GROUP ASSIGNMENT

TECHNOLOGY PARK MALAYSIA

CT077-3-2-DSTR-022022-CMM

DATA STRUCTURES

APU2F2109CS(DA)

HAND OUT DATE: 28 MARCH 2022

HAND IN DATE: 3 JUNE 2022

WEIGHTAGE: 40%

GROUP 30

NAME	TP NUMBER
RYAN MARTIN	TP058091
MATTHEW VINCENT GUNAWAN	TP059019
SELVAN NICHOLAS	TP058084

Table of Contents

1.0 Introduction **7**

1.1 System introduction 7

1.2 Data structures used 7

2.0 Implementation **12**

2.1 Tutor struct implementation 12

2.2 ArrayList class implementation 13

2.3 LinkedList class implementation 18

2.4 Utility functions 25

2.5 Menu functions 27

2.6 Program entry point 34

3.0 Results **34**

3.1 System I/O Screenshots 34

3.2 System Evaluation 43

4.0 Conclusion **44**

5.0 References **44**

6.0 Appendix **44**

1.0 Introduction

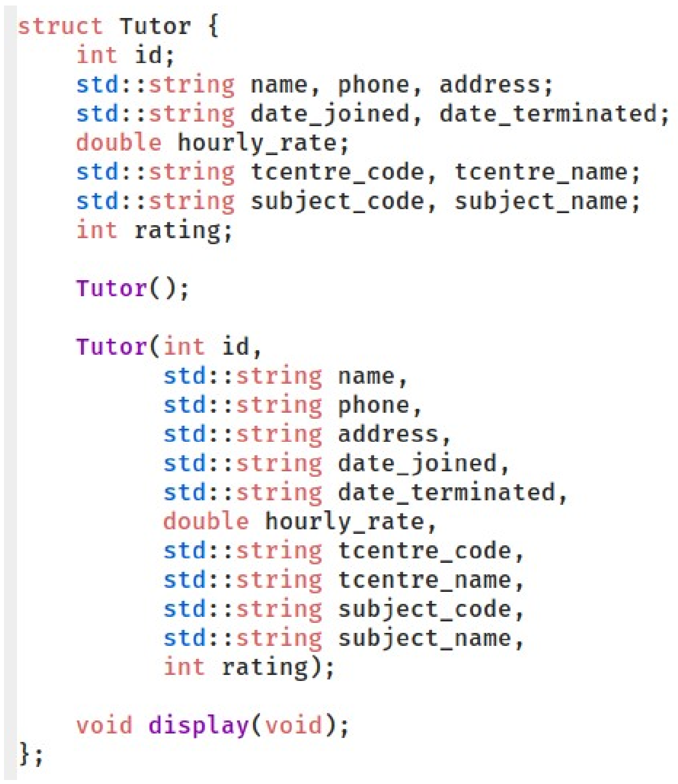
1.1 System introduction

The tuition Centre management system is a text-based command line program to be used by the admin at the eXcel Tuition Centre headquarters to manage the data of their tutors. Admins of the system will need to login with the admin credentials to be able to use the system. The system includes all functionalities of a create, read, update, delete (CRUD) system. These functionalities include adding new tutors, modifying the information on certain tutors, deleting tutor records, searching for tutors according to their IDs or ratings, and sorting the list of tutors ascendingly using either their IDs, hourly pay rates, or ratings. One thing to note is that tutor records can only be deleted after they have been terminated for at least 6 months.

The system is developed in the C++ programming language and uses the list abstract data type as the main data storage. The list structure will be implemented using 2 data structures: arrays and linked lists. The admins can choose which one to use for the system before logging in. Each of these data structures has its own advantages and disadvantages depending on the type of operation performed. The system will be developed using object-oriented programming (OOP), making use of the features provided by C++ for OOP. This approach makes the program more modular and easier to update in the future.

1.2 Data structures used

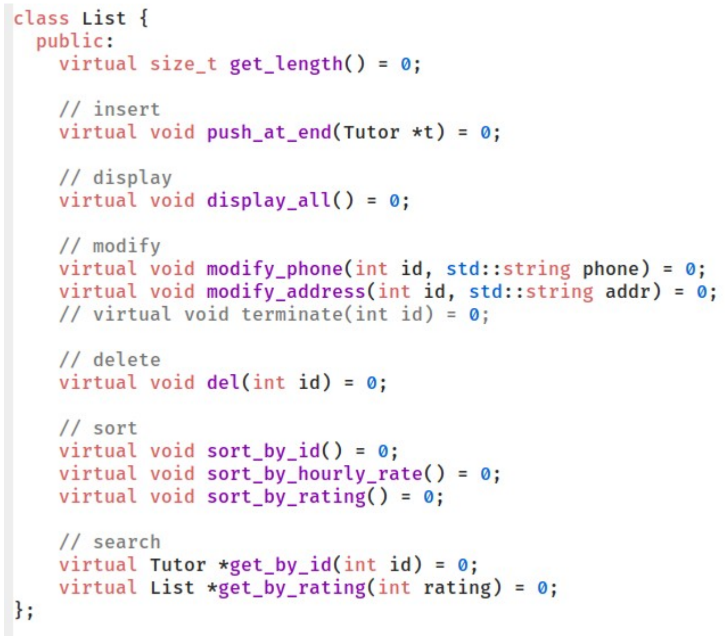
1.2.1 The Tutor structure



```
struct Tutor {
    int id;
    std::string name, phone, address;
    std::string date_joined, date_terminated;
    double hourly_rate;
    std::string tcentre_code, tcentre_name;
    std::string subject_code, subject_name;
    int rating;

    Tutor();

    Tutor(int id,
          std::string name,
          std::string phone,
          std::string address,
          std::string date_joined,
          std::string date_terminated,
          double hourly_rate,
          std::string tcentre_code,
          std::string tcentre_name,
          std::string subject_code,
          std::string subject_name,
          int rating);

    void display(void);
};
```

The Tutor structure contains all the information needed about the tutor, which includes their ID, name, phone number, address, date joined, date terminated, hourly rate, tuition Centre code, tuition Centre name, subject code, subject name, and rating (overall performance). This data type is implemented using structs because of the automatic public visibility of its properties. In C++, classes and structs have the same implementation underneath, with the only difference being that class properties by default have private visibility, while structs have public visibility (Jaeger, 2008). The constructor for this struct takes in all the necessary information specified earlier as its parameters. It also contains a single method, the *display* method. This method prints the details of the tutor stored in a readable format for the admins of this system.

1.2.2 The List interface



```
class List {
  public:
    virtual size_t get_length() = 0;

    // insert
    virtual void push_at_end(Tutor *t) = 0;

    // display
    virtual void display_all() = 0;

    // modify
    virtual void modify_phone(int id, std::string phone) = 0;
    virtual void modify_address(int id, std::string addr) = 0;
    // virtual void terminate(int id) = 0;

    // delete
    virtual void del(int id) = 0;

    // sort
    virtual void sort_by_id() = 0;
    virtual void sort_by_hourly_rate() = 0;
    virtual void sort_by_rating() = 0;

    // search
    virtual Tutor *get_by_id(int id) = 0;
    virtual List *get_by_rating(int rating) = 0;
};
```

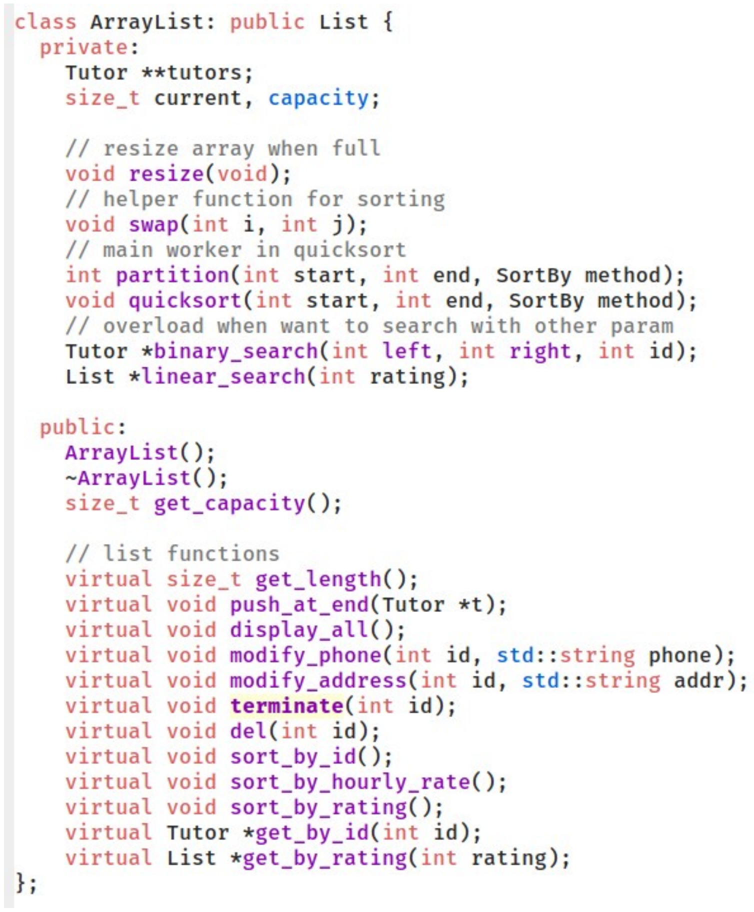
The List interface is used to define the methods needed by a list abstract data type. In OOP, an interface defines the set of properties and/or methods that are available for classes that “implements” the interface. In languages like Java, there is a built-in interface in the language semantics (Uri, 2010). In C++ however, there is no such language construct. In this case, an abstract class with only virtual methods is used to imitate an interface. This class will then be inherited by the array and linked list classes. For the sake of simplicity, from this point on this abstract class will be referred to as just an interface for simplicity.

This interface contains 12 virtual methods required to be implemented by the child classes that inherited from it. The *get_length* method, as its name suggests, is used to get the current length (the number of items) in the list. Adding elements to the list is done using the *push_at_end* method, which takes in the tutor to add to the list and inserts it to the end of the list. The *display_all* method displays all the tutors in the list.

The *modify_phone* and *modify_address* methods are used to modify the *phone* and *address* properties of a certain tutor, determined by the argument passed for the id parameter. The terminate method is the same as the previous modification methods, with the only difference being that it does not require user input to change the value to. It will use the current date to replace the *date_terminated* property. The *del* method, as its name suggests, is used to delete or remove tutors from the list.

The *sort_by* methods are used to sort the list depending on the properties of the tutor such as ID, hourly rate, and rating. The sorting algorithms used will depend on the data structure that implements it, and different classes can use different algorithms for these sorting methods. The same goes for the searching methods, where different classes can use different searching algorithms such as binary or linear search. The methods for searching are the *get_by* methods, depending on the property to search by, which in this case are the ID and rating of the tutor. In the case of searching by the tutor rating, there can be multiple results as there can be multiple tutors with the same rating.

1.2.3 The ArrayList class



```cpp
class ArrayList: public List {
  private:
    Tutor **tutors;
    size_t current, capacity;

    // resize array when full
    void resize(void);
    // helper function for sorting
    void swap(int i, int j);
    // main worker in quicksort
    int partition(int start, int end, SortBy method);
    void quicksort(int start, int end, SortBy method);
    // overload when want to search with other param
    Tutor *binary_search(int left, int right, int id);
    List *linear_search(int rating);

  public:
    ArrayList();
    ~ArrayList();
    size_t get_capacity();

    // list functions
    virtual size_t get_length();
    virtual void push_at_end(Tutor *t);
    virtual void display_all();
    virtual void modify_phone(int id, std::string phone);
    virtual void modify_address(int id, std::string addr);
    virtual void terminate(int id);
    virtual void del(int id);
    virtual void sort_by_id();
    virtual void sort_by_hourly_rate();
    virtual void sort_by_rating();
    virtual Tutor *get_by_id(int id);
    virtual List *get_by_rating(int rating);
};
```

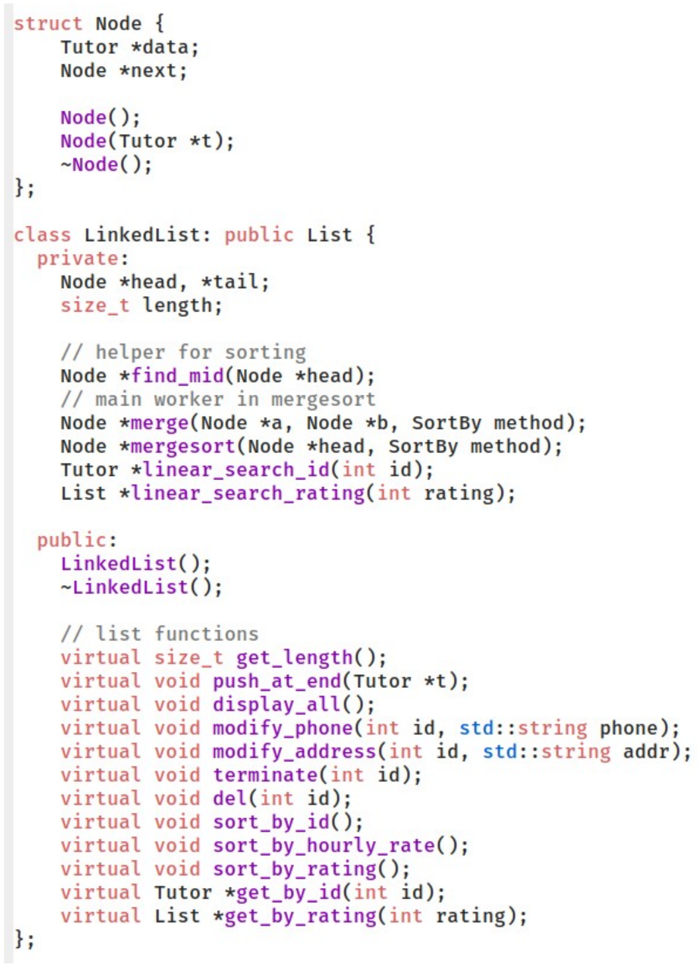
The ArrayList class is used to implement the array data structure. It inherits the List abstract class and implements all the list methods mentioned above. The properties of the class include the actual array used to store the tutors (*tutors*), the current list length (*current*), and the current maximum capacity of the array (*capacity*). The array will be dynamically allocated during object initialization, and the initial capacity is defined by the *ARRAY_INITIAL_CAPACITY* macro definition, which is set to 10. When the array is full, it will be resized to double the current capacity to accommodate future tutors. This is done by the *resize* method.

The searching algorithms used will be both binary and linear search, depending on which tutor property is being searched by. For searching using ID, binary search will be used, because of the algorithm's time complexity of O(log n). One drawback of using this search algorithm is that the array needs to be sorted beforehand, and in this case, it needs to be sorted according to the tutor ID. In our current implementation, we sort the array every time before running the binary search, which makes the searching process slower. The sorting algorithm used is quicksort, which has an average complexity of O(nlog n), and this results in a total time complexity of O(nlog n + log n) for search.

There are several ways to avoid this, one of which is to have a "flag" property in the class to keep track of whether the array has been sorted using the ID. Then, before each binary search, the program will check for the status of this flag, and if the value means that the array has been sorted, it will proceed directly to the search, because the array would already be sorted. However, for this assignment, we chose to not implement this as to keep the program simpler.

Regular linear search will be used for searching by rating because multiple results may be returned, while binary search returns only 1 result. The corresponding methods for these search algorithms are the *binary_search* and *linear_search* methods. As mentioned before, the sorting algorithm used will be the quicksort algorithm. This algorithm will be implemented using 3 methods: *quicksort*, which is the method to call to start the sort, *partition*, the main worker of the algorithm, and *swap*, a helper method.

1.2.4 The LinkedList class



```cpp
struct Node {
    Tutor *data;
    Node *next;

    Node();
    Node(Tutor *t);
    ~Node();
};

class LinkedList: public List {
  private:
    Node *head, *tail;
    size_t length;

    // helper for sorting
    Node *find_mid(Node *head);
    // main worker in mergesort
    Node *merge(Node *a, Node *b, SortBy method);
    Node *mergesort(Node *head, SortBy method);
    Tutor *linear_search_id(int id);
    List *linear_search_rating(int rating);

  public:
    LinkedList();
    ~LinkedList();

    // list functions
    virtual size_t get_length();
    virtual void push_at_end(Tutor *t);
    virtual void display_all();
    virtual void modify_phone(int id, std::string phone);
    virtual void modify_address(int id, std::string addr);
    virtual void terminate(int id);
    virtual void del(int id);
    virtual void sort_by_id();
    virtual void sort_by_hourly_rate();
    virtual void sort_by_rating();
    virtual Tutor *get_by_id(int id);
    virtual List *get_by_rating(int rating);
};
```

The LinkedList class is used to implement the linked list data structure. Same as the ArrayList class, this class also inherits the List abstract class and implements its methods. This class implements a singly linked list (SLL) instead of a doubly linked list (DLL). The reason for this is that the functionalities available to DLLs but not to SLLs, such as reverse list traversal, are not necessary for this assignment. DLLs also bring additional complexity by using more pointers. Therefore, because DLL functionalities are not necessary to use and because it is simpler to implement, the SLL will be used.

