)            |CN           |98.9   |92.8    |97.7   |
|18.   |36  |Kyoto University                                              |Japan                       |JP           |98.6   |94.8    |98.9   |
|19.   |29  |Seoul National University                                     |South Korea                 |KR           |98.6   |87      |97.8   |
|20.   |9   |ETH Zurich - Swiss Federal Institute of Technology            |Switzerland                 |CH           |98.6   |74.2    |91.3   |
Select an action:
1. Check previous 20 universities
2. Check next 20 universities
3. Save university
4. Quit
Enter the corresponding number: 3
Enter the ID of university you want to save, enter Q to quit: 10
Saved the university Princeton University successfully!
Enter the ID of university you want to save, enter Q to quit:
```

Figure 6.2.6: output of saving favorite universities

When the customer enters "3" on the customer home page, they will be directed to the "Save Favorite Universities" page. This page provides an opportunity for the customer to explore and review the details of different universities, assisting them in determining which universities they would like to save as favorites. To save a university as a favorite, the customer needs to enter the corresponding number or ID associated with the university. This action indicates the customer's interest in keeping that university for future reference. By saving universities as favorites, the customer can easily access and retrieve the information of their preferred universities later. This feature enhances convenience and allows customers to organize their university choices efficiently. To access their list of favorite universities, the customer can enter "6" on the customer home page. This action will redirect them to a dedicated page or function where their saved universities will be displayed. Here, the customer can conveniently view the details of their favorite universities and make informed decisions based on their preferences.

Send feedback:

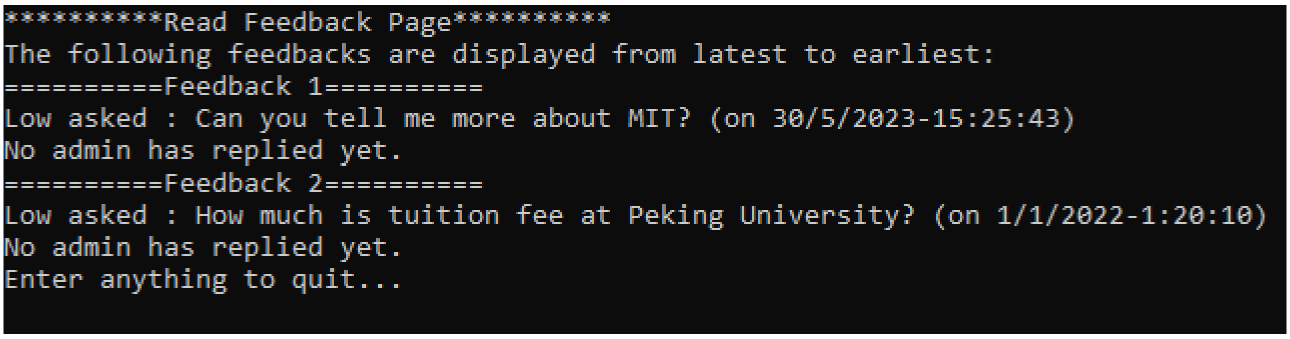


```
**********Send Feedback Page**********
Send a feedback to consult further information about the universities, enter Q to quit: Can you tell me more about MIT?
The feedback has been added successfully! Enter anything to quit...
```

Figure 6.2.7: output of sending feedback

When the customer enters "4" on the customer home page, they will be directed to the "Send Feedback" page. This page allows the customer to send their feedback or inquiries regarding the universities and any other relevant information they wish to communicate. The customer can type their message in the designated input area to express their thoughts, ask questions, or seek additional information. To exit the "Send Feedback" page and return to the customer home page, the customer can simply type capital "Q" to quit the function.

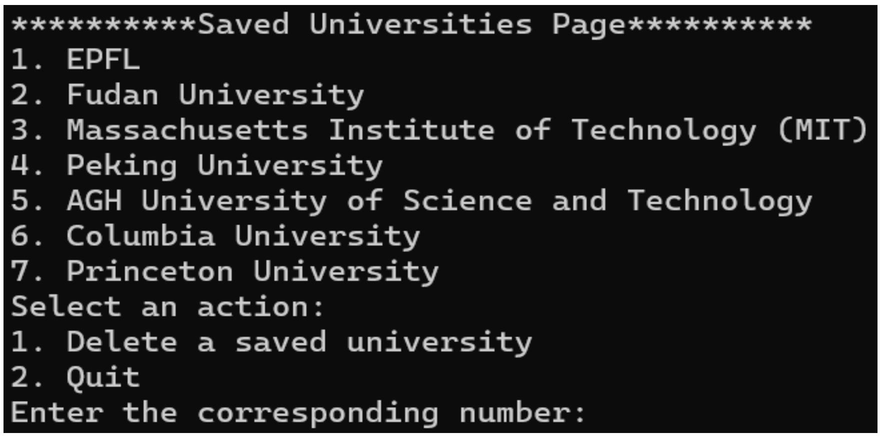
Read feedback replies:



```
**********Read Feedback Page**********
The following feedbacks are displayed from latest to earliest:
=========Feedback 1=========
Low asked : Can you tell me more about MIT? (on 30/5/2023-15:25:43)
No admin has replied yet.
=========Feedback 2=========
Low asked : How much is tuition fee at Peking University? (on 1/1/2022-1:20:10)
No admin has replied yet.
Enter anything to quit...
```

Figure 6.2.8: output of reading feedback replies

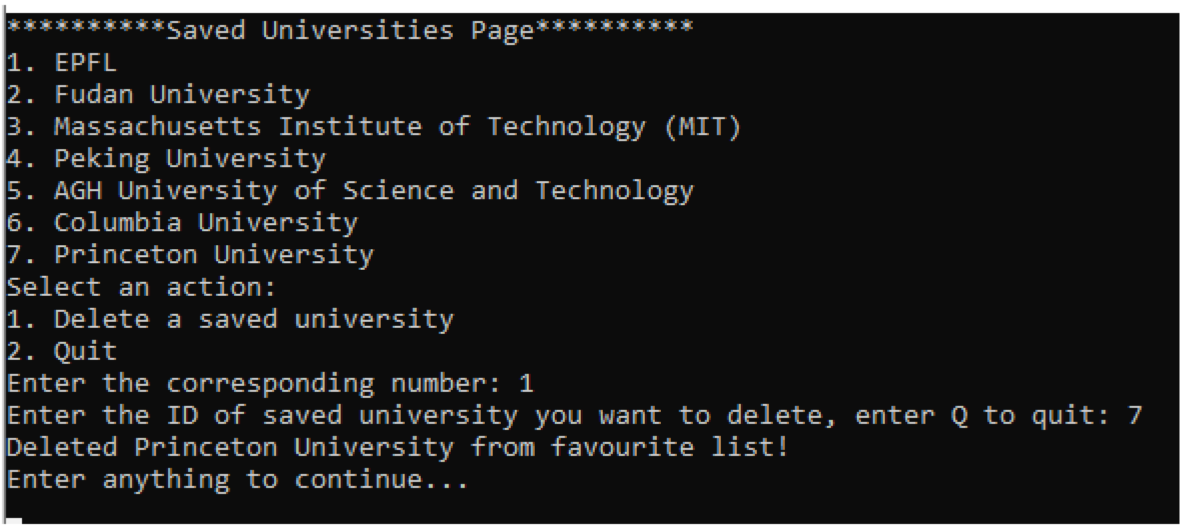
When the customer enters "5" on the customer home page, they will be redirected to the "Read Feedback" page. This page provides an interface for customers to view and read their feedback that has been replied to by the admin. The feedback is displayed starting from the latest to the earliest (To prove this, *Figure 6.2.8* shows that the feedback 1 is the “Can you tell me more about MIT?” which was added in *Figure 6.2.7*). By accessing the "Read Feedback" page, customers can stay updated on the status of their feedback and view any responses or information provided by the admin. To exit the "Read Feedback" page and return to the customer home page, the customer can simply type anything to quit the function. This action allows the customer to navigate back to the main customer interface and continue using other functionalities provided by the customer portal.



