Adama Science and Technology University

School of Electrical Engineering and Computing (SoEEC)

Department of computer science and engineering

Data Structures and Algorithms mini project

Group members	ID Number
1. Amanuel	UGR/
2. Aron Teklu	UGR/25406/14
3. Biruk Abebe	UGR/25917/14
4. Rahel Abu	UGR/25554/14
5.	

Submitted to: Mr. Melkamu A.
Submission Date: Thursday, April 27, 2023

Introduction

The ASTU bookstore is the place where we are able to borrow books without any payment as long as we return it with no physical harm caused to it. This bookstore used to be managed by using handwritten files and papers. Although this can be efficient for small number of users, it is known that ASTU has a lot of students who can make use of the bookstore to get their hands on books. This project work aims to provide an efficient algorithm to track students, their IDs and the book they borrowed. It also helps ease the paperwork by adding another element to the code that helps the bookstore workers to register and track incoming books. The program provides a feature that registers books with their names, IDs and authors' names. With this all in mind though, this project is a console application and needs a considerable amount of effort to export this into a proper application format with a working UI. What we have tried to explain here is just the algorithm and the driver code that helps this program to run. Keep in mind we are students and we may make mistakes in the making of this program so we kindly request to be excused for minor inconveniences.

Objective: To help digitalize the current bookkeeping system in ASTU to make it more efficient and easy to handle.

Hardware and Software requirements: This project is very lightweight so it doesn't require much computing power. But it is important to note that as the number of the users increase, lags may occur depending on the amount of storage present on the machine. Other factors such as RAM and CPU are also key elements in helping this algorithm run smoothly.

Regarding the software requirement, we need a C++ code compiler to be able to use this file as it is still a console application. We have tried to cover this in the introduction part and we mention it here again for good measure.

1. In this section we describe how the student registering mechanism works. We define a structure that holds a node that contains the student's name, ID and the book they borrowed. We have functions to add and delete students at the front, at the middle (positions are obtained from user input) and at the end. We also have functions to display all existing students in the system. Finally, we have a function to display and delete students by ID numbers. We have commented on the code in a pretty straightforward way such that one can read and understand the code without much issues. As for the algorithms, we will discuss each algorithm and plot the corresponding flow charts accordingly.

1. Adding new student at the front:
    Prompt user for student name, ID and book borrowed
    Set the head of the list as this new node (student)
    Set the pointer of this student to the previous head node

2. Adding new student at the end:
    Prompt user for student name, ID and book borrowed
    Check if the list is empty and if it is, set this new student as the head of the list
    Otherwise, set the pointer of the last student in the list to this new student
    Set the pointer of the new student to NULL

3. Adding new student at the middle:
    Prompt the user for student name, ID and book borrowed
    Prompt the user for the positioning of this new student
    Set the pointer of the student before the inputted position to this new student
    Check if there are students after the new student
    If there are more students, set the pointer of the new student to the student that the previous student was pointing to
    If there are no more students, make the new student the last one and set its pointer to NULL

4. Deleting a student from the front:
    If the list is empty, return list is empty
    If the list is not empty, follow these steps
    Delete the student at the front of the linked list
    Set head to the very next student after the deleted one