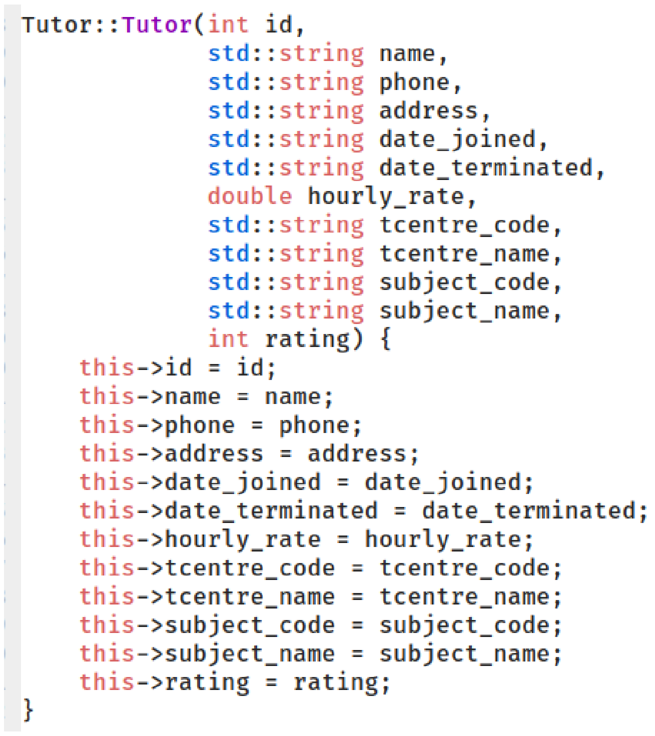
Included in the same file is the class definition for the Node class. This class is used for the nodes of the linked list. It contains 2 properties, *data* which will be used to store the tutor, and *next* which will be used to store the address of the next element in the list. Its constructor takes in a Tutor object to be stored. The LinkedList class will contain 2 node pointers, *head*, and *tail*, which points to the start and the end of the list, respectively. Another property of this class is the *length* property, which stores the current length of the list.

Besides the List interface methods, this class also contains methods for searching and sorting the list. The search algorithm used will be linear search, and this applies for searching using ID and rating. The search is done by the methods *linear_search_id*, which searches using the tutor ID and returns 1 result, and *linear_search_rating*, which searches using the tutor rating and can return 1 or more results. The sorting algorithm used for linked lists will be the merge sort algorithm. It is implemented by the 3 methods: *mergesort*, the main method to call to sort the list, *merge*, a helper method for merging split lists, and *find_mid*, a helper method.

2.0 Implementation

2.1 Tutor struct implementation

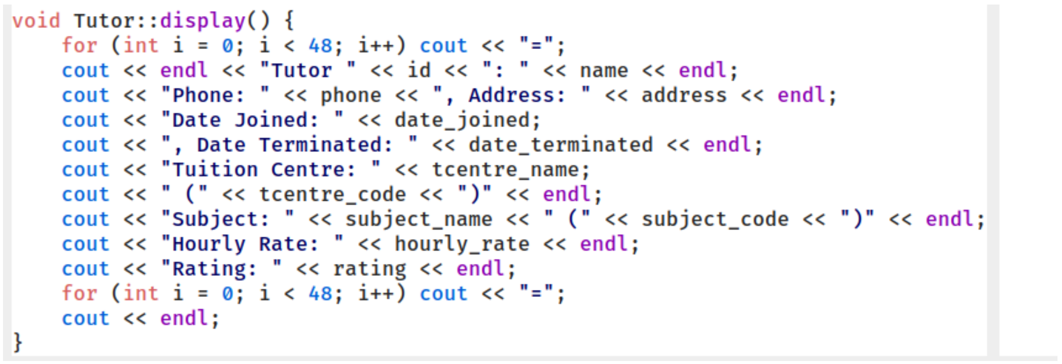
2.1.1 Constructor



```
Tutor::Tutor(int id,
             std::string name,
             std::string phone,
             std::string address,
             std::string date_joined,
             std::string date_terminated,
             double hourly_rate,
             std::string tcentre_code,
             std::string tcentre_name,
             std::string subject_code,
             std::string subject_name,
             int rating) {
    this->id = id;
    this->name = name;
    this->phone = phone;
    this->address = address;
    this->date_joined = date_joined;
    this->date_terminated = date_terminated;
    this->hourly_rate = hourly_rate;
    this->tcentre_code = tcentre_code;
    this->tcentre_name = tcentre_name;
    this->subject_code = subject_code;
    this->subject_name = subject_name;
    this->rating = rating;
}
```

Constructor for the Tutor struct. Simply assigns the arguments passed into the struct properties.

2.1.2 Display method

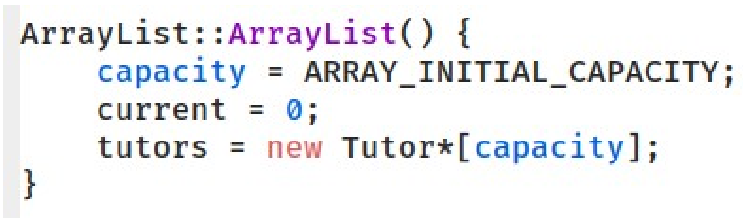


```
void Tutor::display() {
    for (int i = 0; i < 48; i++) cout << "=";
    cout << endl << "Tutor " << id << ": " << name << endl;
    cout << "Phone: " << phone << ", Address: " << address << endl;
    cout << "Date Joined: " << date_joined;
    cout << ", Date Terminated: " << date_terminated << endl;
    cout << "Tuition Centre: " << tcentre_name;
    cout << " (" << tcentre_code << ")" << endl;
    cout << "Subject: " << subject_name << " (" << subject_code << ")" << endl;
    cout << "Hourly Rate: " << hourly_rate << endl;
    cout << "Rating: " << rating << endl;
    for (int i = 0; i < 48; i++) cout << "=";
    cout << endl;
}
```

This method prints the properties of the tutor in a tidy format.

2.2 ArrayList class implementation

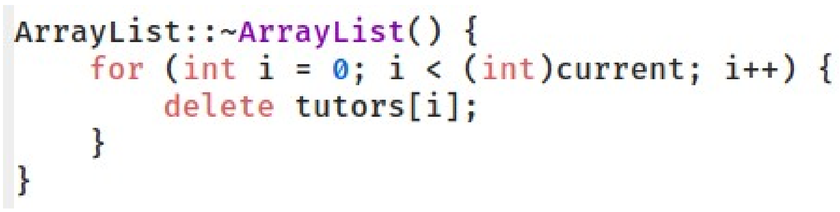
2.2.1 Constructor



```
ArrayList::ArrayList() {
    capacity = ARRAY_INITIAL_CAPACITY;
    current = 0;
    tutors = new Tutor*[capacity];
}
```

The constructor sets *capacity* to the constant, sets *current* to 0 (because the list is still empty), and dynamically allocates the *tutors* array to the size of *capacity*.

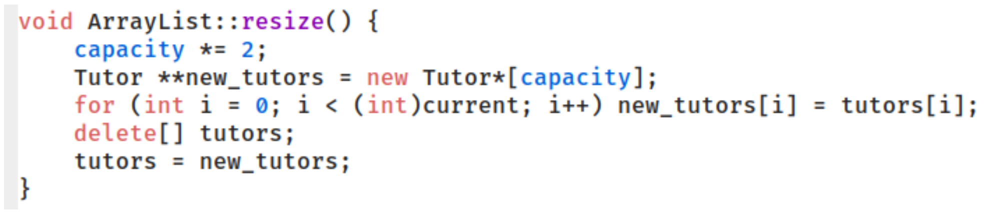
2.2.2 Destructor



```
ArrayList::~ArrayList() {
    for (int i = 0; i < (int)current; i++) {
        delete tutors[i];
    }
}
```

The destructor deletes every tutor item on the array because they were dynamically allocated using the *new* keyword.

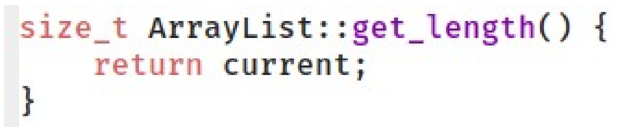
2.2.3 Resize method



```
void ArrayList::resize() {
    capacity *= 2;
    Tutor **new_tutors = new Tutor*[capacity];
    for (int i = 0; i < (int)current; i++) new_tutors[i] = tutors[i];
    delete[] tutors;
    tutors = new_tutors;
}
```

The resize method allocates a new array, *new_tutors* with twice the current capacity, and copies all the items from the *tutors* array into the new one. Then it deletes the *tutors* array and assigns it to the new array.

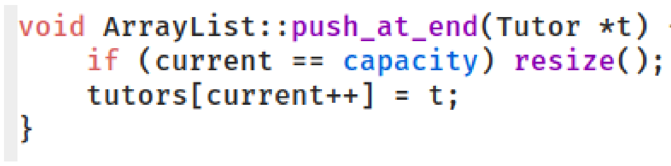
2.2.4 Get_length method



```
size_t ArrayList::get_length() {
    return current;
}
```

Returns the *current* property.

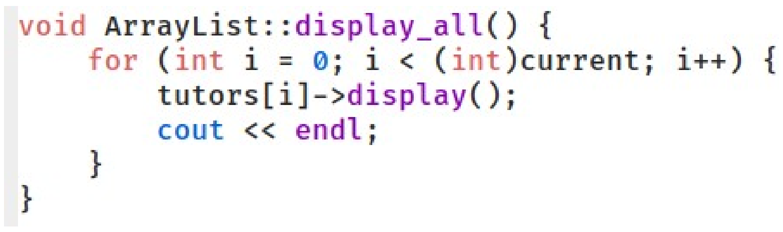
2.2.5 Push_at_end method



```
void ArrayList::push_at_end(Tutor *t) {
    if (current == capacity) resize();
    tutors[current++] = t;
}
```

Before inserting the tutor into the list, check if the number of items in the list (*current)* is the same as the array capacity. If it is, then resize the array. Then insert the tutor in the *tutors* array at index of current and increment current by 1.

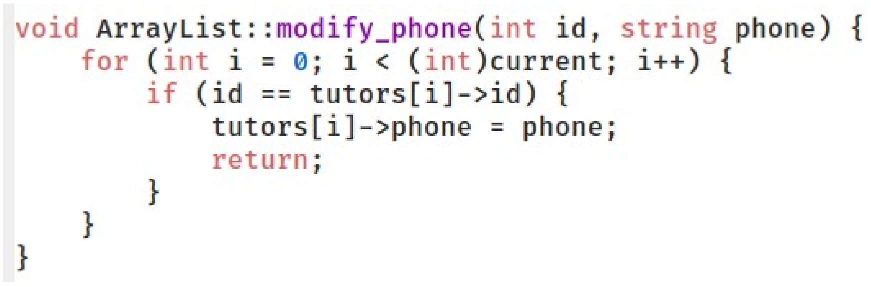
2.2.6 Display_all method



```
void ArrayList::display_all() {
    for (int i = 0; i < (int)current; i++) {
        tutors[i]->display();
        cout << endl;
    }
}
```

Iterate over the *tutors* array and call the *display* method on each tutor in the array.

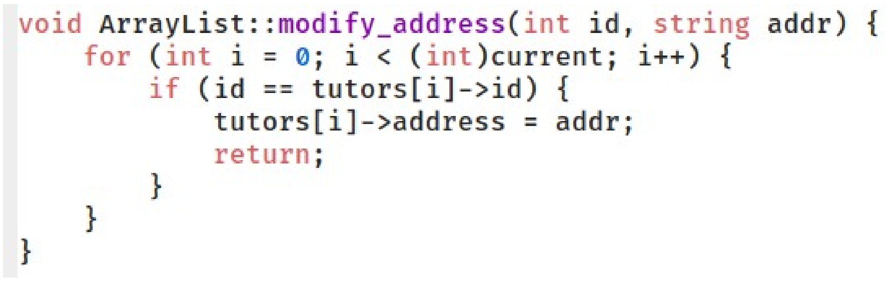
2.2.7 Modify_phone method



```
void ArrayList::modify_phone(int id, string phone) {
    for (int i = 0; i < (int)current; i++) {
        if (id == tutors[i]->id) {
            tutors[i]->phone = phone;
            return;
        }
    }
}
```

Iterate over the *tutors* array, replace the tutor’s phone number property with the *phone* argument passed if the tutor’s *id* property matches with the one passed into the function. Returns immediately after update for efficiency, as there is no need to check the other items because the tutor IDs are unique.

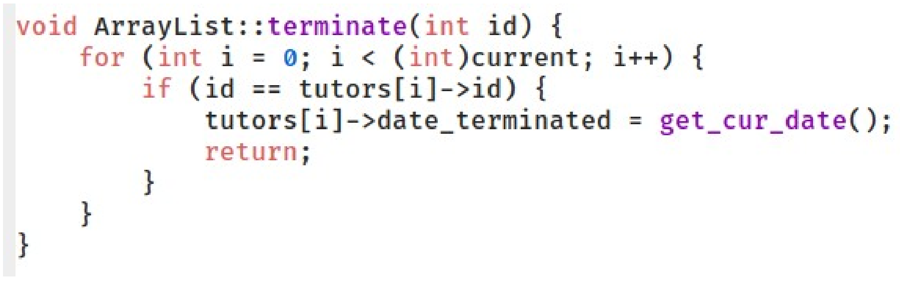
2.2.8 Modify_address method



```
void ArrayList::modify_address(int id, string addr) {
    for (int i = 0; i < (int)current; i++) {
        if (id == tutors[i]->id) {
            tutors[i]->address = addr;
            return;
        }
    }
}
```

Same logic as *modify_phone*.

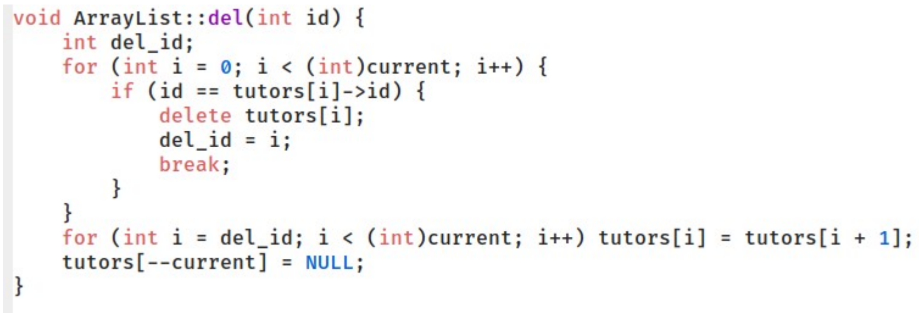
2.2.9 Terminate method



```
void ArrayList::terminate(int id) {
    for (int i = 0; i < (int)current; i++) {
        if (id == tutors[i]->id) {
            tutors[i]->date_terminated = get_cur_date();
            return;
        }
    }
}
```

Same logic as the previous *modify* methods, but replace with current date.

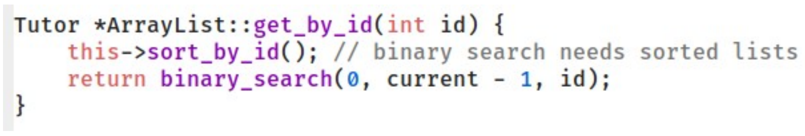
2.2.10 Del method



```
void ArrayList::del(int id) {
    int del_id;
    for (int i = 0; i < (int)current; i++) {
        if (id == tutors[i]->id) {
            delete tutors[i];
            del_id = i;
            break;
        }
    }
    for (int i = del_id; i < (int)current; i++) tutors[i] = tutors[i + 1];
    tutors[--current] = NULL;
}
```

Iterate over the *tutors* array until a tutor with the id of *id* argument passed is found. Then, delete the tutor at that index, and break out of the loop. Iterate again starting from the index of the deleted item and move the item next to the current one into the current slot. Basically, what is being done here is shifting the items at the right of the deleted item slot (which is empty now) to fill the gap. Finally, decrement *current* by 1.

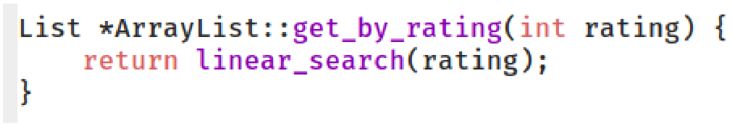
2.2.11 Get_by_id method



```
Tutor *ArrayList::get_by_id(int id) {
    this->sort_by_id(); // binary search needs sorted lists
    return binary_search(0, current - 1, id);
}
```

Sort the array by ID first, then call the binary search method because the actual search algorithm is implemented there.

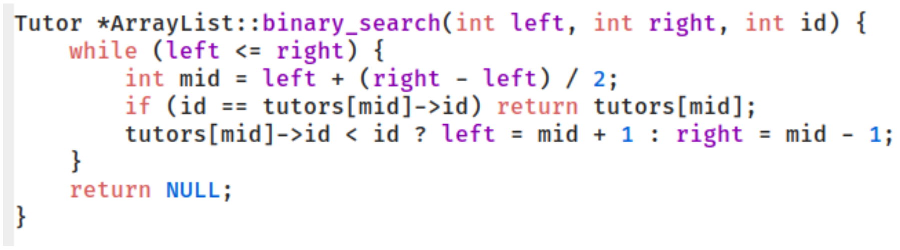
2.2.12 Get_by_rating method



```
List *ArrayList::get_by_rating(int rating) {
    return linear_search(rating);
}
```

Same logic as *get_by_id*, without sorting the array beforehand and calls the *linear_search* method instead of *binary_search*.