```
**********Saved Universities Page**********
1. EPFL
2. Fudan University
3. Massachusetts Institute of Technology (MIT)
4. Peking University
5. AGH University of Science and Technology
6. Columbia University
7. Princeton University
Select an action:
1. Delete a saved university
2. Quit
Enter the corresponding number:
```

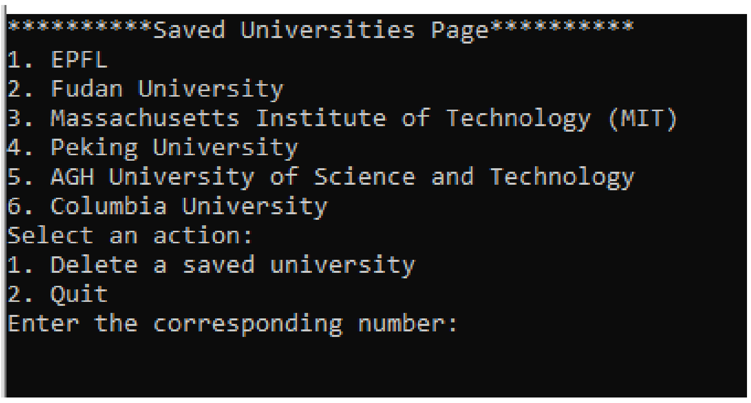
Figure 6.2.9: output of viewing or deleting saved universities

When the customer enters "6" on the customer home page, they will be directed to the "Saved Universities" page. This page displays the list of universities that the customer has previously saved during their interaction with the system, typically using option number "3" to save favorite universities as shown in *Figure 6.2.6*. Additionally, the customer will have the option to delete unnecessary or unwanted universities from their saved list. This feature enables the customer to manage and maintain a curated collection of preferred universities based on their changing preferences or needs. To delete a university from the saved list, the customer can select the corresponding option or follow the provided instructions on the "Saved Universities" page. To exit the "Saved Universities" page and return to the customer home page, the customer needs to enter number "2" to quit the function. This action will redirect them back to the main customer interface, where they can explore other functionalities and features available in the customer portal.



```
**********Saved Universities Page**********
1. EPFL
2. Fudan University
3. Massachusetts Institute of Technology (MIT)
4. Peking University
5. AGH University of Science and Technology
6. Columbia University
7. Princeton University
Select an action:
1. Delete a saved university
2. Quit
Enter the corresponding number: 1
Enter the ID of saved university you want to delete, enter Q to quit: 7
Deleted Princeton University from favourite list!
Enter anything to continue...
```

Figure 6.2.10: Deleting the Princeton University which was saved before

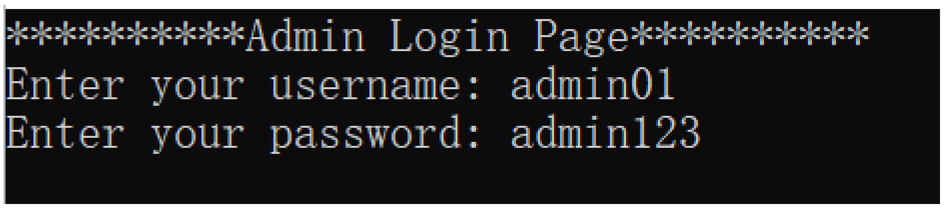


```
**********Saved Universities Page**********
1. EPFL
2. Fudan University
3. Massachusetts Institute of Technology (MIT)
4. Peking University
5. AGH University of Science and Technology
6. Columbia University
Select an action:
1. Delete a saved university
2. Quit
Enter the corresponding number:
```

Figure 6.2.11: Favourite List After Deleting Princeton University

6.3 Admins

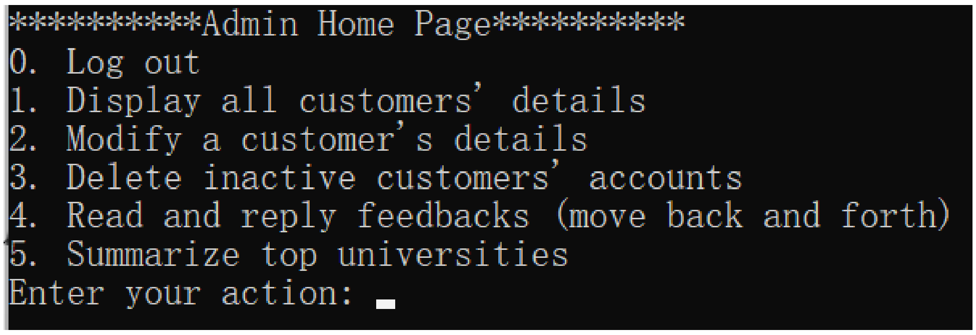
Admin Login Page and Home Page:



```
**********Admin Login Page**********
Enter your username: admin01
Enter your password: admin123
```

Figure 6.3.1: Admin Login Page with correct credentials

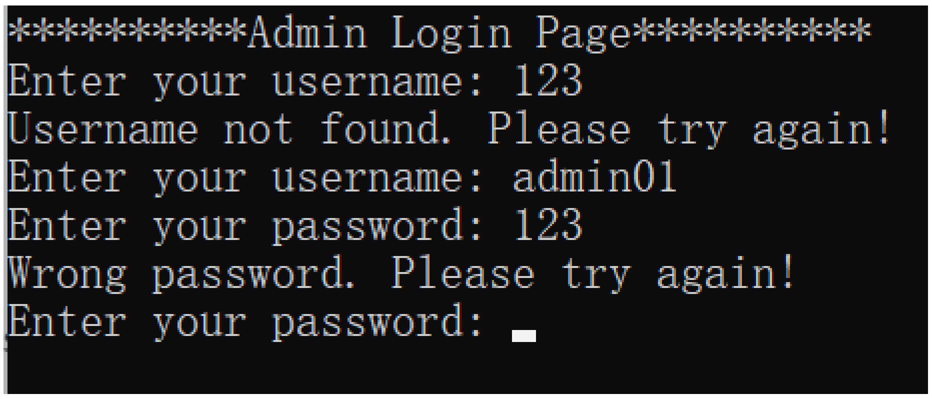
Figure 6.3.1 shows the Admin Login Page with correct credentials. In this case, “admin01” is the username and password “admin123”. With this credential, the user will be directed to Admin Home Page, otherwise there will be error output and ask user to re-enter the credentials.