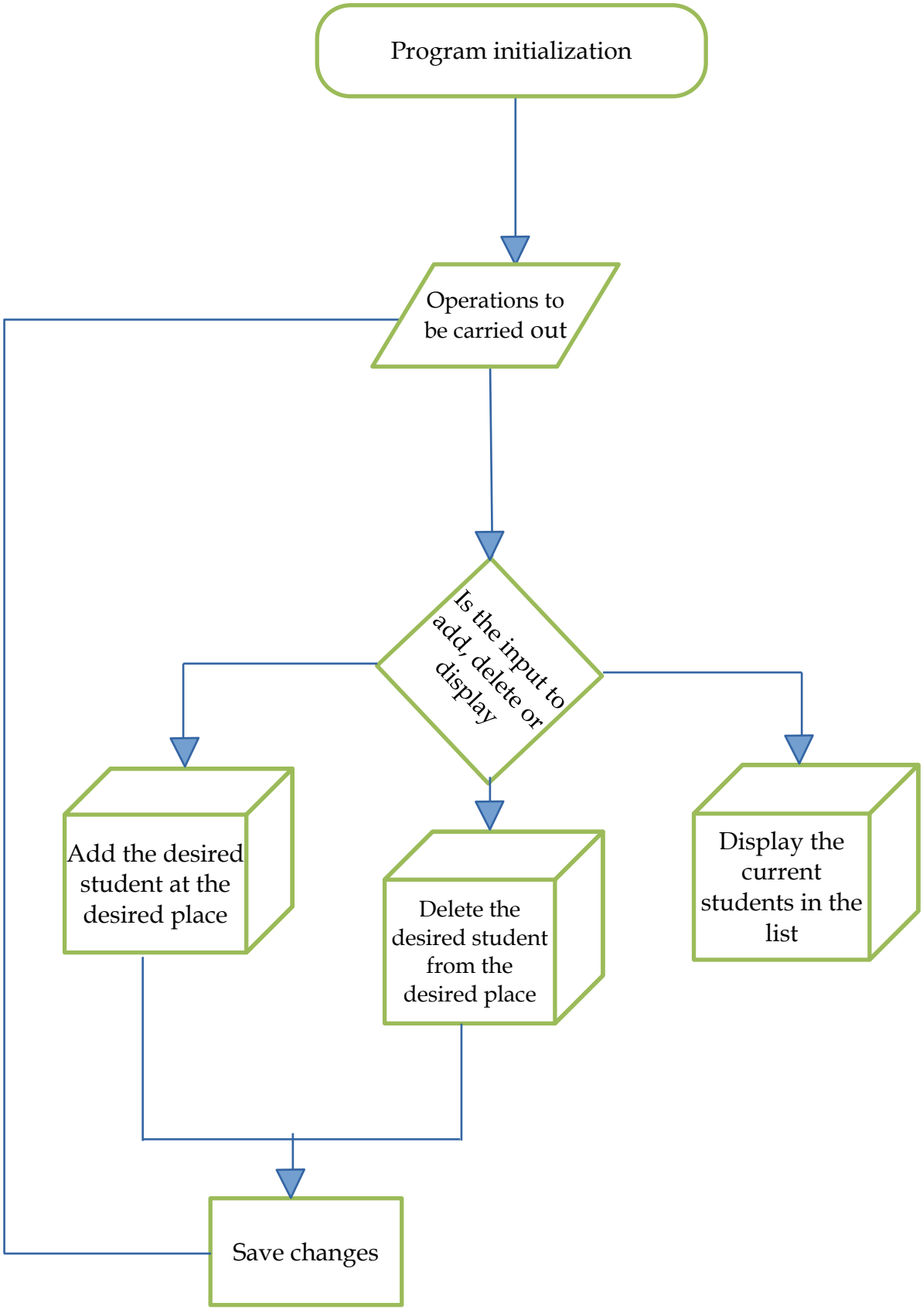
5. Deleting a student from the end:
    If the list is empty, return list is empty
    If the list is not empty, follow these steps
    Delete the student at the very end of the list
    Set the pointer of the newly made last student to NULL

6. Deleting a student at the middle:
    Prompt the user for the position of the student

If the position doesn't exist, return student doesn't exist

If the position exists, delete the student at that position and set the pointer of the previous node to the node that is next to the deleted node

The flowchart for these algorithms is relatively simple as it includes all operations at once. Trying to describe the whole code in flowcharts is going to be very demanding in space so the basic workings of the code will be explained in the following flow chart mechanism.