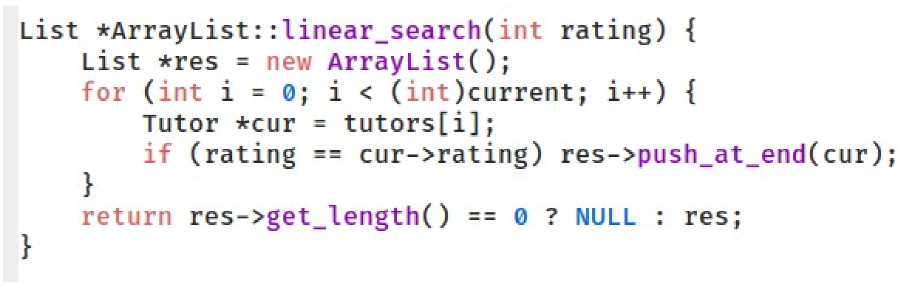
2.2.13 Binary_search method



```
Tutor *ArrayList::binary_search(int left, int right, int id) {
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (id == tutors[mid]->id) return tutors[mid];
        tutors[mid]->id < id ? left = mid + 1 : right = mid - 1;
    }
    return NULL;
}
```

This method has 3 parameters, *left*, which is the leftmost side of the list to search, *right*, the opposite of *left*, and *id*, the id of the tutor to search for. Loop while left is less than or equal to right and calculate the midpoint of the list using the equation in the code snippet above. Then check if the tutor at the midpoint of the array has the id of *id*. If so, return the tutor, else repeat by doing the same process on the left side of the mid-point if the id in the mid-point is greater than *id*, or right side if the midpoint id is less than *id*. If no tutor with the wanted ID is found then return NULL.

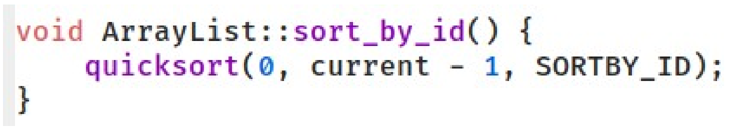
2.2.14 Linear_search method



```
List *ArrayList::linear_search(int rating) {
    List *res = new ArrayList();
    for (int i = 0; i < (int)current; i++) {
        Tutor *cur = tutors[i];
        if (rating == cur->rating) res->push_at_end(cur);
    }
    return res->get_length() == 0 ? NULL : res;
}
```

Because this method may return multiple results, create a new List to store the found tutors. Loop over *tutors* and insert the tutors with ratings that match with the *ratings* argument passed into the new list. If the new list is empty then return NULL, else return the new list.

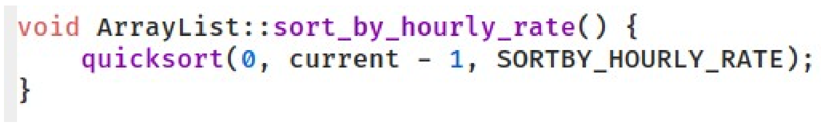
2.2.15 Sort_by_id method



```
void ArrayList::sort_by_id() {
    quicksort(0, current - 1, SORTBY_ID);
}
```

This is similar to the search methods, which calls the methods with different arguments. This method calls quicksort using the *SORTBY_ID* argument, which is an enumeration type that consists of the different properties that the list can be sorted by. The enum itself is defined in the List interface header file.

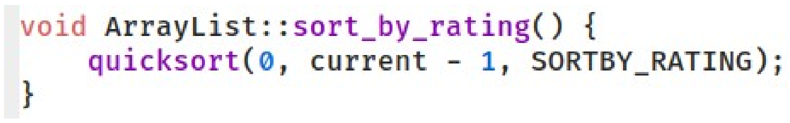
2.2.16 Sort_by_hourly_rate method



```
void ArrayList::sort_by_hourly_rate() {
    quicksort(0, current - 1, SORTBY_HOURLY_RATE);
}
```

Same logic as *sort_by_id*.

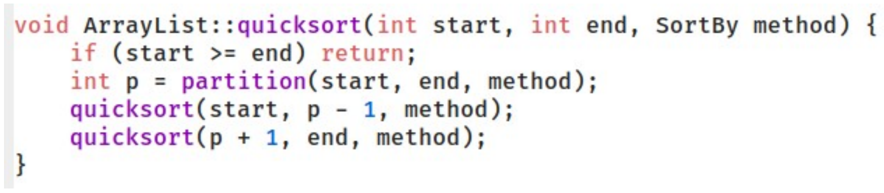
2.2.17 Sort_by_rating method



```
void ArrayList::sort_by_rating() {
    quicksort(0, current - 1, SORTBY_RATING);
}
```

Same logic as the 2 *sort_by* methods above.

2.2.18 Quicksort method

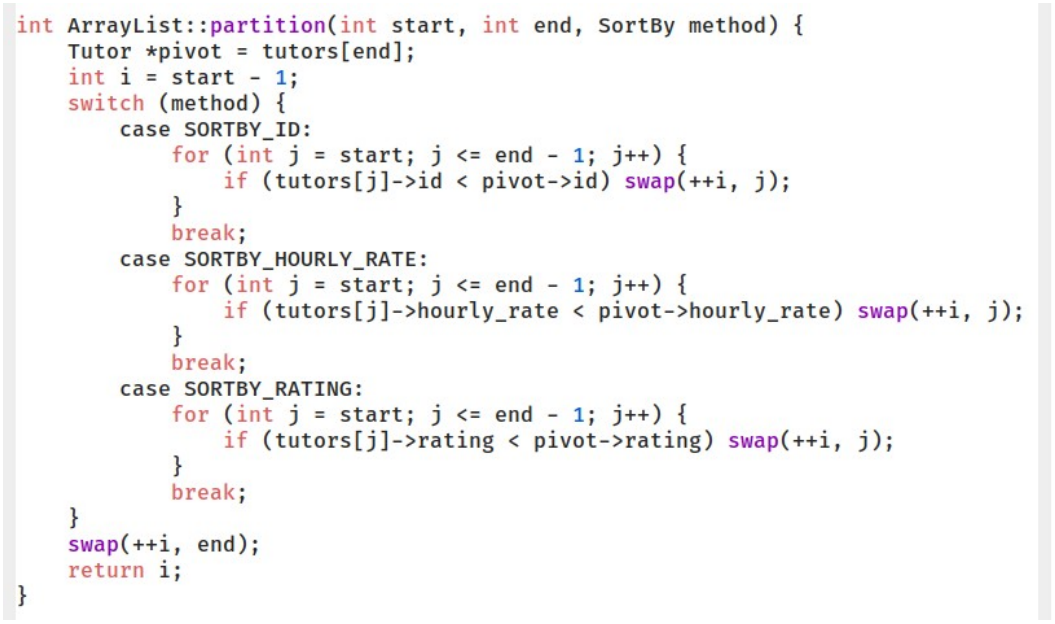


```
void ArrayList::quicksort(int start, int end, SortBy method) {
    if (start >= end) return;
    int p = partition(start, end, method);
    quicksort(start, p - 1, method);
    quicksort(p + 1, end, method);
}
```

This method is recursive and runs until the array is sorted. It first checks if the *start* argument passed is greater than or equal to *end*. If it is, then return. This condition is needed to check in the recursive calls, because in quicksort, the *start* variable will keep moving to the right and will get past *end*, which keeps moving left. When these 2 meet or pass each other, it means that an element in the array has been moved to its correct position. A more detailed explanation of how quicksort works is in our previous proposal document.

After checking the condition, the mid-point of the array is returned by the partition method. Then the method will recursively call quicksort on the sub-arrays on the left and right side of the midpoint. This process will continue until the array can’t be split anymore, which will lead to the array being sorted.

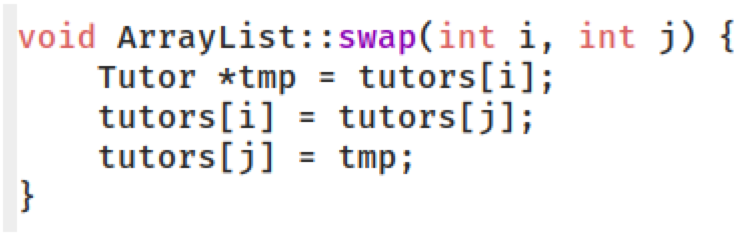
2.2.19 Partition method



```
int ArrayList::partition(int start, int end, SortBy method) {
    Tutor *pivot = tutors[end];
    int i = start - 1;
    switch (method) {
        case SORTBY_ID:
            for (int j = start; j <= end - 1; j++) {
                if (tutors[j]->id < pivot->id) swap(++i, j);
            }
            break;
        case SORTBY_HOURLY_RATE:
            for (int j = start; j <= end - 1; j++) {
                if (tutors[j]->hourly_rate < pivot->hourly_rate) swap(++i, j);
            }
            break;
        case SORTBY_RATING:
            for (int j = start; j <= end - 1; j++) {
                if (tutors[j]->rating < pivot->rating) swap(++i, j);
            }
            break;
    }
    swap(++i, end);
    return i;
}
```

This method involves a series of swapping until the *pivot* tutor is placed in its correct position in the array. The *pivot* is the last element of the array. The method keeps track of the index *i* which will be the correct index of the pivot element. Then iterate over the array and swap the current element with the element at index *i* + 1 if its property to be compared is less than the pivot’s. After the loop has finished, swap the pivot with the element at index *i* + 1. The pivot will be in its correct position and *i* is now the midpoint for the next *quicksort* calls.

2.2.20 Swap method

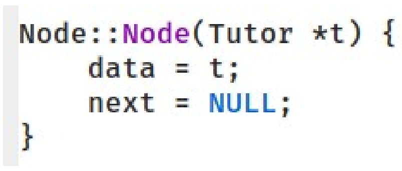


```
void ArrayList::swap(int i, int j) {
    Tutor *tmp = tutors[i];
    tutors[i] = tutors[j];
    tutors[j] = tmp;
}
```

Simple method to swap to elements of the *tutors* array at indices *i* and *j*.

2.3 LinkedList class implementation

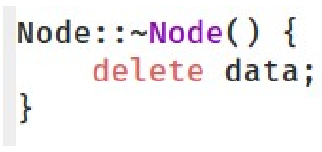
2.3.1 Node constructor



```
Node::Node(Tutor *t) {
    data = t;
    next = NULL;
}
```

The node constructor takes in the tutor object to be inserted into the list and sets its next pointer to NULL.

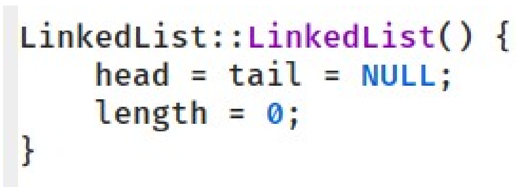
2.3.2 Node destructor



```
Node::~Node() {
    delete data;
}
```

The destructor simply deletes the tutor contained in the node, as it is dynamically allocated.

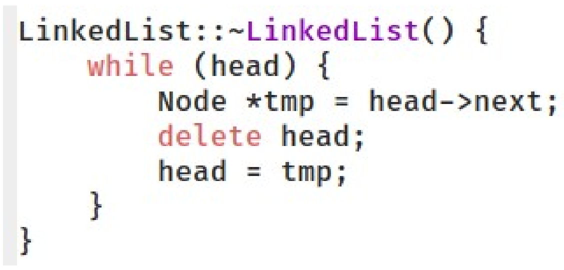
2.3.3 Linked list constructor



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

The linked list constructor simply sets the *head* and *tail* pointers to NULL and the length to 0.

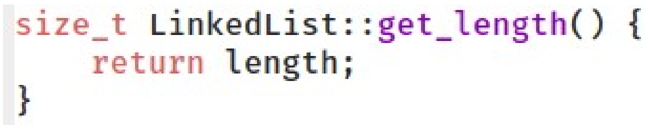
2.3.4 Linked list destructor



```
LinkedList::~LinkedList() {
    while (head) {
        Node *tmp = head->next;
        delete head;
        head = tmp;
    }
}
```

The destructor iterates over the list by following the *next* pointers and deletes the tutor contained in each node.

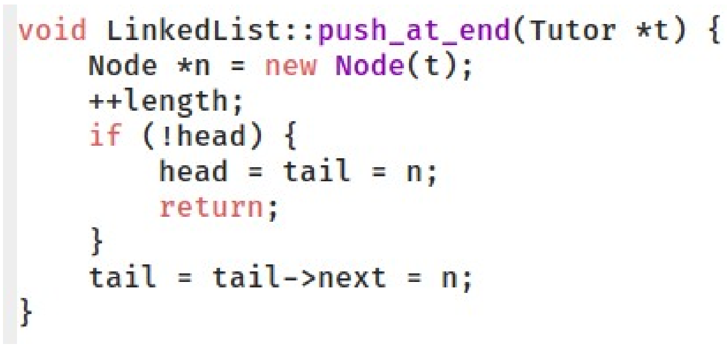
2.3.5 Get_length method



```
size_t LinkedList::get_length() {
    return length;
}
```

The same as in ArrayList, but here it returns the *length* property.

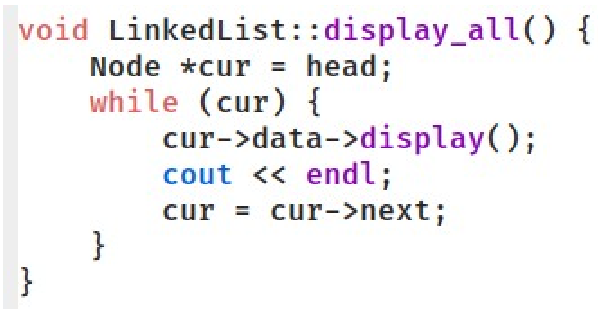
2.3.6 Push_at_end method



```
void LinkedList::push_at_end(Tutor *t) {
    Node *n = new Node(t);
    ++length;
    if (!head) {
        head = tail = n;
        return;
    }
    tail = tail->next = n;
}
```

This method accepts a *Tutor* object and creates a *Node* object using the tutor as its argument. Then it increments the length by 1 and checks if the *head* pointer is NULL, if it is then that means the list is empty and the *head* and *tail* pointer is set to the new node and returns. If not, then it sets the *next* pointer in *tail* (the last item in the list) to the new node and sets the *tail* pointer to point to the new node.

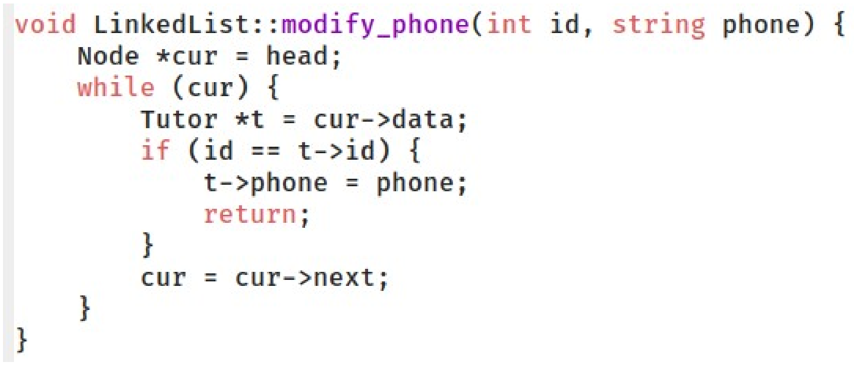
2.3.7 Dislay_all method



```
void LinkedList::display_all() {
    Node *cur = head;
    while (cur) {
        cur->data->display();
        cout << endl;
        cur = cur->next;
    }
}
```

Iterate over the list by following the *next* pointers and call the *display* method on each tutor in the list.

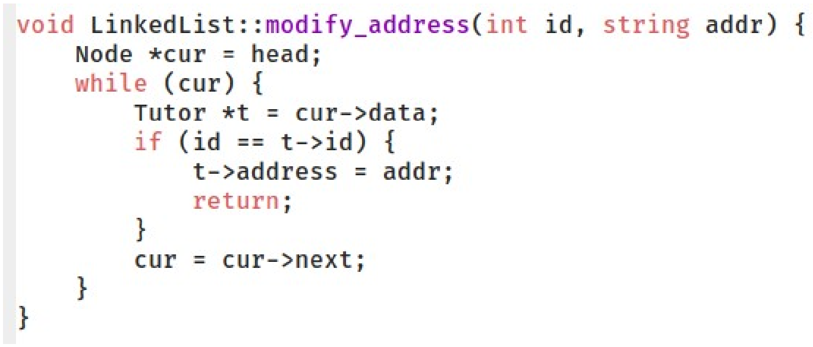
2.3.8 Modify_phone method



```
void LinkedList::modify_phone(int id, string phone) {
    Node *cur = head;
    while (cur) {
        Tutor *t = cur->data;
        if (id == t->id) {
            t->phone = phone;
            return;
        }
        cur = cur->next;
    }
}
```

Iterate over the list by following the *next* pointer, and check if the current node has a tutor with the ID of the *id* argument passed. If it does, replace the current *phone* property of the tutor with the *phone* argument passed into the method. Return from the method after as there is no need to continue any further. If it does not, keep iterating until it is found, or the end of the list is reached. If no correct tutor is found, then no modification will happen.