```
**********Admin Home Page**********
0. Log out
1. Display all customers’ details
2. Modify a customer’s details
3. Delete inactive customers’ accounts
4. Read and reply feedbacks (move back and forth)
5. Summarize top universities
Enter your action: _
```

Figure 6.3.2: Admin Home Page

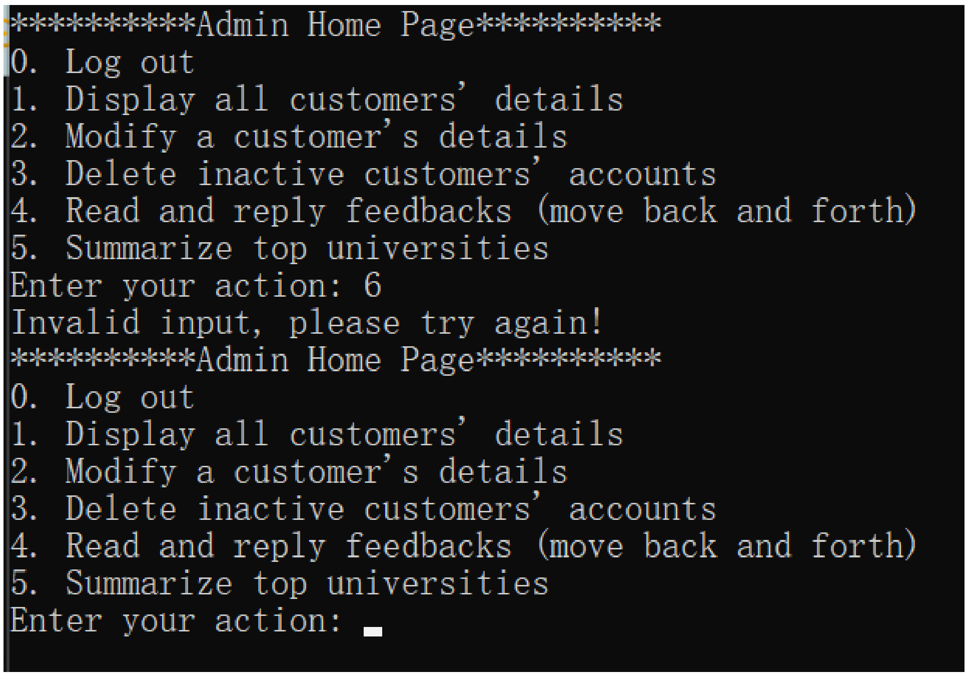
Figure above shows the Admin Home Page once the user has entered the correct admin credentials. In this page, admin has 6 functions from 0 to 5. Such as log out, display all customer’s details, modify customer’s details, delete inactive customer accounts, read and reply feedback, and summarize top universities. The user will be directed to each function’s page after entering the corresponding number that represents the function.



```
**********Admin Login Page**********
Enter your username: 123
Username not found. Please try again!
Enter your username: admin01
Enter your password: 123
Wrong password. Please try again!
Enter your password: _
```

Figure 6.3.3: Invalid admin username and password output

When the user tries to enter any invalid admin username and password that is not stored in the system, a warning message will be displayed for each field as shown in the figure above. The system will only continue to the next step when the user enters the correct admin credentials that are stored in the system.

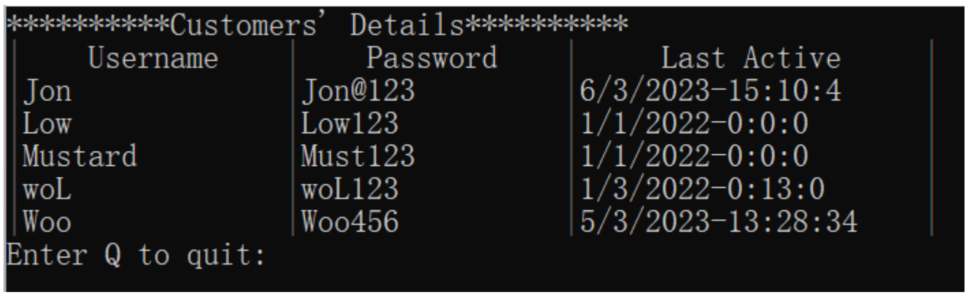


```
**********Admin Home Page**********
0. Log out
1. Display all customers' details
2. Modify a customer's details
3. Delete inactive customers' accounts
4. Read and reply feedbacks (move back and forth)
5. Summarize top universities
Enter your action: 6
Invalid input, please try again!
**********Admin Home Page**********
0. Log out
1. Display all customers' details
2. Modify a customer's details
3. Delete inactive customers' accounts
4. Read and reply feedbacks (move back and forth)
5. Summarize top universities
Enter your action:
```

Figure 6.3.4: Invalid option entered at Admin Home Page

While the admin is at Admin Home Page, and he enters an option that is not shown at the page, and warning message will be outputted telling admin to enter a valid input and a new Admin Home Page will be generated and ask the admin to enter his desired action again. A visual image is shown in the figure above.

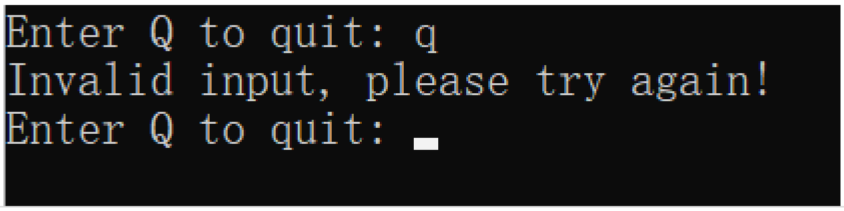
Display All User's Details



```
**********Customers' Details**********
|    Username        |    Password        |     Last Active         |
| Jon                | Jon@123            | 6/3/2023-15:10:4        |
| Low                | Low123             | 1/1/2022-0:0:0          |
| Mustard            | Must123            | 1/1/2022-0:0:0          |
| woL                | woL123             | 1/3/2022-0:13:0         |
| Woo                | Woo456             | 5/3/2023-13:28:34       |
Enter Q to quit:
```

Figure 6.3.5: Table output of every customer's detail

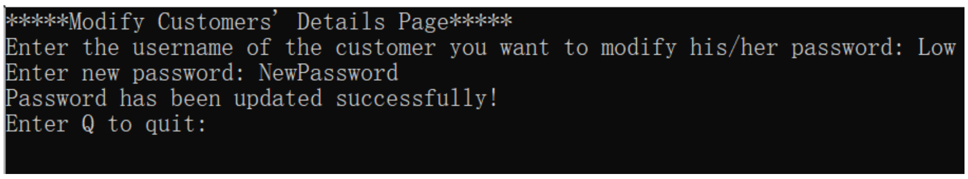
When the admin enters "1" on the admin home page, the system will redirect the user to the Customer Details Page. In this page as shown in the figure above, every registered customer username, password, and their last active time will be shown in a table form for admin to read the information easily and clearly. And once the admin is done looking for customers details, he/she can enter "Q" to exit this page and go back to admin home page. This command is case sensitive, so only capital Q will be recognized by the system. Otherwise, the system will output warning message and will ask admin to enter again until capital letter Q is entered. A demonstration screenshot of this process is shown below.