The algorithms listed above are the main functions we used to implement this code. So without further ado, here is the code:

```
#include <iostream>
#include <string>

using namespace std;

// Define the student record structure
struct student_record {
    string student_name;
    int student_id;
    string borrowed_book;
    student_record* next;
};

// Define a function to add a new student to the front of the list
void add_stud_front(student_record** head, string name, int id, string book) {
    // Create a new student record
    student_record* new_node = new student_record;
    new_node->student_name = name;
    new_node->student_id = id;
    new_node->borrowed_book = book;

    // Make the new node the head of the list
    new_node->next = *head;
    *head = new_node;
}

// Define a function to add a new student to the end of the list
void add_stud_end(student_record** head, string name, int id, string book) {
    // Create a new student record
    student_record* new_node = new student_record;
    new_node->student_name = name;
    new_node->student_id = id;
    new_node->borrowed_book = book;
    new_node->next = NULL;

    // If the list is empty, make the new node the head
    if (*head == NULL) {
        *head = new_node;
        return;
    }

    // Traverse the linked list to find the last node
    student_record* last = *head;
    while (last->next != NULL) {
        last = last->next;
    }

    // Add the new node to the end of the list
```

```
    last->next = new_node;
}

// Define a function to add a new student to the middle of the linked list
void add_stud_middle(student_record** head, int pos, string name, int id,
string book) {
    // Create a new student record
    student_record* new_node = new student_record;
    new_node->student_name = name;
    new_node->student_id = id;
    new_node->borrowed_book = book;

    // If the list is empty, make the new node the head
    if (*head == NULL) {
        *head = new_node;
        return;
    }

    // If the position is 0, make the new node the head
    if (pos == 0) {
        new_node->next = *head;
        *head = new_node;
        return;
    }

    // Traverse the linked list to find the node at the given position
    int count = 0;
    student_record* curr = *head;
    student_record* prev = NULL;
    while (curr != NULL && count < pos) {
        prev = curr;
        curr = curr->next;
        count++;
    }

    // Add the new node to the middle of the linked list
    prev->next = new_node;
    new_node->next = curr;
}

// Define a function to delete the first student from the linked list
void delete_node_front(student_record** head) {
    // If the linked list is empty, return nothing
    if (*head == NULL) {
        return;
    }

    // Delete the first node and make the next node the new head
    student_record* temp = *head;
    *head = (*head)->next;
    delete temp;
}

// Define a function to delete a student from the middle of the linked list by
position
void delete_stud_middle(student_record** head, int pos) {
```

```
    // If the linked list is empty, do nothing
    if (*head == NULL) {
        return;
    }

    // If the position is 0, delete the head node
    if (pos == 0) {
        student_record* temp = *head;
        *head = (*head)->next;
        delete temp;
        return;
    }

    // Traverse the linked list to find the node at the given position
    int count = 0;
    student_record* curr = *head;
    student_record* prev = NULL;
    while (curr != NULL && count < pos) {
        prev = curr;
        curr = curr->next;
        count++;
    }

    // If the position is greater than the length of the linked list, do
nothing
    if (curr == NULL) {
        return;
    }

    // Delete the node at the given position
    prev->next = curr->next;
    delete curr;
}

// Define a function to delete the last student from the linked list
void delete_node_end(student_record** head) {
    // If the linked list is empty, return nothing
    if (*head == NULL) {
        return;
    }

    // If the linked list has only one node, delete it and set the head to NULL
    if ((*head)->next == NULL) {
        delete *head;
        *head = NULL;
        return;
    }

    // Traverse the linked list to find the second-to-last node
    student_record* second_last = *head;
    while (second_last->next->next != NULL) {
        second_last = second_last->next;
    }

    // Delete the last node and set the next pointer of the second-to-last node
to NULL
```

```
    delete second_last->next;
    second_last->next = NULL;
}

// Define a function to delete a node from the linked list by ID
void delete_node_id(student_record** head, int id) {
    // If the linked list is empty, do nothing
    if (*head == NULL) {
        return;
    }

    // If the head node has the matching ID, delete it and set the next node as
the new head
    if ((*head)->student_id == id) {
        student_record* temp = *head;
        *head = (*head)->next;
        delete temp;