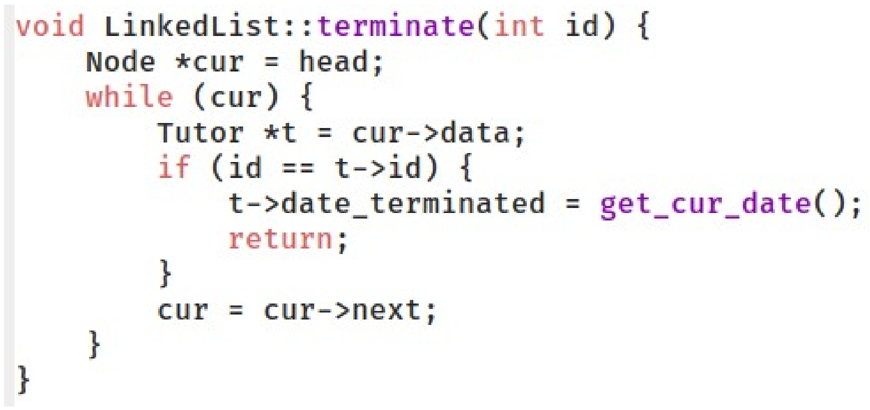
2.3.9 Modify_address method



```
void LinkedList::modify_address(int id, string addr) {
    Node *cur = head;
    while (cur) {
        Tutor *t = cur->data;
        if (id == t->id) {
            t->address = addr;
            return;
        }
        cur = cur->next;
    }
}
```

The same logic as *modify_phone*, but updates the *address* property instead.

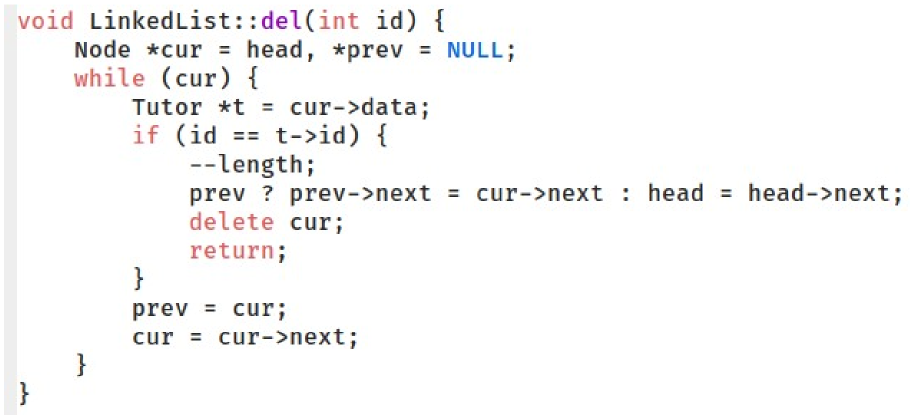
2.3.10 Terminate method



```
void LinkedList::terminate(int id) {
    Node *cur = head;
    while (cur) {
        Tutor *t = cur->data;
        if (id == t->id) {
            t->date_terminated = get_cur_date();
            return;
        }
        cur = cur->next;
    }
}
```

Same logic as the *modify* methods, but replace the value with the current date.

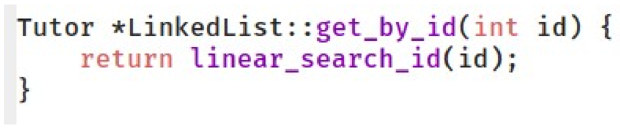
2.3.11 Del method



```
void LinkedList::del(int id) {
    Node *cur = head, *prev = NULL;
    while (cur) {
        Tutor *t = cur->data;
        if (id == t->id) {
            --length;
            prev ? prev->next = cur->next : head = head->next;
            delete cur;
            return;
        }
        prev = cur;
        cur = cur->next;
    }
}
```

Iterate over the array using 2 pointer variables, *cur* to keep track of the current node, and *prev* to store the previously visited node. Iterate over the list and check if the current node's tutor has an ID of the *id* argument passed. If it does, check if the *prev* variable is NULL. If it is, then it means the current node is the first one in the list so set the *head* to be the item after it. If it isn't, then set the previous node's *next* pointer to be the current node's *next* pointer. After that delete the current node and return from the function. At the end of the current iteration set *cur* to be the next item and *prev* to be the current one.

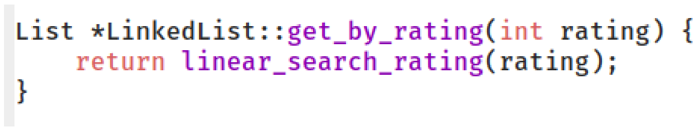
2.3.12 Get_by_id method



```
Tutor *LinkedList::get_by_id(int id) {
    return linear_search_id(id);
}
```

Same as the ArrayList methods, the *get_by* methods implemented by the LinkedList class only call the needed search methods.

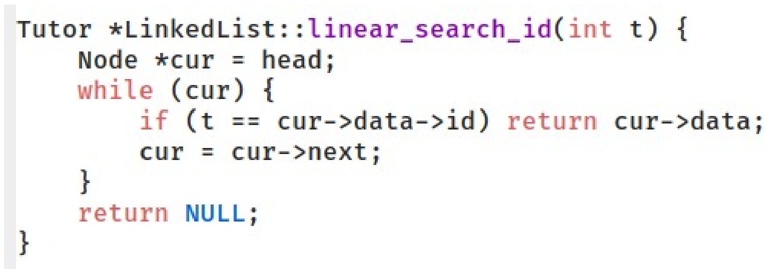
2.3.13 Get_by_rating method



```
List *LinkedList::get_by_rating(int rating) {
    return linear_search_rating(rating);
}
```

Same logic as the *get_by_id* method.

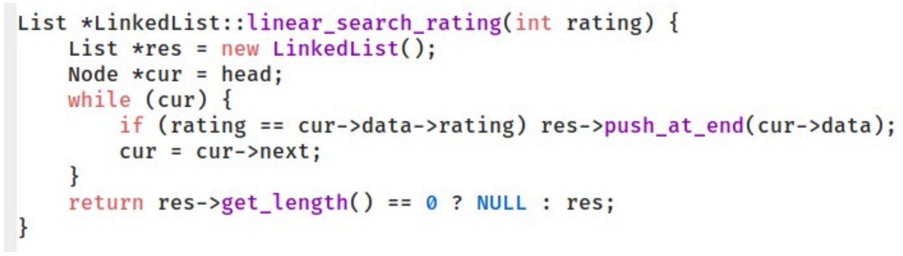
2.3.14 Linear_search_id method



```
Tutor *LinkedList::linear_search_id(int t) {
    Node *cur = head;
    while (cur) {
        if (t == cur->data->id) return cur->data;
        cur = cur->next;
    }
    return NULL;
}
```

Iterates over the list by following the *next* pointer and returns the current node’s tutor if it has an ID that matches with the *id* argument passed. Return NULL if none of the items in the list has a tutor with the correct ID.

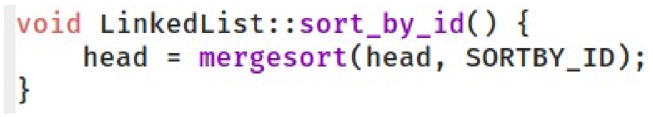
2.3.15 Linear_search_rating method



```
List *LinkedList::linear_search_rating(int rating) {
    List *res = new LinkedList();
    Node *cur = head;
    while (cur) {
        if (rating == cur->data->rating) res->push_at_end(cur->data);
        cur = cur->next;
    }
    return res->get_length() == 0 ? NULL : res;
}
```

Same logic as the *linear_search* method from the ArrayList class.

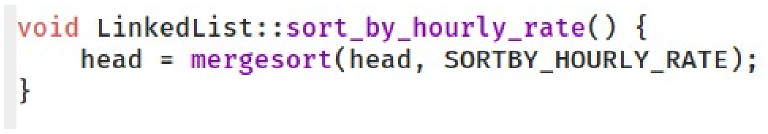
2.3.16 Sort_by_id method



```
void LinkedList::sort_by_id() {
    head = mergesort(head, SORTBY_ID);
}
```

Same as the *sort_by* methods implemented by the ArrayList class, the methods in the LinkedList class also only call the *mergesort* method. The called method returns the head of the list after it has been sorted, so set the *head* property to it.

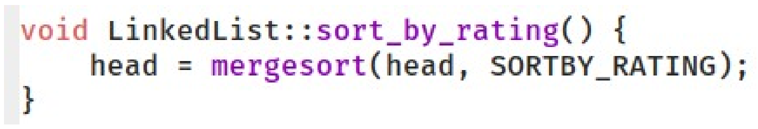
2.3.17 Sort_by_hourly_rate method



```
void LinkedList::sort_by_hourly_rate() {
    head = mergesort(head, SORTBY_HOURLY_RATE);
}
```

Same logic as the *sort_by_id* method above.

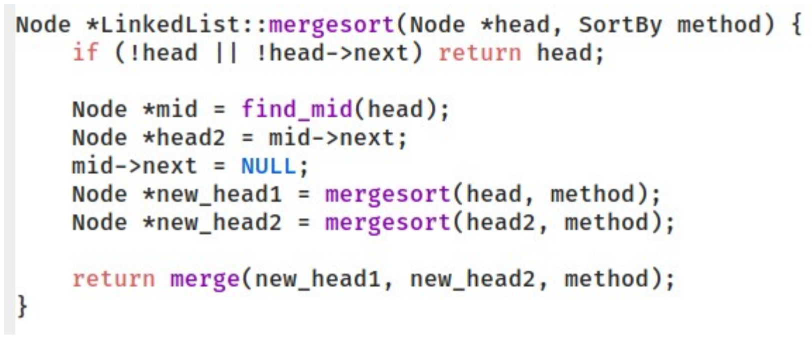
2.3.18 Sort_by_rating method



```
void LinkedList::sort_by_rating() {
    head = mergesort(head, SORTBY_RATING);
}
```

Same logic as the *sort_by* methods above.

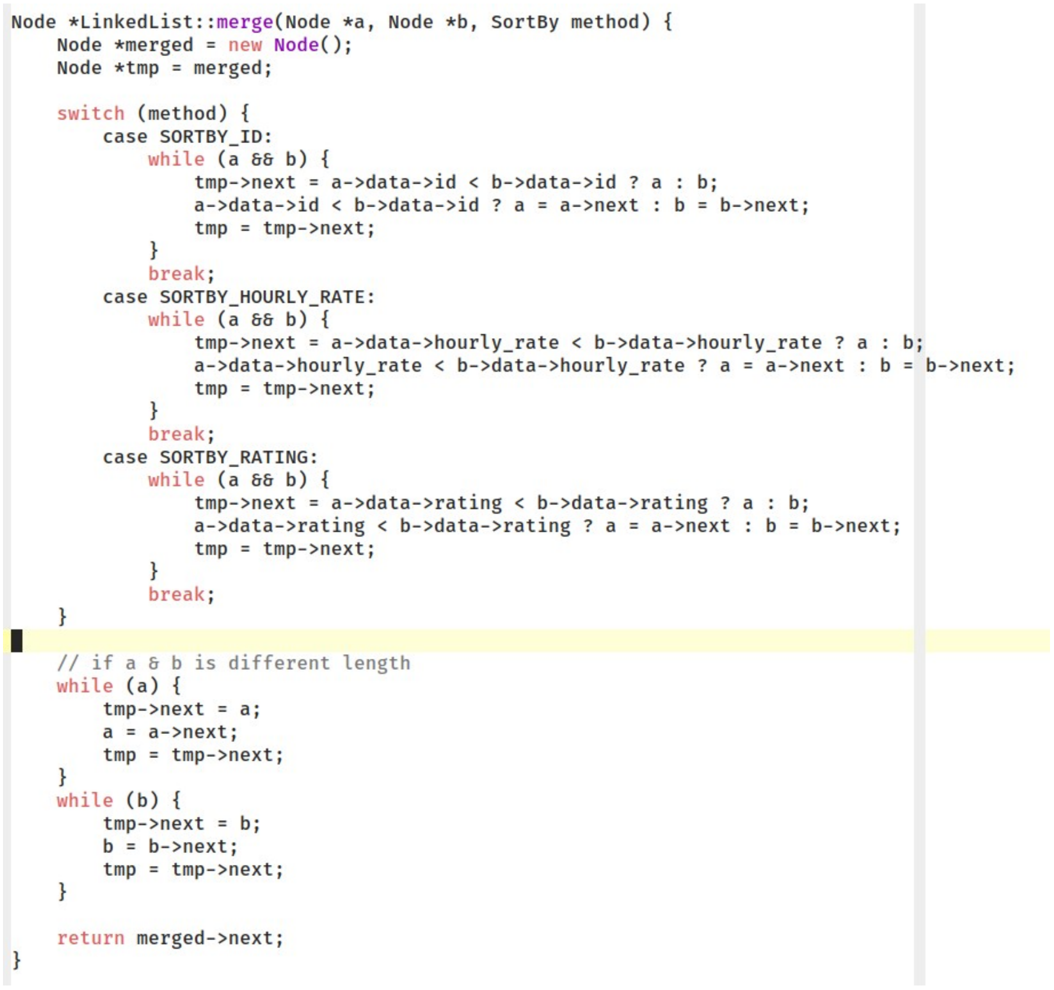
2.3.19 Mergesort method



```
Node *LinkedList::mergesort(Node *head, SortBy method) {
    if (!head || !head->next) return head;

    Node *mid = find_mid(head);
    Node *head2 = mid->next;
    mid->next = NULL;
    Node *new_head1 = mergesort(head, method);
    Node *new_head2 = mergesort(head2, method);

    return merge(new_head1, new_head2, method);
}
```

This method is a recursive method. Firstly, it returns the *head* if it or its next item is NULL. This is used to break out of the recursive calls, and this means that the list has been broken down into just 1 item. Next, find the midpoint of the list using the *find_mid* method. This mid-point will be used to break the list into 2 halves. Then recursively call the *mergesort* method on the 2 halves of the list and store the return values of the 2 method calls to *new_head1* and *new_head2*. Then merge the 2 lists using the *merge* method & return the head of the sorted list.

2.3.20 Merge method



```
Node *LinkedList::merge(Node *a, Node *b, SortBy method) {
    Node *merged = new Node();
    Node *tmp = merged;

    switch (method) {
        case SORTBY_ID:
            while (a && b) {
                tmp->next = a->data->id < b->data->id ? a : b;
                a->data->id < b->data->id ? a = a->next : b = b->next;
                tmp = tmp->next;
            }
            break;
        case SORTBY_HOURLY_RATE:
            while (a && b) {
                tmp->next = a->data->hourly_rate < b->data->hourly_rate ? a : b;
                a->data->hourly_rate < b->data->hourly_rate ? a = a->next : b = b->next;
                tmp = tmp->next;
            }
            break;
        case SORTBY_RATING:
            while (a && b) {
                tmp->next = a->data->rating < b->data->rating ? a : b;
                a->data->rating < b->data->rating ? a = a->next : b = b->next;
                tmp = tmp->next;
            }
            break;
    }

    // if a & b is different length
    while (a) {
        tmp->next = a;
        a = a->next;
        tmp = tmp->next;
    }
    while (b) {
        tmp->next = b;
        b = b->next;
        tmp = tmp->next;
    }

    return merged->next;
}
```

This method iterates over 2 lists and merges them together. The *tmp* variable is used to hold the head of the merged list. It loops over both arrays and adds to *tmp* in order of the smallest property being compared (this can be ID, hourly rate, or rating). Then, because these 2 lists can be of different lengths, some items can be left over. These items will then be inserted into the list, and finally the list is sorted, and the head is returned.

2.3.21 Find_mid method

```
Node *LinkedList::find_mid(Node *head) {
    Node *slow = head, *fast = head->next;
    while (fast && fast->next) {
        slow = slow->next;
        fast = fast->next->next;
    }
    return slow;
}
```

This method is a helper function to find the midpoint of the linked list. It uses 2 pointer variables, *slow* and *fast*. The former iterates over the list 1 step at a time, while the latter iterates 2 steps at a time. When *fast* reaches or passes the end of the list, *slow* will be at the midpoint of the list. Then return *slow*.

2.4 Utility functions

2.4.1 Clear function

```
void clear() {
    cout << "\033[2J\033[1;1H";
}
```

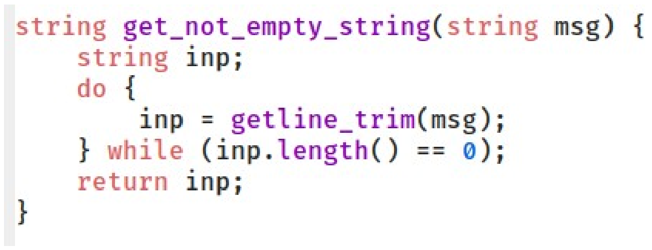
This function clears the screen.