```
Enter Q to quit: q
Invalid input, please try again!
Enter Q to quit:
```

Figure 6.3.6: Warning message for invalid input of exiting the page

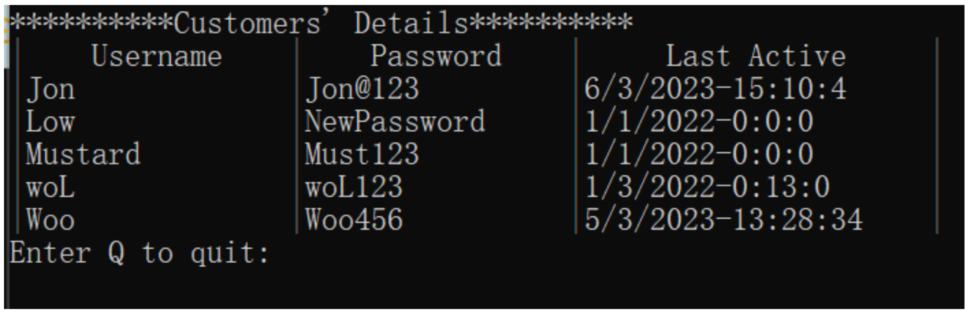
Modify A Customer's Details



```
*****Modify Customers' Details Page*****
Enter the username of the customer you want to modify his/her password: Low
Enter new password: NewPassword
Password has been updated successfully!
Enter Q to quit:
```

Figure 6.3.7: Password changing for "Low"

For this function, admin can enter a customer's username and reset the current customer's password. This page will be redirected for admin once he/she enters number "2" at admin home page. As shown in figure above, the system will ask for customer's username to check if the searched customer username exists in the system. If found, the system will ask again for the new password, any output will be accepted. After these two processes, the system will ask admin to enter "Q" to exit the current page and go back to admin home page.

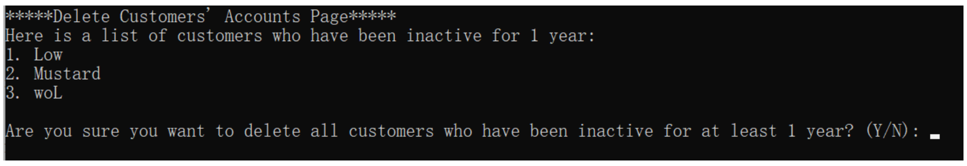


```
**********Customers' Details**********
|    Username        |    Password        |     Last Active      |
| Jon                | Jon@123            | 6/3/2023-15:10:4     |
| Low                | NewPassword        | 1/1/2022-0:0:0       |
| Mustard            | Must123            | 1/1/2022-0:0:0       |
| woL                | woL123             | 1/3/2022-0:13:0      |
| Woo                | Woo456             | 5/3/2023-13:28:34    |
Enter Q to quit:
```

Figure 6.3.8: New password set for "Low"

As seen from figure above, the password of the customer "Low" has changed to "NewPassword" successfully. Admins can change every existing customer's password using their username. And customers can only login with the new changed password.

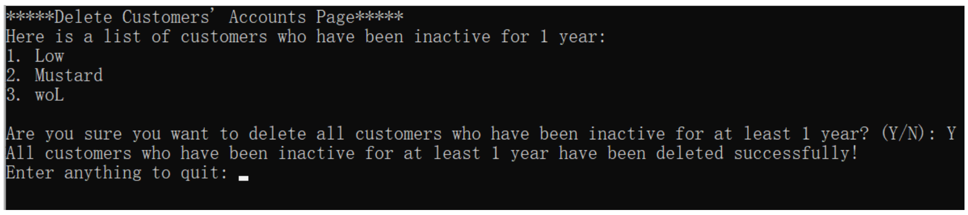
Delete Inactive Customers' Accounts Page



```
*****Delete Customers' Accounts Page*****
Here is a list of customers who have been inactive for 1 year:
1. Low
2. Mustard
3. woL

Are you sure you want to delete all customers who have been inactive for at least 1 year? (Y/N): 
```

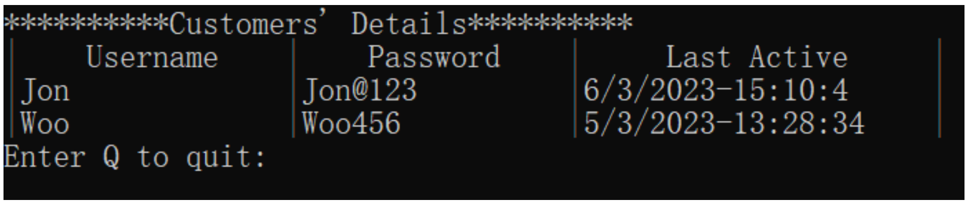
Figure 6.3.9: Delete all inactive customer accounts page

This function will display all customer accounts that have been inactive for more than 1 year. This standard is set to avoid deleting active accounts. As shown in figure above, there are three customer accounts that have not been active more than 1 year and their username is displayed on the screen. Admin needs to enter capital letter "Y" or "N" to decide whether to delete all inactive accounts or not to. When admin enters "Y", all inactive customers' accounts will be deleted from the system and leave the remaining active customer accounts and a successful message will be prompt to the admin as shown in the figure below. As if admins decide not to delete those inactive accounts, the system will redirect admin back to the Admin Home Page.



```
*****Delete Customers' Accounts Page*****
Here is a list of customers who have been inactive for 1 year:
1. Low
2. Mustard
3. woL

Are you sure you want to delete all customers who have been inactive for at least 1 year? (Y/N): Y
All customers who have been inactive for at least 1 year have been deleted successfully!
Enter anything to quit:
```

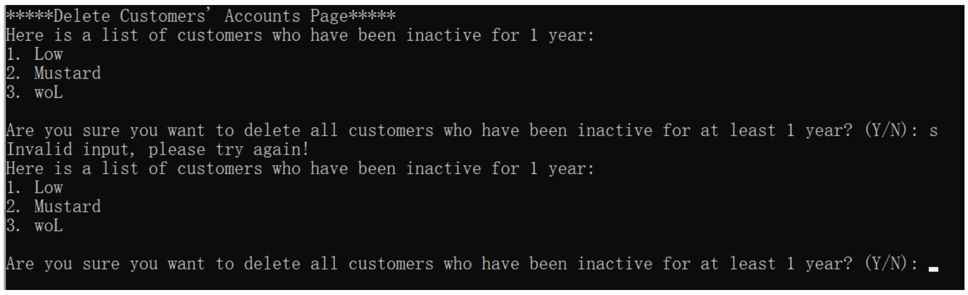
Figure 6.3.10: Inactive customers' accounts deleted successfully



```
**********Customers' Details**********
|    Username          |    Password          |    Last Active        |
| Jon                  |Jon@123               |6/3/2023-15:10:4       |
| Woo                  |Woo456                |5/3/2023-13:28:34      |
Enter Q to quit:
```

Figure 6.3.11: Customers' details page after deleted inactive accounts

Figure 5.3.11 shows the Customers' Details Page after all inactive accounts have been deleted by admin successfully. Comparing this figure with figure 5.3.6 can demonstrate the results of this in a clearer way. The remaining customers' accounts can only be considered as inactive after 1 year of the last active time, which are 6/3/2024 and 5/3/2024 respectively.

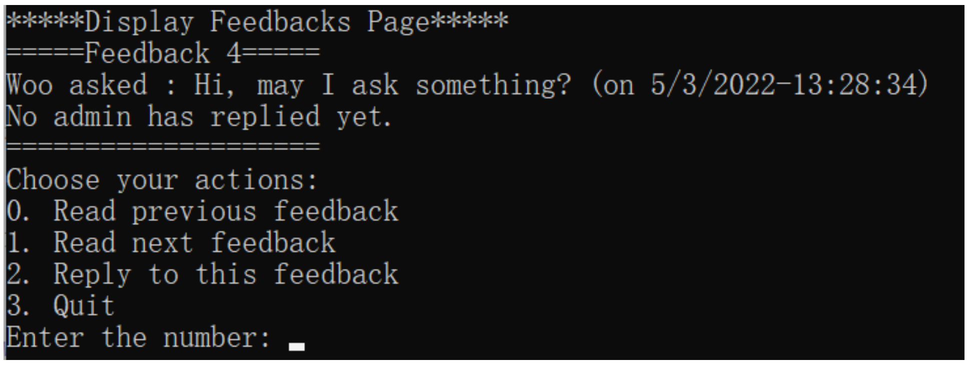


```
*****Delete Customers' Accounts Page*****
Here is a list of customers who have been inactive for 1 year:
1. Low
2. Mustard
3. woL

Are you sure you want to delete all customers who have been inactive for at least 1 year? (Y/N): s
Invalid input, please try again!
Here is a list of customers who have been inactive for 1 year:
1. Low
2. Mustard
3. woL

Are you sure you want to delete all customers who have been inactive for at least 1 year? (Y/N):
```

Figure 6.3.12: Invalid input at deleting customers' account

As mentioned in the first paragraph of this section, the system will only operate 2 options, "Y" or "N". The system is case sensitive, so only capitals of these two letters will be accepted as a valid input. Any other input will not be accepted, and warning message will be outputted, and admins need to choose again in the newly generated page as shown in the figure above.

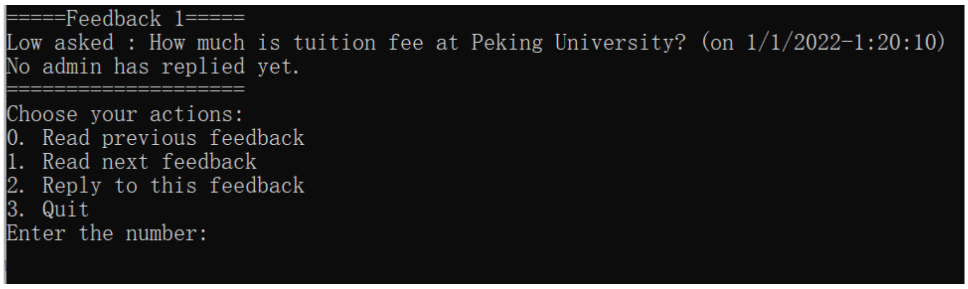
Read And Reply Feedback Page



```
*****Display Feedbacks Page*****
=====Feedback 4=====
Woo asked : Hi, may I ask something? (on 5/3/2022-13:28:34)
No admin has replied yet.
====================
Choose your actions:
0. Read previous feedback
1. Read next feedback
2. Reply to this feedback
3. Quit
Enter the number:
```

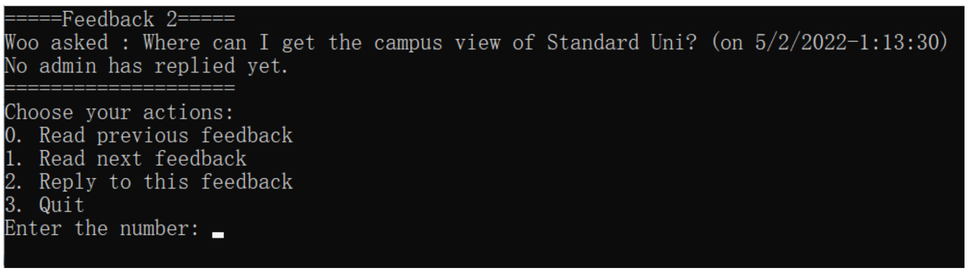
Figure 6.3.13: Display Feedback Page

Admins will be directed to this page when they choose option “4”. This page initially will display the latest feedback from customers with customer username, feedback dates, and whether any admin has replied to the feedback. Admins have four options, which are read previous feedback, read next feedback, reply to current feedback, and quit the function corresponding to 0 to 3. For option 3, admin will be redirected to Admin Home Page for admin to choose any other desired functions.



```
=====Feedback 1=====
Low asked : How much is tuition fee at Peking University? (on 1/1/2022-1:20:10)
No admin has replied yet.
====================
Choose your actions:
0. Read previous feedback
1. Read next feedback
2. Reply to this feedback
3. Quit
Enter the number:
```

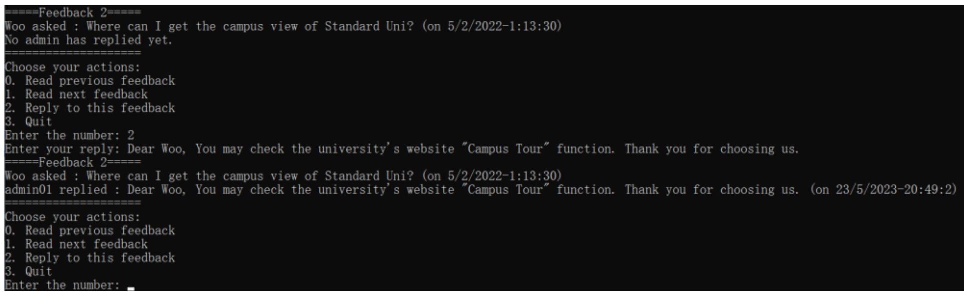
Figure 6.3.14: Reading previous feedback



```
=====Feedback 2=====
Woo asked : Where can I get the campus view of Standard Uni? (on 5/2/2022-1:13:30)
No admin has replied yet.
====================
Choose your actions:
0. Read previous feedback
1. Read next feedback
2. Reply to this feedback
3. Quit
Enter the number:
```

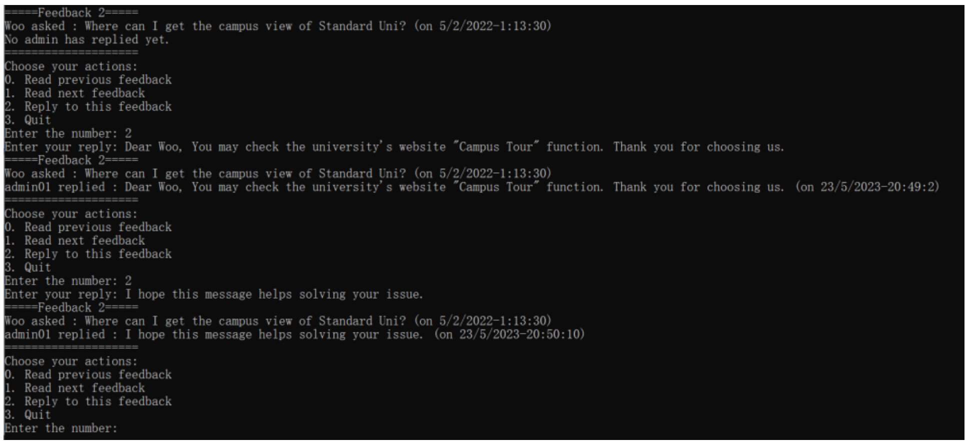
Figure 6.3.15: Reading next feedback

As shown in *Figure 6.3.14*, the feedback has moved to the first feedback sent by “Low” using option 0. And the next figure shows the next feedback sent by “Woo” using option 1. Admin can choose to continue checking other feedback using option 0 and 1 or reply to the current feedback displayed on the screen using option 2.