2.4.2 Getline_trim function

```
string getline_trim(string msg) {
    cout << msg;
    string inp;
    getline(cin, inp);
    return regex_replace(inp, regex("(^[ ]+)|([ ]+$)"),"");
}
```

This function takes a string message argument and prints it, then asks for input from the user and returns it with leading and trailing whitespaces removed using regex.

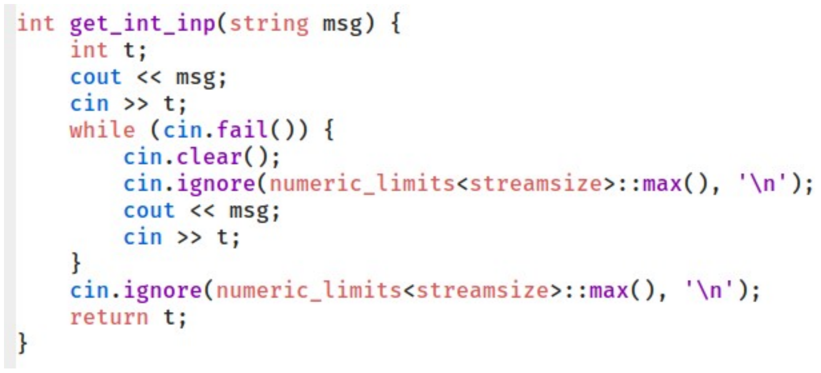
2.4.3 Get_not_empty_string function



```cpp
string get_not_empty_string(string msg) {
    string inp;
    do {
        inp = getline_trim(msg);
    } while (inp.length() == 0);
    return inp;
}
```

Calls *getline_trim* while the string is empty.

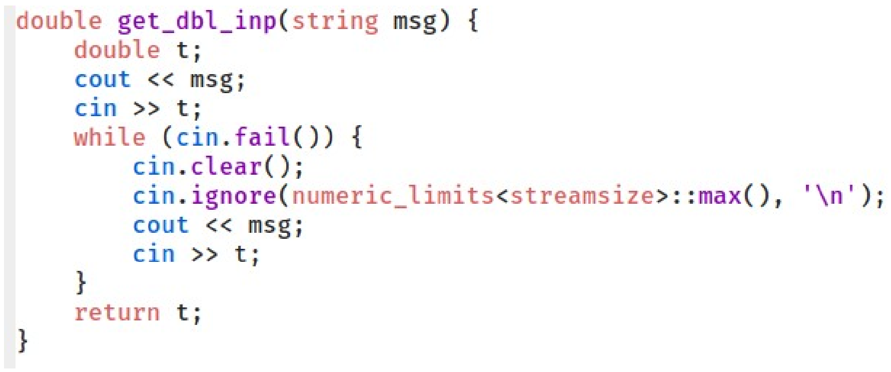
2.4.4 Get_int_inp function



```cpp
int get_int_inp(string msg) {
    int t;
    cout << msg;
    cin >> t;
    while (cin.fail()) {
        cin.clear();
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
        cout << msg;
        cin >> t;
    }
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
    return t;
}
```

Ask the user for integer input and keep trying if the user enters a non-integer input.

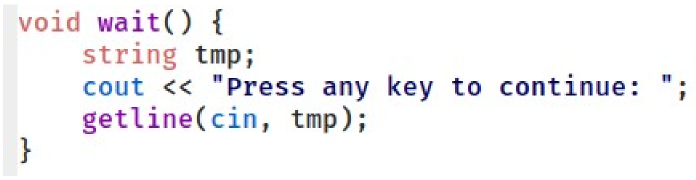
2.4.5 Get_dbl_inp function



```cpp
double get_dbl_inp(string msg) {
    double t;
    cout << msg;
    cin >> t;
    while (cin.fail()) {
        cin.clear();
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
        cout << msg;
        cin >> t;
    }
    return t;
}
```

Same logic as *get_int_inp*.

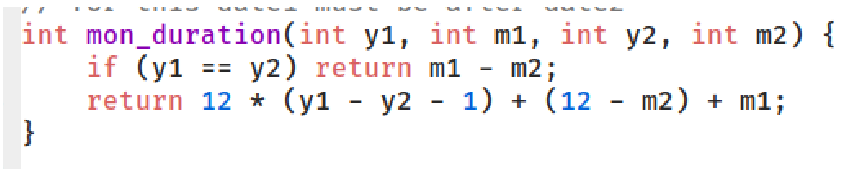
2.4.6 Wait function



```
void wait() {
    string tmp;
    cout << "Press any key to continue: ";
    getline(cin, tmp);
}
```

Prints a message and waits for user input.

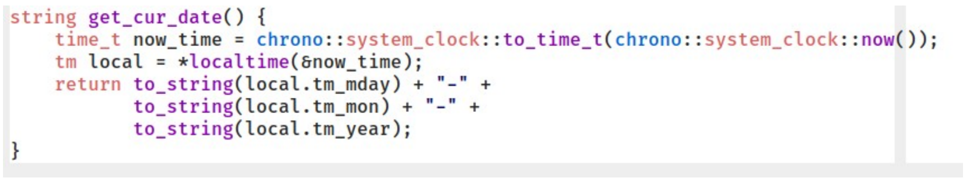
2.4.7 Mon_duration function



```
int mon_duration(int y1, int m1, int y2, int m2) {
    if (y1 == y2) return m1 - m2;
    return 12 * (y1 - y2 - 1) + (12 - m2) + m1;
}
```

Returns the duration in months, 2 dates using their year and month fields. This function has 4 parameters, the first 2 being the year and month of the first date, which needs to be after the second date, and the last 2 being the second date.

2.4.8 Get_cur_date function

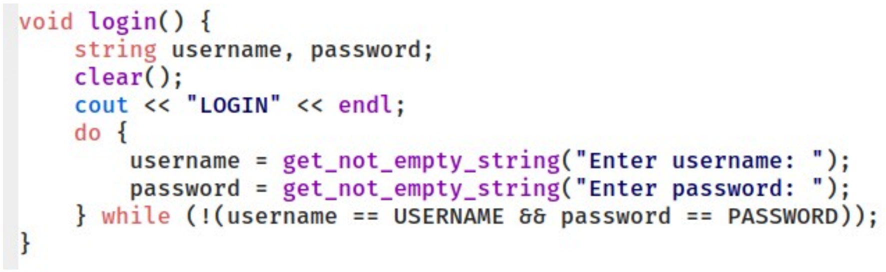


```
string get_cur_date() {
    time_t now_time = chrono::system_clock::to_time_t(chrono::system_clock::now());
    tm local = *localtime(&now_time);
    return to_string(local.tm_mday) + "-" +
           to_string(local.tm_mon) + "-" +
           to_string(local.tm_year);
}
```

Returns the current date in the dd-mm-yyyy format.

2.5 Menu functions

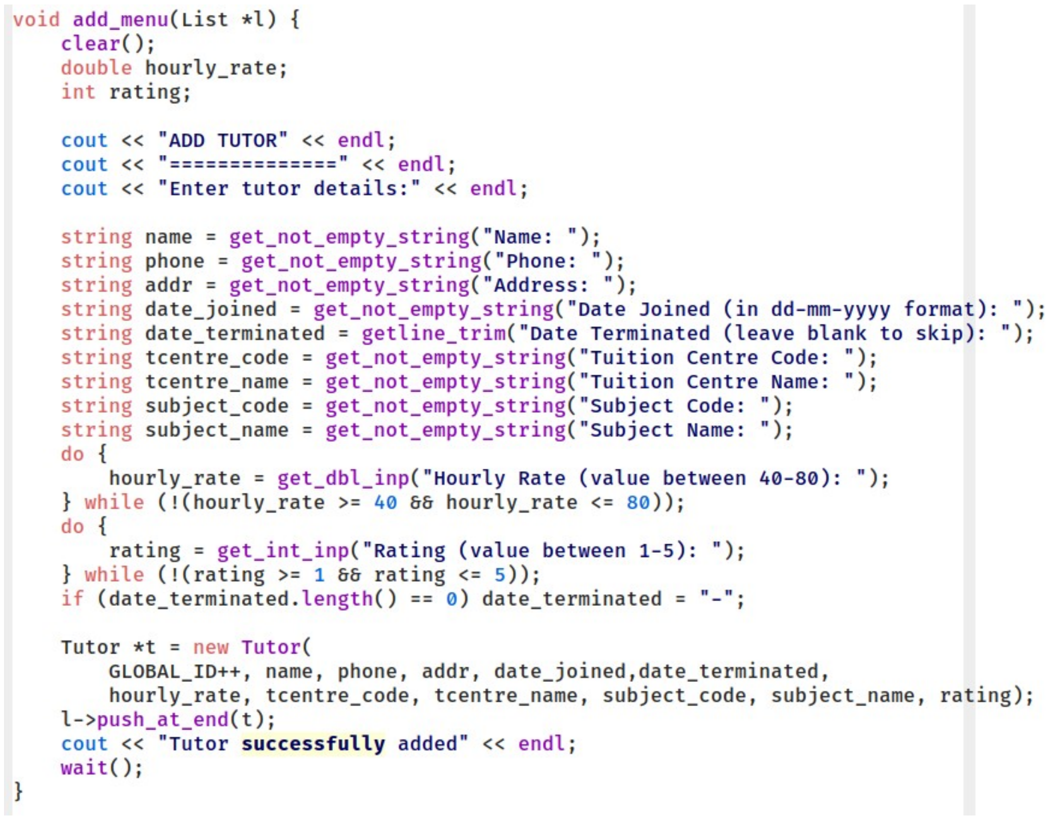
2.5.1 Login function



```
void login() {
    string username, password;
    clear();
    cout << "LOGIN" << endl;
    do {
        username = get_not_empty_string("Enter username: ");
        password = get_not_empty_string("Enter password: ");
    } while (!(username == USERNAME && password == PASSWORD));
}
```

Function for login before users can access the system. Takes input from the user and checks it with the predetermined username and password, which are defined as macro definitions.

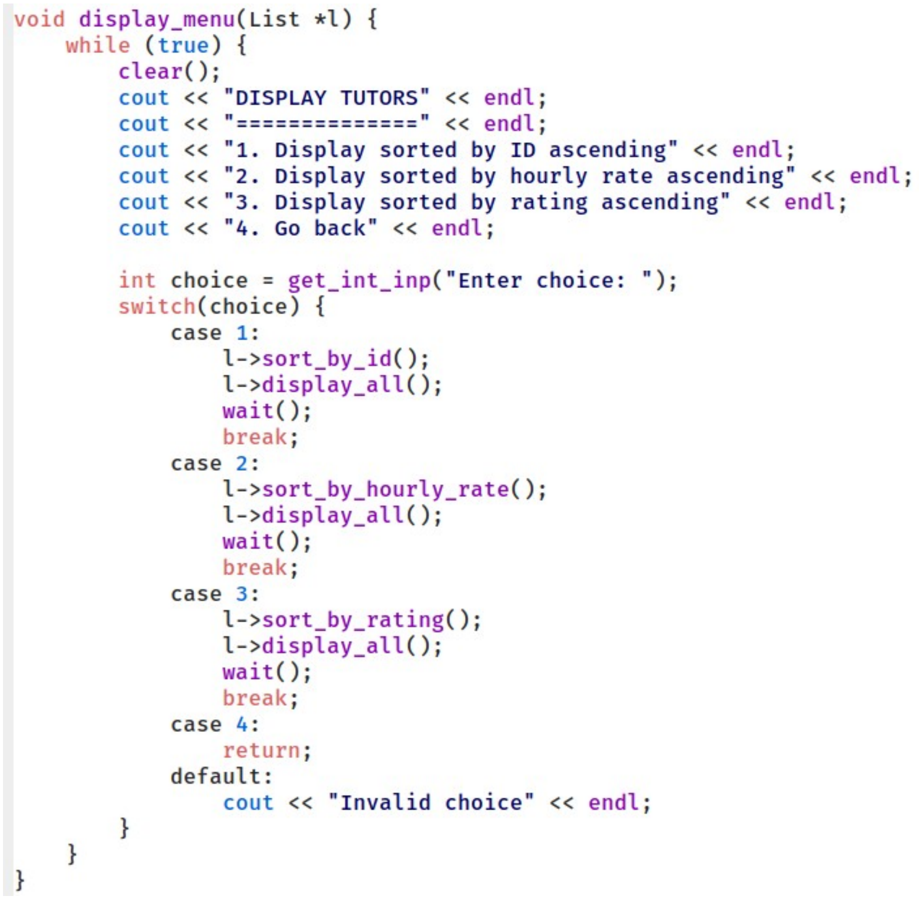
2.5.2 Add_menu function



```
void add_menu(List *l) {
    clear();
    double hourly_rate;
    int rating;

    cout << "ADD TUTOR" << endl;
    cout << "=============" << endl;
    cout << "Enter tutor details:" << endl;

    string name = get_not_empty_string("Name: ");
    string phone = get_not_empty_string("Phone: ");
    string addr = get_not_empty_string("Address: ");
    string date_joined = get_not_empty_string("Date Joined (in dd-mm-yyyy format): ");
    string date_terminated = getline_trim("Date Terminated (leave blank to skip): ");
    string tcentre_code = get_not_empty_string("Tuition Centre Code: ");
    string tcentre_name = get_not_empty_string("Tuition Centre Name: ");
    string subject_code = get_not_empty_string("Subject Code: ");
    string subject_name = get_not_empty_string("Subject Name: ");
    do {
        hourly_rate = get_dbl_inp("Hourly Rate (value between 40-80): ");
    } while (!(hourly_rate >= 40 && hourly_rate <= 80));
    do {
        rating = get_int_inp("Rating (value between 1-5): ");
    } while (!(rating >= 1 && rating <= 5));
    if (date_terminated.length() == 0) date_terminated = "-";

    Tutor *t = new Tutor(
        GLOBAL_ID++, name, phone, addr, date_joined,date_terminated,
        hourly_rate, tcentre_code, tcentre_name, subject_code, subject_name, rating);
    l->push_at_end(t);
    cout << "Tutor successfully added" << endl;
    wait();
}
```

This function asks for tutor fields from the user, creates the *Tutor* object and pushes it to the list, which can either be implemented using an array or a linked list. The *GLOBAL_ID* variable, which holds the value of the id of the next tutor to be added to the list, is also incremented.

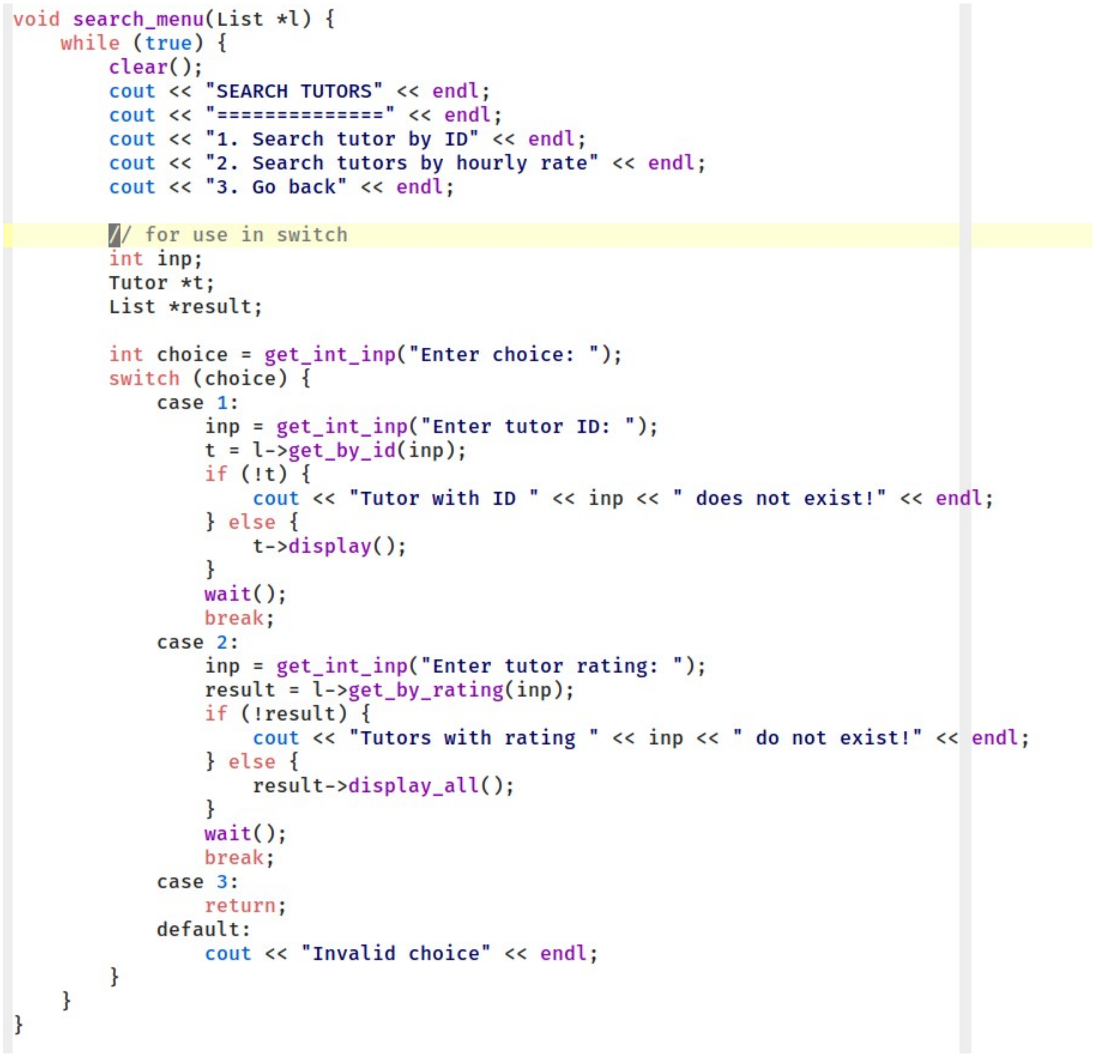
2.5.3 Display_menu function



```
void display_menu(List *l) {
    while (true) {
        clear();
        cout << "DISPLAY TUTORS" << endl;
        cout << "==============" << endl;
        cout << "1. Display sorted by ID ascending" << endl;
        cout << "2. Display sorted by hourly rate ascending" << endl;
        cout << "3. Display sorted by rating ascending" << endl;
        cout << "4. Go back" << endl;

        int choice = get_int_inp("Enter choice: ");
        switch(choice) {
            case 1:
                l->sort_by_id();
                l->display_all();
                wait();
                break;
            case 2:
                l->sort_by_hourly_rate();
                l->display_all();
                wait();
                break;
            case 3:
                l->sort_by_rating();
                l->display_all();
                wait();
                break;
            case 4:
                return;
            default:
                cout << "Invalid choice" << endl;
        }
    }
}
```

Function to display all the tutors in the list. Sorting also happens here, as users can choose how the list will be sorted.