```
=====Feedback 2=====
Woo asked : Where can I get the campus view of Standard Uni? (on 5/2/2022-1:13:30)
No admin has replied yet.
====================
Choose your actions:
0. Read previous feedback
1. Read next feedback
2. Reply to this feedback
3. Quit
Enter the number: 2
Enter your reply: Dear Woo, You may check the university's website "Campus Tour" function. Thank you for choosing us.
=====Feedback 2=====
Woo asked : Where can I get the campus view of Standard Uni? (on 5/2/2022-1:13:30)
admin01 replied : Dear Woo, You may check the university's website "Campus Tour" function. Thank you for choosing us. (on 23/5/2023-20:49:2)
====================
Choose your actions:
0. Read previous feedback
1. Read next feedback
2. Reply to this feedback
3. Quit
Enter the number:
```

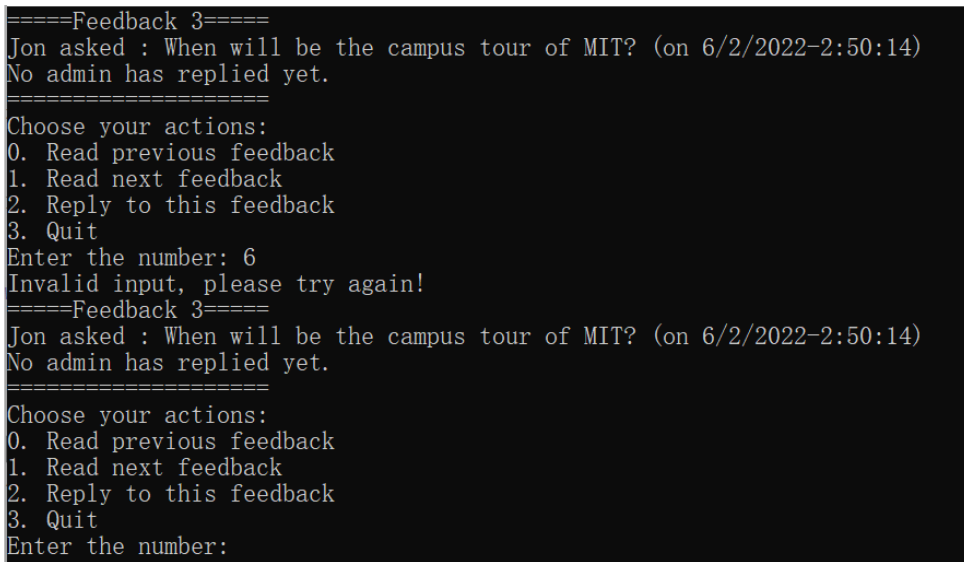
Figure 6.3.16: Replying feedback



```
=====Feedback 2=====
Woo asked : Where can I get the campus view of Standard Uni? (on 5/2/2022-1:13:30)
No admin has replied yet.
====================
Choose your actions:
0. Read previous feedback
1. Read next feedback
2. Reply to this feedback
3. Quit
Enter the number: 2
Enter your reply: Dear Woo, You may check the university's website "Campus Tour" function. Thank you for choosing us.
=====Feedback 2=====
Woo asked : Where can I get the campus view of Standard Uni? (on 5/2/2022-1:13:30)
admin01 replied : Dear Woo, You may check the university's website "Campus Tour" function. Thank you for choosing us. (on 23/5/2023-20:49:2)
====================
Choose your actions:
0. Read previous feedback
1. Read next feedback
2. Reply to this feedback
3. Quit
Enter the number: 2
Enter your reply: I hope this message helps solving your issue.
=====Feedback 2=====
Woo asked : Where can I get the campus view of Standard Uni? (on 5/2/2022-1:13:30)
admin01 replied : I hope this message helps solving your issue. (on 23/5/2023-20:50:10)
====================
Choose your actions:
0. Read previous feedback
1. Read next feedback
2. Reply to this feedback
3. Quit
Enter the number:
```

Figure 6.3.17: Replying another message to the same feedback

Figure 6.3.16 and *Figure 6.3.17* show how admin can reply to feedback using option 2. The first figure shows how admin can respond to feedback and once it is replied, customer can check the respond from his side. Admin account will be displayed with the respond information and respond date and time. As for the second figure, admin has replied again to the same feedback. In this case, the previous respond feedback will be replaced by the latest response from the admin.

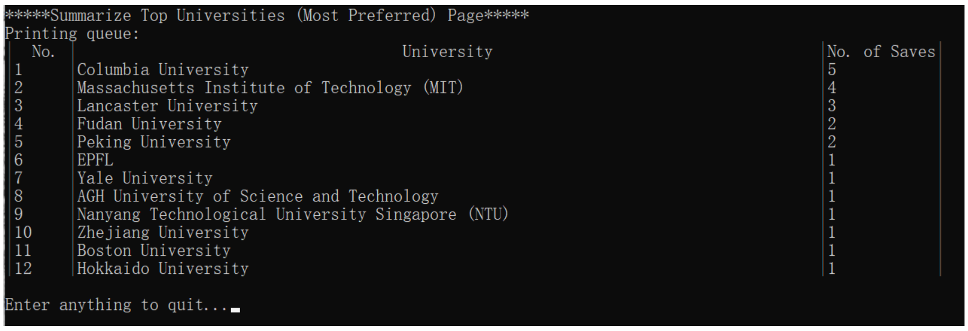


```
=====Feedback 3=====
Jon asked : When will be the campus tour of MIT? (on 6/2/2022-2:50:14)
No admin has replied yet.
====================
Choose your actions:
0. Read previous feedback
1. Read next feedback
2. Reply to this feedback
3. Quit
Enter the number: 6
Invalid input, please try again!
=====Feedback 3=====
Jon asked : When will be the campus tour of MIT? (on 6/2/2022-2:50:14)
No admin has replied yet.
====================
Choose your actions:
0. Read previous feedback
1. Read next feedback
2. Reply to this feedback
3. Quit
Enter the number:
```

Figure 6.3.18: Invalid input for options

Figure above shows the warning message and newly generated Feedback Page for admin to enter the correct input when admin tries to enter any other input that is out of the options showing in the page.

Summarize Top Universities Page



```
*****Summarize Top Universities (Most Preferred) Page*****
Printing queue:
|   No.  |                  University                      |No. of Saves|
| 1      |Columbia University                               |5           |
| 2      |Massachusetts Institute of Technology (MIT)       |4           |
| 3      |Lancaster University                              |3           |
| 4      |Fudan University                                  |2           |
| 5      |Peking University                                 |2           |
| 6      |EPFL                                              |1           |
| 7      |Yale University                                   |1           |
| 8      |AGH University of Science and Technology          |1           |
| 9      |Nanyang Technological University Singapore (NTU)  |1           |
| 10     |Zhejiang University                               |1           |
| 11     |Boston University                                 |1           |
| 12     |Hokkaido University                               |1           |

Enter anything to quit...
```

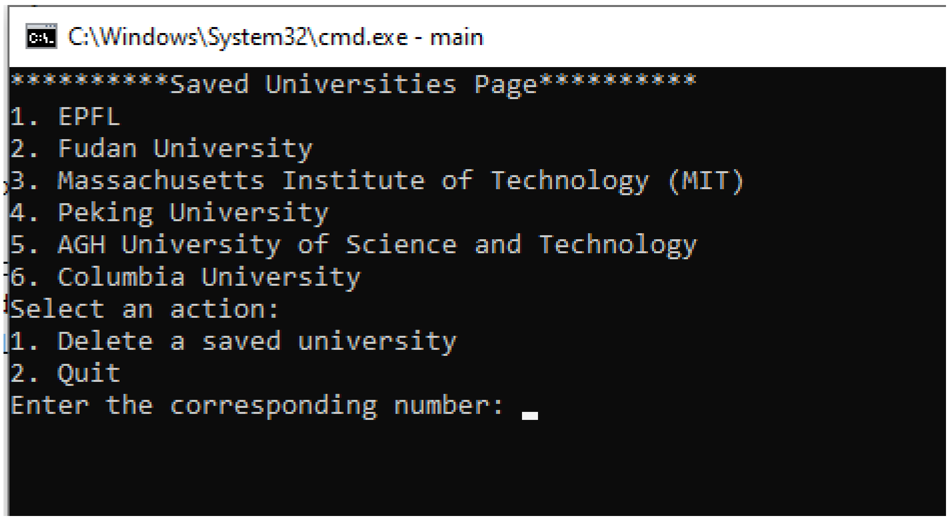
Figure 6.3.19: Summarized table of top saved universities page

Admin will be directed to this page when he enters option 5 at Admin Home Page. As shown in the figure above, all top universities that are saved by the customers are displayed in a table format with numbering, university name, and number of saves by the customers. This page shows the university most preferred by every customer. Number of saves column shows how many customers have saved a particular university, the ranking/numbering is ranked based on this result. Admin can choose to enter any integer or string to exit the function and the system will redirect admin back to Admin Home Page.

7 Extra Features

This section is for highlighting some of the extra features which are not stated in assignment requirements.

7.1 Check and Delete of Saved Universities from Favourite List

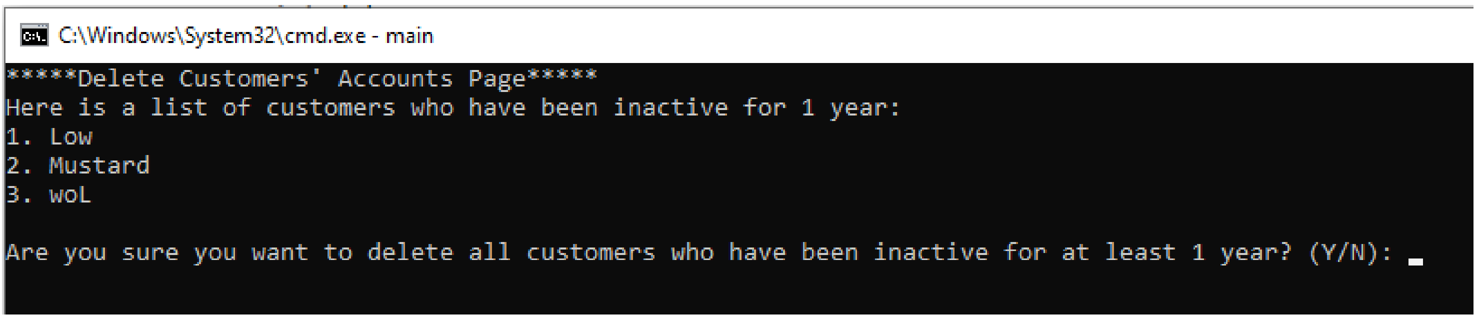


```
***********Saved Universities Page***********
1. EPFL
2. Fudan University
3. Massachusetts Institute of Technology (MIT)
4. Peking University
5. AGH University of Science and Technology
6. Columbia University
Select an action:
1. Delete a saved university
2. Quit
Enter the corresponding number: _
```

Figure 7.1.1: Deleting Saved University from Favourite List

Customers are allowed to check all the saved universities on this page, then enter digit 1 to delete any saved university from the favourite list.

7.2 Displaying Inactive Customer Accounts



```
*****Delete Customers' Accounts Page*****
Here is a list of customers who have been inactive for 1 year:
1. Low
2. Mustard
3. woL