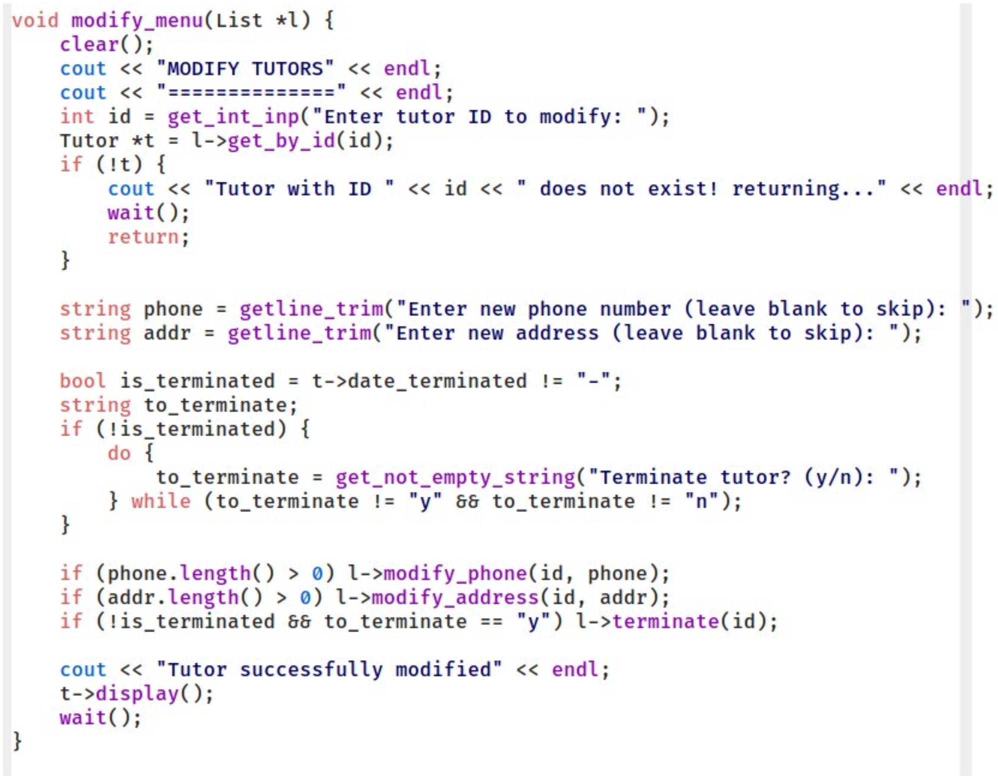
2.5.4 Search_menu function



```
void search_menu(List *l) {
    while (true) {
        clear();
        cout << "SEARCH TUTORS" << endl;
        cout << "=============" << endl;
        cout << "1. Search tutor by ID" << endl;
        cout << "2. Search tutors by hourly rate" << endl;
        cout << "3. Go back" << endl;

        // for use in switch
        int inp;
        Tutor *t;
        List *result;

        int choice = get_int_inp("Enter choice: ");
        switch (choice) {
            case 1:
                inp = get_int_inp("Enter tutor ID: ");
                t = l->get_by_id(inp);
                if (!t) {
                    cout << "Tutor with ID " << inp << " does not exist!" << endl;
                } else {
                    t->display();
                }
                wait();
                break;
            case 2:
                inp = get_int_inp("Enter tutor rating: ");
                result = l->get_by_rating(inp);
                if (!result) {
                    cout << "Tutors with rating " << inp << " do not exist!" << endl;
                } else {
                    result->display_all();
                }
                wait();
                break;
            case 3:
                return;
            default:
                cout << "Invalid choice" << endl;
        }
    }
}
```

This function allows users to search for tutors using either their ID or rating. If no tutor with the specified ID or rating is found, a message will be printed out.

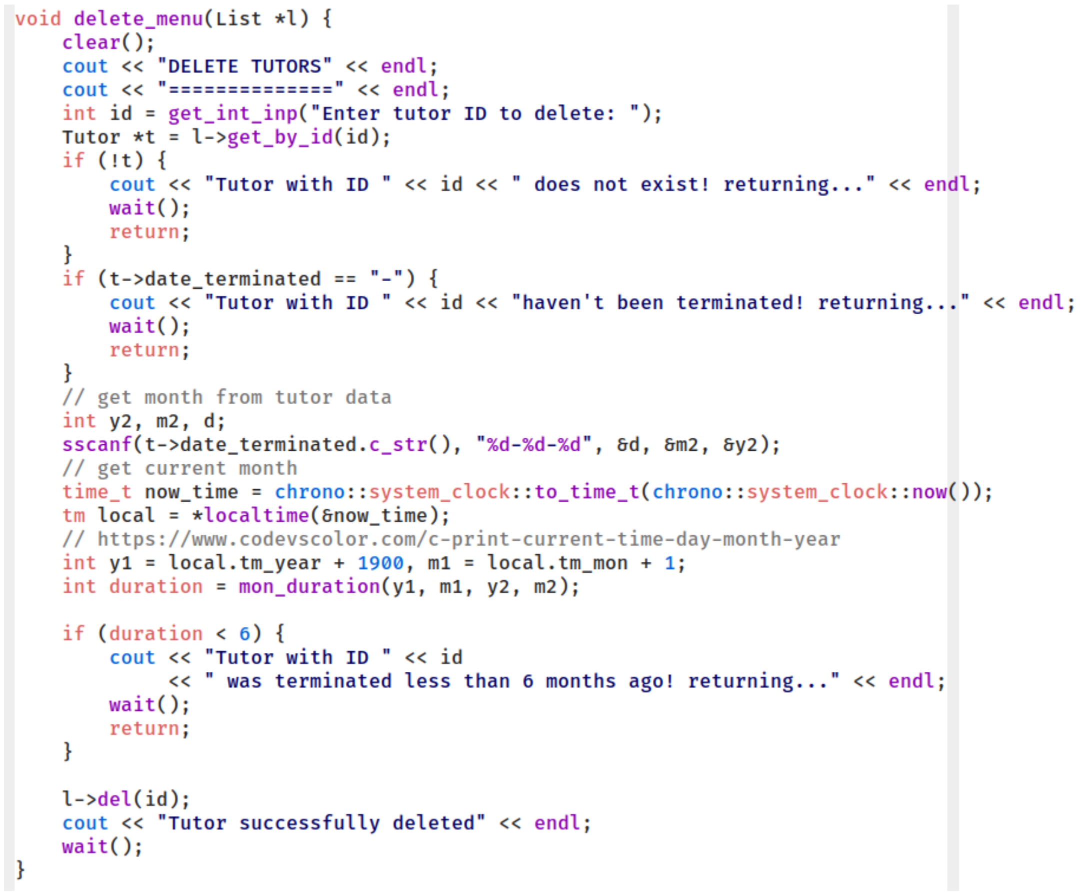
2.5.5 Modify_menu function



```cpp
void modify_menu(List *l) {
    clear();
    cout << "MODIFY TUTORS" << endl;
    cout << "=============" << endl;
    int id = get_int_inp("Enter tutor ID to modify: ");
    Tutor *t = l->get_by_id(id);
    if (!t) {
        cout << "Tutor with ID " << id << " does not exist! returning..." << endl;
        wait();
        return;
    }

    string phone = getline_trim("Enter new phone number (leave blank to skip): ");
    string addr = getline_trim("Enter new address (leave blank to skip): ");

    bool is_terminated = t->date_terminated != "-";
    string to_terminate;
    if (!is_terminated) {
        do {
            to_terminate = get_not_empty_string("Terminate tutor? (y/n): ");
        } while (to_terminate != "y" && to_terminate != "n");
    }

    if (phone.length() > 0) l->modify_phone(id, phone);
    if (addr.length() > 0) l->modify_address(id, addr);
    if (!is_terminated && to_terminate == "y") l->terminate(id);

    cout << "Tutor successfully modified" << endl;
    t->display();
    wait();
}
```

This function allows users to modify the phone number or address of tutors, and to terminate them based on their ID. If no tutor with the specified ID is found, it prints a message and returns. If the input given is blank, then the modification of the blank property (either *phone* or *address*) will not be done. If the tutor is already terminated, then it will not prompt the user for it.

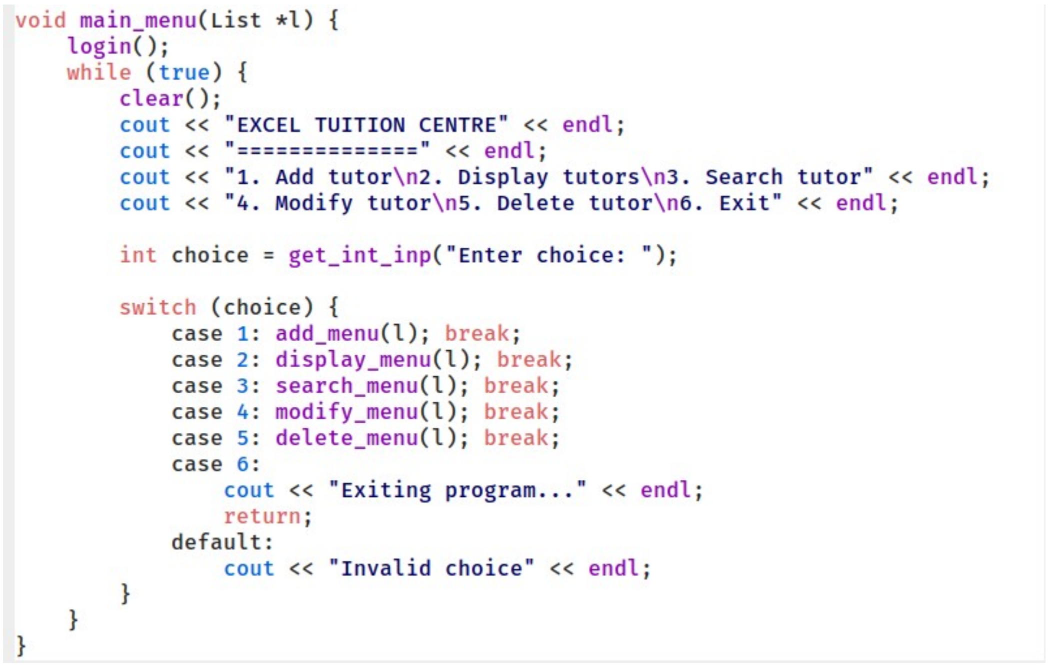
2.5.6 Delete_menu function



```
void delete_menu(List *l) {
    clear();
    cout << "DELETE TUTORS" << endl;
    cout << "=============" << endl;
    int id = get_int_inp("Enter tutor ID to delete: ");
    Tutor *t = l->get_by_id(id);
    if (!t) {
        cout << "Tutor with ID " << id << " does not exist! returning..." << endl;
        wait();
        return;
    }
    if (t->date_terminated == "-") {
        cout << "Tutor with ID " << id << "haven't been terminated! returning..." << endl;
        wait();
        return;
    }
    // get month from tutor data
    int y2, m2, d;
    sscanf(t->date_terminated.c_str(), "%d-%d-%d", &d, &m2, &y2);
    // get current month
    time_t now_time = chrono::system_clock::to_time_t(chrono::system_clock::now());
    tm local = *localtime(&now_time);
    // https://www.codevscolor.com/c-print-current-time-day-month-year
    int y1 = local.tm_year + 1900, m1 = local.tm_mon + 1;
    int duration = mon_duration(y1, m1, y2, m2);

    if (duration < 6) {
        cout << "Tutor with ID " << id
             << " was terminated less than 6 months ago! returning..." << endl;
        wait();
        return;
    }

    l->del(id);
    cout << "Tutor successfully deleted" << endl;
    wait();
}
```

This function asks for the ID of the tutor to be deleted. It goes through several validations before the actual deletion. It checks if the tutor with the inputted ID exists, then checks if they have been terminated, and finally checks if it has been at least 6 months after the termination. After all checks have been done, the record will be deleted from the list.

2.5.7 Main_menu function

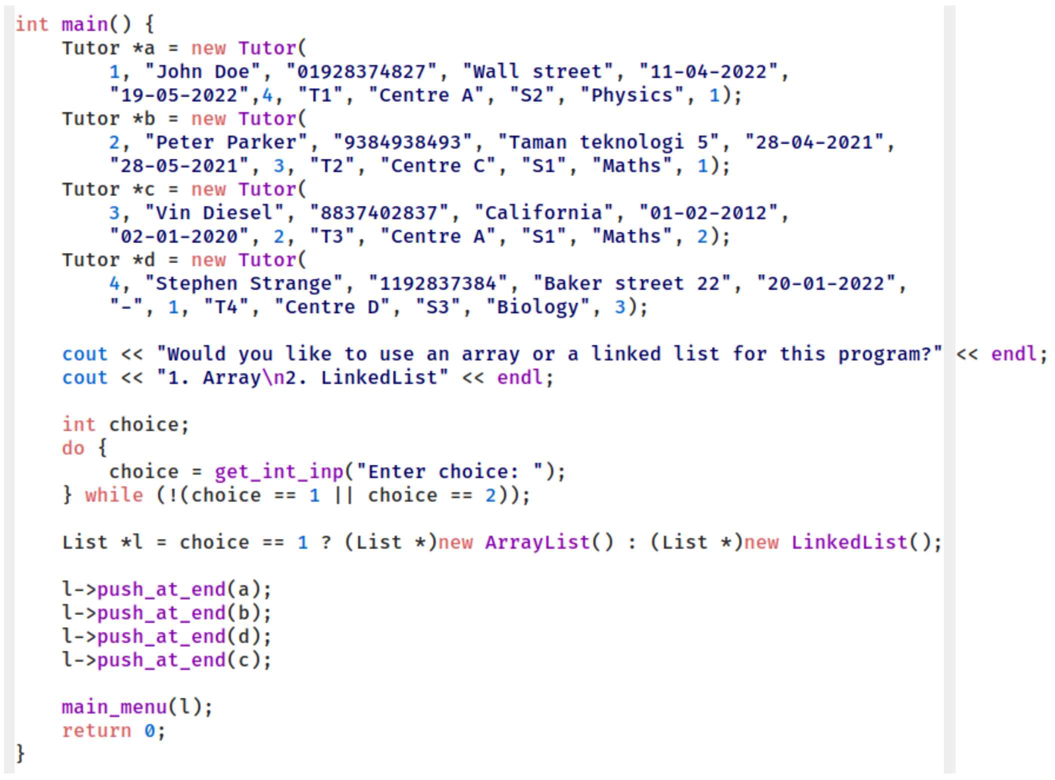


```
void main_menu(List *l) {
    login();
    while (true) {
        clear();
        cout << "EXCEL TUITION CENTRE" << endl;
        cout << "=============" << endl;
        cout << "1. Add tutor\n2. Display tutors\n3. Search tutor" << endl;
        cout << "4. Modify tutor\n5. Delete tutor\n6. Exit" << endl;

        int choice = get_int_inp("Enter choice: ");

        switch (choice) {
            case 1: add_menu(l); break;
            case 2: display_menu(l); break;
            case 3: search_menu(l); break;
            case 4: modify_menu(l); break;
            case 5: delete_menu(l); break;
            case 6:
                cout << "Exiting program..." << endl;
                return;
            default:
                cout << "Invalid choice" << endl;
        }
    }
}
```

The main menu asks for user input and depending on the value, calls the corresponding functions mentioned earlier in this section.