Are you sure you want to delete all customers who have been inactive for at least 1 year? (Y/N): _
```

Figure 7.1.1: Displaying Inactive Customer Accounts before Deletion

Figure 7.1.1 shows that a list of inactive customers will be displayed first to admins. Admins can decide whether they want to delete all those inactive customers.

8. Conclusion

High school parents will find this new university recommendation system a huge time-saver when it comes to helping their children apply to colleges. This cutting-edge system was developed to facilitate the discovery of fresh, precise information about businesses anywhere on the globe. It was developed in tandem with the MoHE and its associated agencies. This method, which is grounded on the enormous 2023 QS University Rankings dataset, is meant to assist parents in making well-informed decisions about their children's educational opportunities.

The recommendation system's user-friendly interface guarantees that the demands of different user types, such as casual browsers, registered users (customers), and MoHE administrators, are met effectively. Features and functionality are tailored to meet the individual requirements of each user population. In addition, our talented group has cleverly added extra features, including the "Save Favourite University" capability, to further improve the user experience and provide useful information.

The university recommendation system has unquestionable advantages, but its limits must be understood before using it to guide decisions about foreign higher education. Constraints include its reliance on a single dataset (2023 QS University Rankings), which, despite its size, may not be able to capture all aspects of a university's performance or consumers' preferences. For optimal efficiency, the system makes use of complicated data structures and C++ programming; yet it requires regular updates to keep up with the dynamic nature of higher education.

In addition, the system's emphasis on quantitative indicators may not completely account for qualitative elements like campus atmosphere, extracurricular possibilities, or the availability of support services that may impact a user's decision-making process. Also, it is important to think about the possibility of inequalities in the recommendation system's accessibility, especially for users with low digital literacy or who live in areas with poor internet connectivity.

Despite these limitations, the university recommendation system is a major improvement for guidance in applying to universities and preparing for higher education. The cutting-edge technology, extensive data, and user-friendly layout that make up this new standard set the bar in terms of quality. The system is well-suited to inspire well-informed decision making and support the aspirations of numerous students thanks to its richness of features and attention to satisfying the various expectations of its users.

In conclusion, the university recommendation system is a prime example of the efficacy of innovation in meeting the difficult needs of parents and students as they attempt to negotiate the vast terrain of international higher education. Users may get a wealth of information to help them choose the best educational route for them with this innovative approach. The system's potential to revolutionize the process of preparing for higher education is highlighted by its forward-thinking design, wealth of features, and commitment to user delight, notwithstanding the caveats.

9. Future Works

As we look forward to the evolution and enhancement of the present university recommendation system, there are many possibilities to solve the limitations that we discussed above. If it is going to help parents and students make sense of the maze that is today's higher education system, the platform has to keep developing in this manner.

So, including additional data sources like the Times Higher Education World University Rankings and the Academic Ranking of World Universities is one method to better serve the demands of a broad range of consumers. Users will have access to a more in-depth understanding of each institution thanks to the incorporation of qualitative data including student evaluations, descriptions of campus culture, and lists of accessible support services.

The system's ability to provide recommendations will be much improved by the incorporation of an adaptive algorithm that takes into account individuals' unique requirements and tastes. The more data it collects from users, the more tailored and exact the suggestions it can provide. Future improvements should concentrate on making the platform more accessible to people with low levels of digital literacy or who live in locations with spotty internet service in order to close this gap. A user-friendly interface, optimisation for low-bandwidth connections, and offline support might all be required to accomplish this.

Maintaining compatibility in the ever-changing world of higher education requires regular system upgrades. Regular data updates, the incorporation of new institutions, and the adaptation of the system's algorithms and features to reflect developments in technology and analysis are all in the works.

Finally, we can offer a comprehensive solution for higher education planning that vastly improves the user experience by incorporating new features like virtual campus tours, direct communication with university representatives, and social media platforms.

In conclusion, efforts to enhance the university recommendation system will centre on eliminating obstacles and increasing the system's functions to accommodate the needs of a broad variety of students. By regularly combining new data sources, qualitative data, adaptive algorithms, and state-of-the-art features, the system will continually adapt to the evolving nature of higher education. Updates like this will make the university recommendation system an even more innovative and critical part of the higher education planning and decision-making process.

10. Reflection

The development of the university recommendation system has been very helpful and illuminating, offering a unique glimpse into the complexities of higher education planning and decision making. We encountered several challenges and picked up many lessons while we worked on this project, all of which contributed to our maturing grasp of data structures, algorithms, and their practical applications. Clarity, concision, formality, inclusiveness, and a broad vocabulary were all factors in making the finished result seem fresh and creative.

We have learned through this process how important it is to use several data sets to build a powerful recommendation engine. Using the 2023 QS University Rankings data set, we were able to see the problems of depending on only one data source. To create a more precise and comprehensive system, we investigated including further datasets and qualitative information.

We gained an appreciation for the value of algorithm efficiency via our comparison and analysis of different sorts and searches. Our success in identifying the best algorithms for our system exemplifies the practical importance of algorithm optimization.

The university recommendation system has come a long way, but we are well aware that there is still potential for improvement. Accessibility, user experience, including adaptive algorithms, and introducing advanced user feedback mechanisms are all possible areas for development in future editions. Taking care of these problems would ensure that the system remains updated and functional for its users.

We learned the importance of using only secure coding practises by the project's completion. It was found that using correct indentation, meaningful identifier names, and strategically placed comments were crucial to preserving code readability and fostering productive communication among team members. The value of these methods and their bearing on the final product of a project have been shown by this experience.

In conclusion, we found this project on developing a university recommendation system to be really fruitful since it allowed us to grow in our knowledge of the many factors that go into the planning and decision-making processes inherent in higher education.

The challenges we have faced and the talents we have refined are enhancing our knowledge of data structures, algorithms, and their practical applications. Eagerly anticipating building upon this basis and correcting the system's limits, we want to develop an original, irreplaceable resource for the design of higher education.

11. References

- *C++ Tutorial - GeeksforGeeks*. (2019, November 25). GeeksforGeeks. https://www.geeksforgeeks.org/cpp-tutorial/
- *C++ Tutorial*. (2019). W3schools.com. https://www.w3schools.com/cpp/
- *C++ Tutorial - Tutorialspoint*. (2019). Tutorialspoint.com. https://www.tutorialspoint.com/cplusplus/index.htm
- (2022). Cplusplus.com. https://cplusplus.com/doc/tutorial/
- *Learn C++ Tutorial - javatpoint*. (n.d.). Www.javatpoint.com. https://www.javatpoint.com/cpp-tutorial
- *Learn C++ (Introduction and Tutorials to C++ Programming)*. (2016). Programiz.com. https://www.programiz.com/cpp-programming
- *Recommendation Systems: Data Structures*. (n.d.). Www.cs.carleton.edu. https://www.cs.carleton.edu/cs_comps/0607/recommend/recommender/datastructures.html
- Zayed, Y., Salman, Y., & Hasasneh, A. (2022). A Recommendation System for Selecting the Appropriate Undergraduate Program at Higher Education Institutions Using Graduate Student Data. *Applied Sciences*, *12*(24), 12525. https://doi.org/10.3390/app122412525
- P, N., Saiteja, K., Ram, K. K., Kanta, K. M., Aditya, S. K., & V, M. (2022, April 1). *University Recommender System based on Student Profile using Feature Weighted Algorithm and KNN*. IEEE Xplore. https://doi.org/10.1109/ICSCDS53736.2022.9760852
- ranadeepika2409 (n.d.). *What are Hash Functions and How to choose a good Hash Function?*. Geeks for geeks. https://www.geeksforgeeks.org/what-are-hash-functions-and-how-to-choose-a-good-hash-function/
- Fulber-Garcia, V. (2023, March 11). *Understanding Hash Tables*. Baeldung. https://www.baeldung.com/cs/hash-tables#:~:text=Hash%20table%20is%20a%20great,1)%20%E2%80%94%20a%20constant%20time.
- aayushi2402 (n.d.). *Applications, Advantages and Disadvantages of Doubly Linked List*. Geeks for geeks. https://www.geeksforgeeks.org/applications-advantages-and-disadvantages-of-doubly-linked-list/
- ankit15697 (n.d.). *Memory leak in C++ and How to avoid it?*. Geeks for geeks. https://www.geeksforgeeks.org/memory-leak-in-c-and-how-to-avoid-it/
- Vithlani, H. (2021, October 4). *Priority Queue*. Scaler. https://www.scaler.com/topics/data-structures/priority-queue-in-data-structure/
- GeeksforGeeks (n.d.). *Introduction to Red-Black Tree*. Geeks for geeks. https://www.geeksforgeeks.org/introduction-to-red-black-tree/

- GeeksforGeeks (n.d.). *Insertion in Red-Black Tree*. Geeks for geeks. https://www.geeksforgeeks.org/insertion-in-red-black-tree/
- Gupta, V. (n.d.). *Red Black Tree: Insertion*. Iq.opengenus. https://iq.opengenus.org/red-black-tree-insertion/#:~:text=Inserting%20a%20value%20in%20Red,O(N)%20space%20complexity.
- Pandey, D. (2022, July 5). *Traversal of Binary Tree*. Scaler. https://www.scaler.com/topics/traversal-of-binary-tree/
- Anand, A. (2021, August 7). *Uses of Doubly Linked List with its Advantages and Disadvantages*. PrepBytes. https://www.prepbytes.com/blog/linked-list/advantages-disadvantages-and-uses-of-a-doubly-linked-list/#:~:text=A%20doubly%20linked%20list%20contains,insertion%20and%20deletion%20of%20nodes.
- Sharma, R. (2022, Oct 4). *Binary Search Algorithm: Function, Benefits, Time & Space Complexity*. upGrad. https://www.upgrad.com/blog/binary-search-algorithm/#:~:text=The%20time%20complexity%20of%20the,values%20not%20in%20the%20list.
- Gupta, V. (n.d.). *Red Black Tree: Search*. Opengenus. https://iq.opengenus.org/red-black-tree-search/#:~:text=Searching%20in%20Red%20Black%20tree,O(N)%20space%20complexity.

12. Appendices

Workload Matrix

NAME (TP. NUMBER)	**WORKLOAD**	**SIGNATURE**
Nivethan Ramesh (TP062192)	• Acknowledgement • Abstract/ Executive Summary • Introduction • Explanations of algorithms • Implementation and Results (Searching and Sorting Algorithms) • Conclusion • Future works • Reflection	Nivethan
Low Sim Chuan (TP065697)	• Implementations of hash table and red black tree • Explanations of algorithms • Implementation and Results (Searching and Sorting Algorithms) • Combining source codes from all members	Chuan.
Andy Wijaya (TP062789)	• Singly linked list • Expected output for customer • Explanation of algorithms	andy
Han RuiMing (TP066161)	• Feedback list (Double linked list) • Saved Uni List (Double linked list) • Uni Array (Array of Structure) • Expected output for normal users • Expected output for admins	[illegible]