2.6 Program entry point

2.6.1 The main function



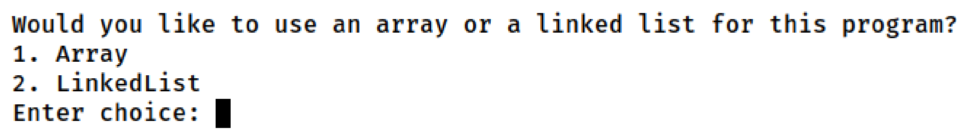
```cpp
int main() {
    Tutor *a = new Tutor(
        1, "John Doe", "01928374827", "Wall street", "11-04-2022",
        "19-05-2022",4, "T1", "Centre A", "S2", "Physics", 1);
    Tutor *b = new Tutor(
        2, "Peter Parker", "9384938493", "Taman teknologi 5", "28-04-2021",
        "28-05-2021", 3, "T2", "Centre C", "S1", "Maths", 1);
    Tutor *c = new Tutor(
        3, "Vin Diesel", "8837402837", "California", "01-02-2012",
        "02-01-2020", 2, "T3", "Centre A", "S1", "Maths", 2);
    Tutor *d = new Tutor(
        4, "Stephen Strange", "1192837384", "Baker street 22", "20-01-2022",
        "-", 1, "T4", "Centre D", "S3", "Biology", 3);

    cout << "Would you like to use an array or a linked list for this program?" << endl;
    cout << "1. Array\n2. LinkedList" << endl;

    int choice;
    do {
        choice = get_int_inp("Enter choice: ");
    } while (!(choice == 1 || choice == 2));

    List *l = choice == 1 ? (List *)new ArrayList() : (List *)new LinkedList();

    l->push_at_end(a);
    l->push_at_end(b);
    l->push_at_end(d);
    l->push_at_end(c);

    main_menu(l);
    return 0;
}
```

The main function creates the dummy tutor data for the list, then asks the user whether they want to use an array or a linked list for the program, then creates the list using the appropriate class based on the user input. Then it pushes the tutors into the list and calls the *main_menu* function.

3.0 Results

3.1 System I/O Screenshots

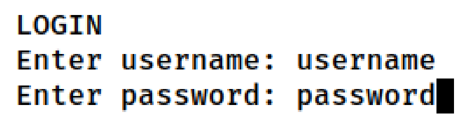
3.1.1 Choosing Data Structure



```
Would you like to use an array or a linked list for this program?
1. Array
2. LinkedList
Enter choice: █
```

The first screen shown to the user after running the program is the option to choose which data structure to use for the currently running program. The functionalities for both data structures will be the same, and the differences will be only minor (such as speed of execution) and almost unnoticeable to the user.

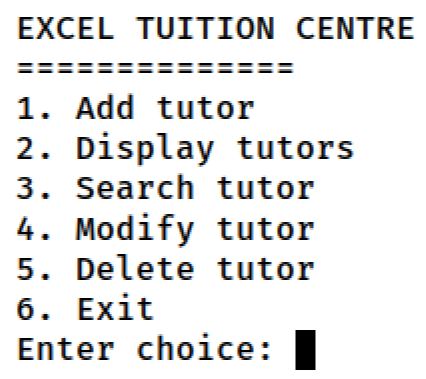
3.1.2 User Login



```
LOGIN
Enter username: username
Enter password: password
```

After choosing the data structure, the user will be presented with the login screen where they would be presented with the username & password prompt. The program will keep asking until the correct username & password is given.

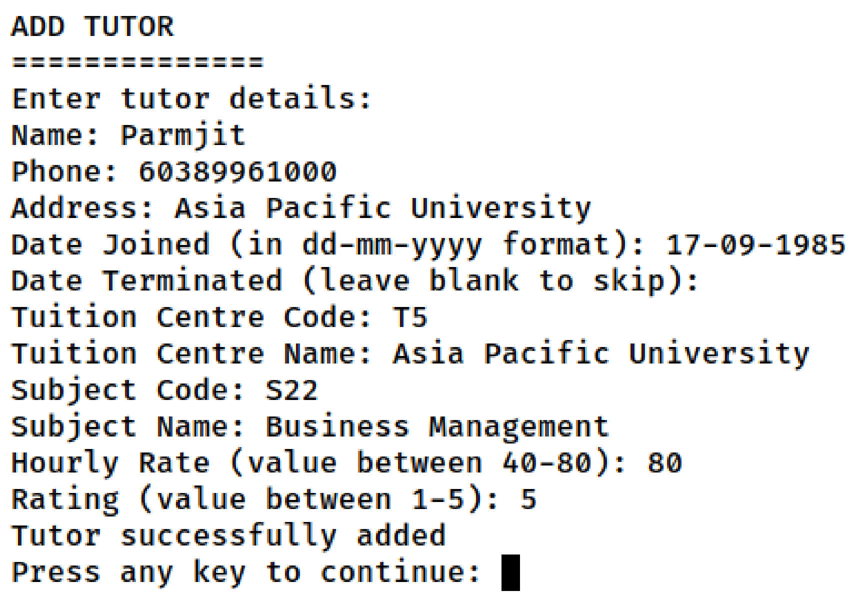
3.1.3 Main Menu



```
EXCEL TUITION CENTRE
==============
1. Add tutor
2. Display tutors
3. Search tutor
4. Modify tutor
5. Delete tutor
6. Exit
Enter choice: 
```

This is the main menu shown after a successful login. They will be prompted for a choice and based on the input they will be shown different screens.

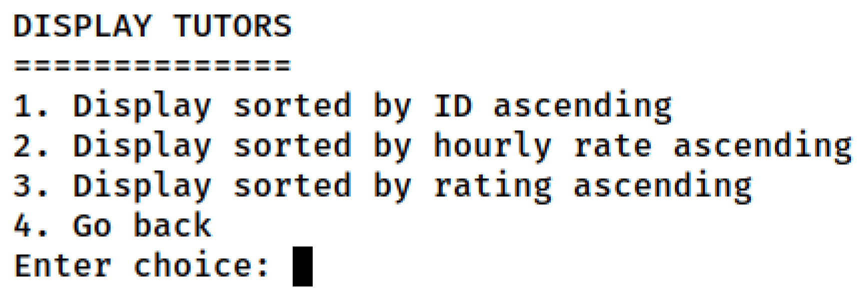
3.1.4 Add Tutor Menu



```
ADD TUTOR
==============
Enter tutor details:
Name: Parmjit
Phone: 60389961000
Address: Asia Pacific University
Date Joined (in dd-mm-yyyy format): 17-09-1985
Date Terminated (leave blank to skip):
Tuition Centre Code: T5
Tuition Centre Name: Asia Pacific University
Subject Code: S22
Subject Name: Business Management
Hourly Rate (value between 40-80): 80
Rating (value between 1-5): 5
Tutor successfully added
Press any key to continue:
```

After inputting 1 from the main menu, this menu will be shown. Users are prompted for the details of the tutor. Some fields can be left blank, such as the date terminated field. Some fields have validation such as the hourly rate & rating field. The rest of the fields only have validation for empty or whitespace only strings. Fields like phone number validation and date format validation are left out for the sake of simplicity of the program.

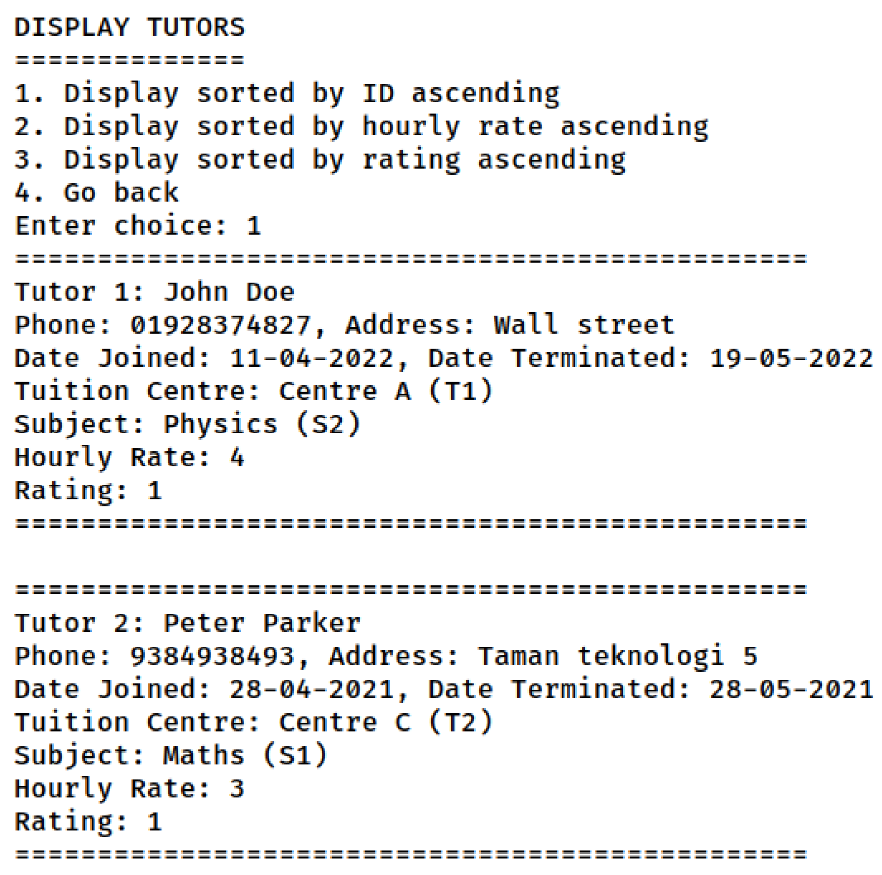
3.1.5 Display Tutors Menu



```
DISPLAY TUTORS
==============
1. Display sorted by ID ascending
2. Display sorted by hourly rate ascending
3. Display sorted by rating ascending
4. Go back
Enter choice:
```

Inputting 2 from the main menu leads to the display tutors menu. Here, users are given options to display the tutors sorted ascendingly either by tutor ID, hourly rate, or rating. They also have the choice of going back to the main menu.

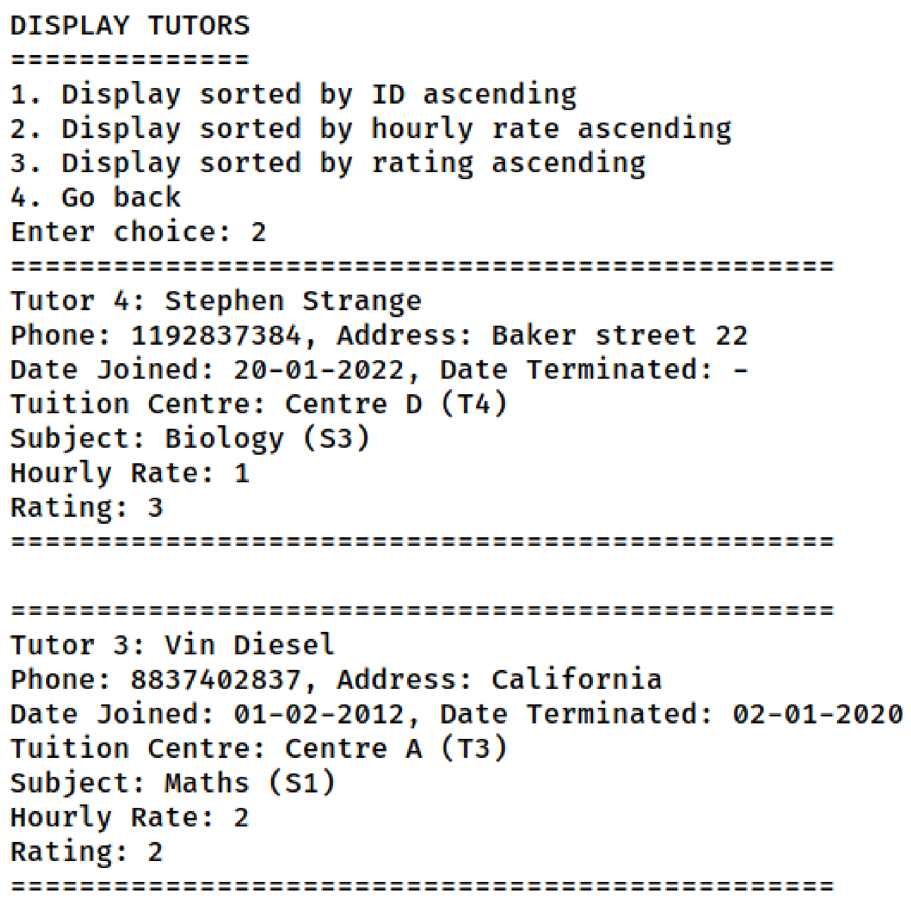
3.1.6 Tutors Sorted by ID



```
DISPLAY TUTORS
==============
1. Display sorted by ID ascending
2. Display sorted by hourly rate ascending
3. Display sorted by rating ascending
4. Go back
Enter choice: 1
================================================
Tutor 1: John Doe
Phone: 01928374827, Address: Wall street
Date Joined: 11-04-2022, Date Terminated: 19-05-2022
Tuition Centre: Centre A (T1)
Subject: Physics (S2)
Hourly Rate: 4
Rating: 1
================================================

================================================
Tutor 2: Peter Parker
Phone: 9384938493, Address: Taman teknologi 5
Date Joined: 28-04-2021, Date Terminated: 28-05-2021
Tuition Centre: Centre C (T2)
Subject: Maths (S1)
Hourly Rate: 3
Rating: 1
================================================
```

The screenshot above is cut to avoid it being too long. Here the list of tutors will be printed with all their details.

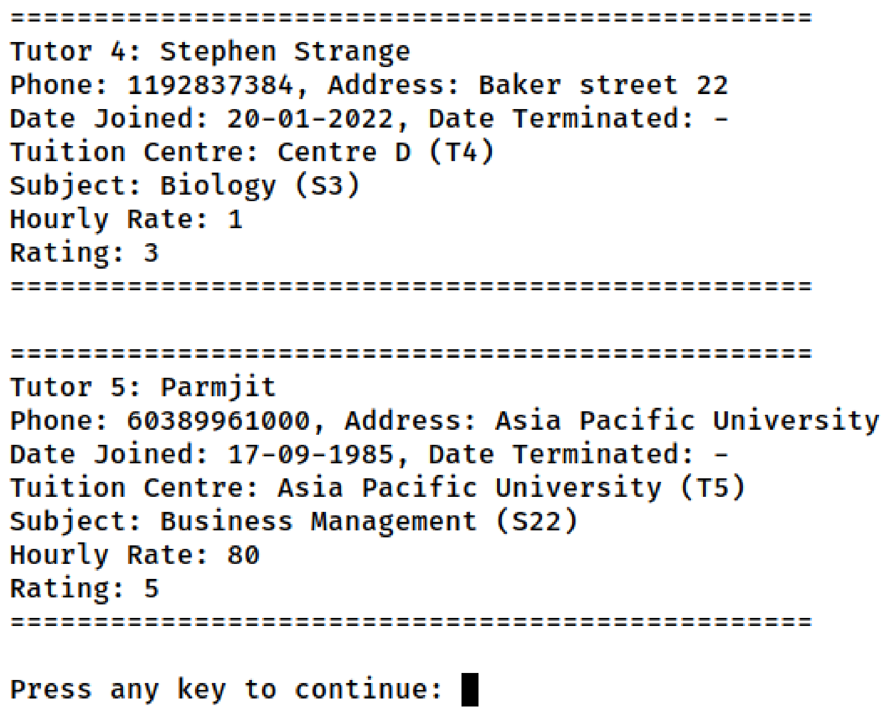
3.1.7 Tutors Sorted by Hourly Rate



```
DISPLAY TUTORS
==============
1. Display sorted by ID ascending
2. Display sorted by hourly rate ascending
3. Display sorted by rating ascending
4. Go back
Enter choice: 2
================================================
Tutor 4: Stephen Strange
Phone: 1192837384, Address: Baker street 22
Date Joined: 20-01-2022, Date Terminated: -
Tuition Centre: Centre D (T4)
Subject: Biology (S3)
Hourly Rate: 1
Rating: 3
================================================

================================================
Tutor 3: Vin Diesel
Phone: 8837402837, Address: California
Date Joined: 01-02-2012, Date Terminated: 02-01-2020
Tuition Centre: Centre A (T3)
Subject: Maths (S1)
Hourly Rate: 2
Rating: 2
================================================
```

Same as the previous screen, the structure of the display screen will be exactly the same, with the only difference being the order of the tutors, which in this case is being ordered by the hourly rate in ascending order.

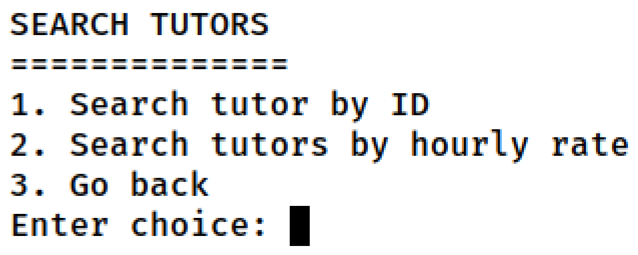
3.1.8 Tutors Sorted by Rating



```
================================================
Tutor 4: Stephen Strange
Phone: 1192837384, Address: Baker street 22
Date Joined: 20-01-2022, Date Terminated: -
Tuition Centre: Centre D (T4)
Subject: Biology (S3)
Hourly Rate: 1
Rating: 3
================================================

================================================
Tutor 5: Parmjit
Phone: 60389961000, Address: Asia Pacific University
Date Joined: 17-09-1985, Date Terminated: -
Tuition Centre: Asia Pacific University (T5)
Subject: Business Management (S22)
Hourly Rate: 80
Rating: 5
================================================

Press any key to continue:
```

This time the end of the screen will be shown, with the prompt to continue with other operations. It is also to show the tutor being added during the add menu screen.

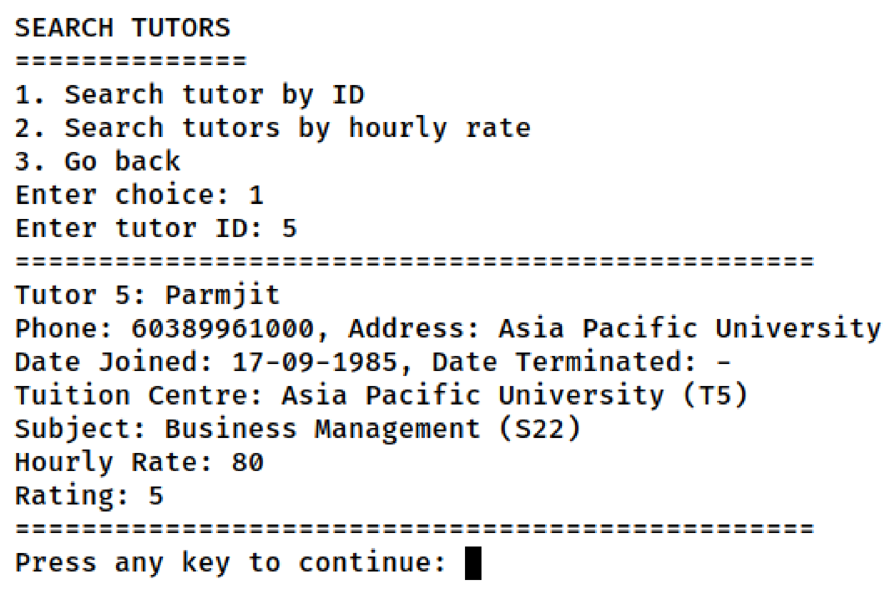
3.1.9 Search Tutors Menu



```
SEARCH TUTORS
==============
1. Search tutor by ID
2. Search tutors by hourly rate
3. Go back
Enter choice:
```

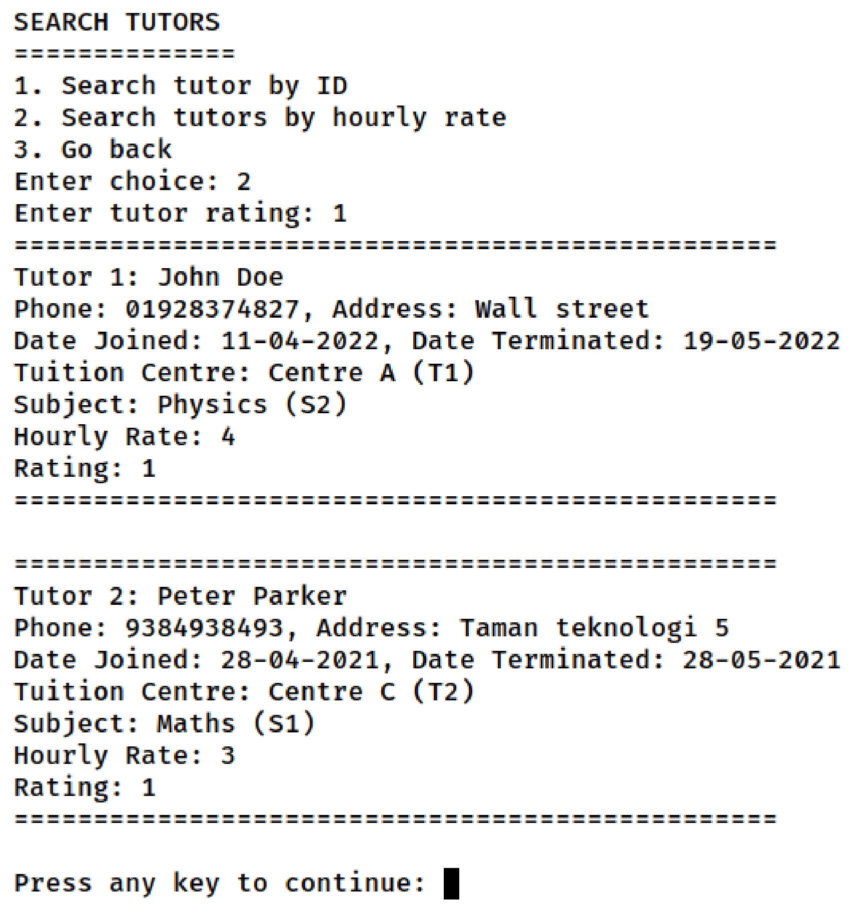
This menu is shown after inputting 3 in the main menu. The user is given 3 options, whether to search for tutors by ID or by hourly rate, and the option to go back to the previous screen.

3.1.10 Search Tutor by ID


```
SEARCH TUTORS
==============
1. Search tutor by ID
2. Search tutors by hourly rate
3. Go back
Enter choice: 1
Enter tutor ID: 5
================================================
Tutor 5: Parmjit
Phone: 60389961000, Address: Asia Pacific University
Date Joined: 17-09-1985, Date Terminated: -
Tuition Centre: Asia Pacific University (T5)
Subject: Business Management (S22)
Hourly Rate: 80
Rating: 5
================================================
Press any key to continue:
```

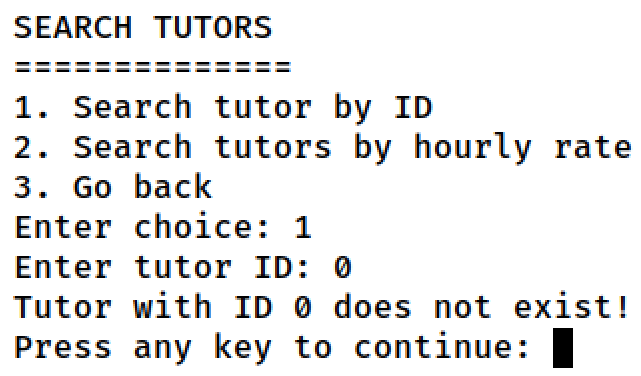
After inputting 1, the user will be prompted for the ID of the tutor to find. Inputting the correct one will display the tutor with the correct ID.

3.1.11 Search Tutors by Hourly Rate


```
SEARCH TUTORS
==============
1. Search tutor by ID
2. Search tutors by hourly rate
3. Go back
Enter choice: 2
Enter tutor rating: 1
================================================
Tutor 1: John Doe
Phone: 01928374827, Address: Wall street
Date Joined: 11-04-2022, Date Terminated: 19-05-2022
Tuition Centre: Centre A (T1)
Subject: Physics (S2)
Hourly Rate: 4
Rating: 1
================================================

================================================
Tutor 2: Peter Parker
Phone: 9384938493, Address: Taman teknologi 5
Date Joined: 28-04-2021, Date Terminated: 28-05-2021
Tuition Centre: Centre C (T2)
Subject: Maths (S1)
Hourly Rate: 3
Rating: 1
================================================

Press any key to continue:
```

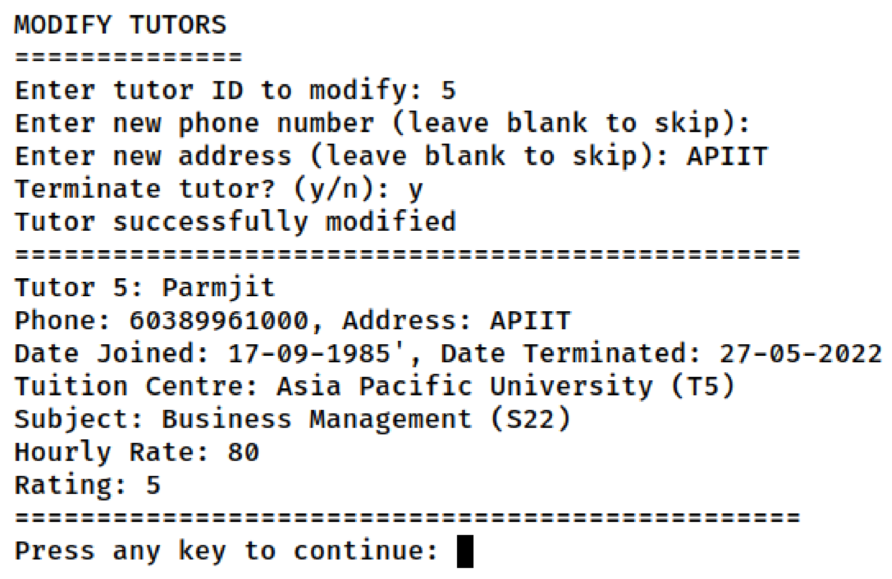
The same goes for the 2nd option, but as multiple tutors can have the same rating, multiple results might be shown.

3.1.12 Search for Invalid Tutor


```
SEARCH TUTORS
==============
1. Search tutor by ID
2. Search tutors by hourly rate
3. Go back
Enter choice: 1
Enter tutor ID: 0
Tutor with ID 0 does not exist!
Press any key to continue:
```

An invalid ID for both options will result in this message being printed.

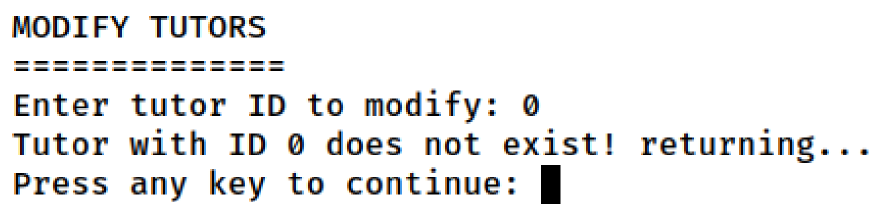
3.1.13 Modify Tutor Menu



```
MODIFY TUTORS
==============
Enter tutor ID to modify: 5
Enter new phone number (leave blank to skip):
Enter new address (leave blank to skip): APIIT
Terminate tutor? (y/n): y
Tutor successfully modified
================================================
Tutor 5: Parmjit
Phone: 60389961000, Address: APIIT
Date Joined: 17-09-1985', Date Terminated: 27-05-2022
Tuition Centre: Asia Pacific University (T5)
Subject: Business Management (S22)
Hourly Rate: 80
Rating: 5
================================================
Press any key to continue:
```

Inputting 4 in the main menu will lead to this screen, where the user is prompted for the ID of the tutor to modify. The fields can be left blank which results in no modification of the field. If the tutor has been terminated, then the terminate option will not be shown.

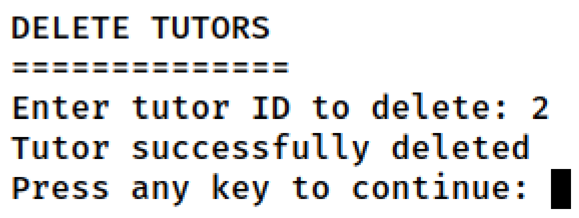
3.1.14 Modify Invalid Tutor



```
MODIFY TUTORS
==============
Enter tutor ID to modify: 0
Tutor with ID 0 does not exist! returning...
Press any key to continue:
```

Again, entering an invalid ID will result in this message being printed.

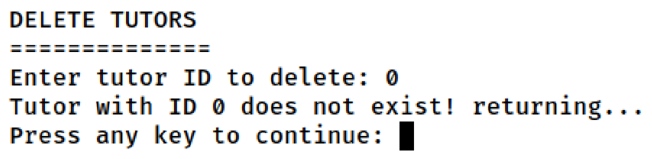
3.1.15 Delete Tutor Menu



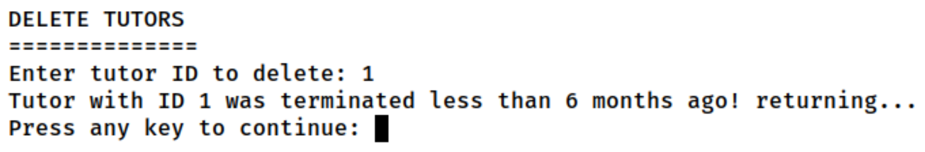
```
DELETE TUTORS
==============
Enter tutor ID to delete: 2
Tutor successfully deleted
Press any key to continue: █
```

Entering 5 in the main menu will bring up this menu. It will prompt the user for the ID to tutor to delete and print out the message above if the operation is successful.

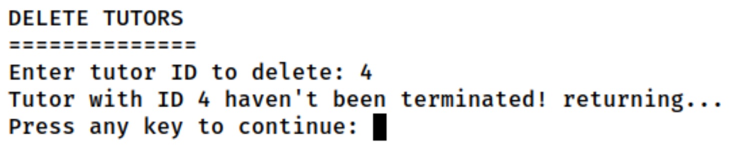
3.1.16 Delete Invalid Tutor



```
DELETE TUTORS
==============
Enter tutor ID to delete: 0
Tutor with ID 0 does not exist! returning...
Press any key to continue: █
```



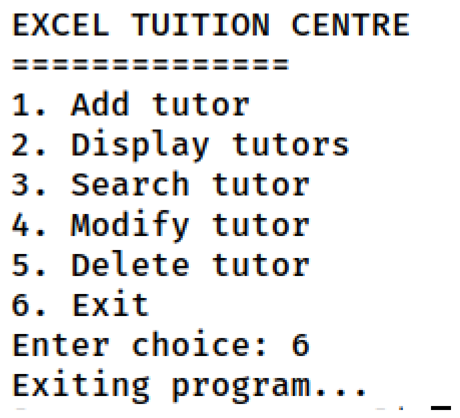
```
DELETE TUTORS
==============
Enter tutor ID to delete: 1
Tutor with ID 1 was terminated less than 6 months ago! returning...
Press any key to continue: █
```



```
DELETE TUTORS
==============
Enter tutor ID to delete: 4
Tutor with ID 4 haven't been terminated! returning...
Press any key to continue: █
```

The above screenshots are the different invalid tutors for deletion.

3.1.17 Exiting the Program



```
EXCEL TUITION CENTRE
==============
1. Add tutor
2. Display tutors
3. Search tutor
4. Modify tutor
5. Delete tutor
6. Exit
Enter choice: 6
Exiting program...
```

Finally, inputting 6 will terminate the program.

3.2 System Evaluation

In this system evaluation, the strengths and weaknesses of this system will be discussed briefly. The effects of the different data structures will not be discussed here as they have been discussed in the earlier sections of this document, as well as in the proposal for this system. One of the strengths of this approach to the program is its flexibility. As the data structures have a common interface to implement, which allows for other classes to be substituted in the program without changing any of code besides the main function.

An example of this is if a doubly linked list were to be added to the system, the only thing the programmer need to do is create the class to implement the interface and change a little bit of code in the main function to allow the user to select it when running the program. The new class, however, needs to implement all the virtual functions of the *List* class for it to work.

A weakness of this approach is the simplicity of the program. Simplicity here means barebones or lacking some features. The features lacking include validations on more user inputs, not having persistent data storage, and options to limit the number of tutors being displayed at a time. For this assignment, the program is kept simple as the focus is on the data structures and algorithms used, but if the system is to be used in the real world, then the missing features would need to be implemented.

4.0 Conclusion

To summarize the system, it is a highly simplified tutor management system that matches the system requirements as per the assignment question document. The user can use both required data structures and complete the necessary tasks of the system. However, several features have been left out to keep the program simple. Also, some functionalities of the system are being limited, such as the search and modify functions, where only certain attributes of the tutor can be used for searching and only certain attributes can be modified.

Future improvements for this system include implementing all the missing features mentioned in this and the previous section. Also, another possible improvement for a better user experience is to implement the system with a graphical user interface (GUI). As a team, we think that the assignment has sufficient hard requirements to fulfill that allows for the students to understand and implement the different aspects of the array and linked list data structure and algorithm that can be used in a real system.

5.0 References

Jaeger, C. (2008, September 10). *When should you use a class vs a struct in C++?* Stack Overflow. Retrieved May 27, 2022, from https://stackoverflow.com/questions/54585/when-should-you-use-a-class-vs-a-struct-in-c

Uri. (2010, May 19). *What is the definition of "interface" in object oriented programming*. Stack Overflow. Retrieved May 27, 2022, from https://stackoverflow.com/questions/2866987/what-is-the-definition-of-interface-in-object-oriented-programming#2867064

6.0 Appendix

Name	TP Number	Tasks	Signature
Ryan Martin	TP058091	Data structures explanation Sorting methods Menu functions Screenshots	[illegible]
Matthew Vincent Gunawan	TP059019	Introduction Tutor structure Insertion & deletion methods System evaluation	Matt
Selvan Nicholas	TP058084	Display & searching methods Utility functions Results explanation Conclusion	[illegible]