        return;
    }

    // Traverse the linked list to find the node with the matching ID
    student_record* curr = *head;
    student_record* prev = NULL;
    while (curr != NULL && curr->student_id != id) {
        prev = curr;
        curr = curr->next;
    }

    // If the matching node is not found, do nothing
    if (curr == NULL) {
        return;
    }

    // Delete the node with the matching ID
    prev->next = curr->next;
    delete curr;
}

// Define a function to display all nodes in the linked list
void display_list(student_record* head) {
    // Traverse the linked list and print the contents of each node
    student_record* curr = head;
    while (curr != NULL) {
        cout << "Name: " << curr->student_name << ", ID: " << curr->student_id
<< ", Book: " << curr->borrowed_book << endl;
        curr = curr->next;
    }
}
// Define a function to display a student by ID
void display_student_by_id(student_record* head, int id) {
    // Traverse the linked list to find the node with the matching ID
    student_record* curr = head;
    while (curr != NULL) {
        if (curr->student_id == id) {
            cout << "Name: " << curr->student_name << ", ID: " << curr-
>student_id << ", Book: " << curr->borrowed_book << endl;
```

```
            return;
        }
        curr = curr->next;
    }
    cout << "Student with ID " << id << " not found." << endl;
}

int main() {
    student_record* head = NULL;
    int choice;
    string name, book;
    int id, pos;

    // Display menu and get user input until the user chooses to exit
    do {
        cout << "1. Add student at front\n";
        cout << "2. Add student at end\n";
        cout << "3. Add student at middle\n";
        cout << "4. Delete student from front\n";
        cout << "5. Delete student from middle\n";
        cout << "6. Delete student from end\n";
        cout << "7. Delete student by ID\n";
        cout << "8. Display all students\n";
        cout << "9. Display student by ID\n";
        cout << "10. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;
        switch (choice) {
            case 1:
                cout << "Enter name: ";
                cin >> name;
                cout << "Enter ID: ";
                cin >> id;
                cout << "Enter borrowed book: ";
                cin >> book;
                add_stud_front(&head, name, id, book);
                break;
            case 2:
                cout << "Enter name: ";
                cin >> name;
                cout << "Enter ID: ";
                cin >> id;
                cout << "Enter borrowed book: ";
                cin >> book;
                add_stud_end(&head, name, id, book);
                break;
            case 3:
                cout << "Enter position: ";
                cin >> pos;
                cout << "Enter name: ";
                cin >> name;
                cout << "Enter ID: ";
                cin >> id;
                cout << "Enter borrowed book: ";
                cin >> book;
                add_stud_middle(&head, pos, name, id, book);
```

```
                break;
            case 4:
                delete_node_front(&head);
                break;
            case 5:
                cout << "Enter position: ";
                cin >> pos;
                delete_stud_middle(&head, pos);
                break;
            case 6:
                delete_node_end(&head);
                break;
            case 7:
                cout << "Enter ID: ";
                cin >> id;
                delete_node_id(&head, id);
                break;
            case 8:
                display_list(head);
                break;
            case 9:
                cout << "Enter ID: ";
                cin >> id;
                display_student_by_id(head, id);
                break;
            default:
                cout << "Invalid choice\n";
            case 10:
                break;
            }
    } while (choice != 10);

    return 0;
}
```

2. This section contains the book store of the program. This program allows user to add books at the front and end of the linked list, delete books by ID and display all existing books in the system. Note that we have used getline command instead of cin. This is so that the program supports multiple words for books that are added and deleted. Example: If we have a certain book named “Charlotte's web”, the usual method of using cin to receive input from the user won’t work as the string has more than one word. This is fixed by using getline instead of cin. This is why the header file “string” is added in the code. Just like the first code, we have used similar commenting style to elaborate on what’s happening in each steps of the code so that a person can understand the code without much issue. Basic algorithms used in this code are:

1. Adding new book at the front:
    Prompt user for book name, ID and author
    Set the head of the node as this new book
    Set the pointer of this node to the previous head node

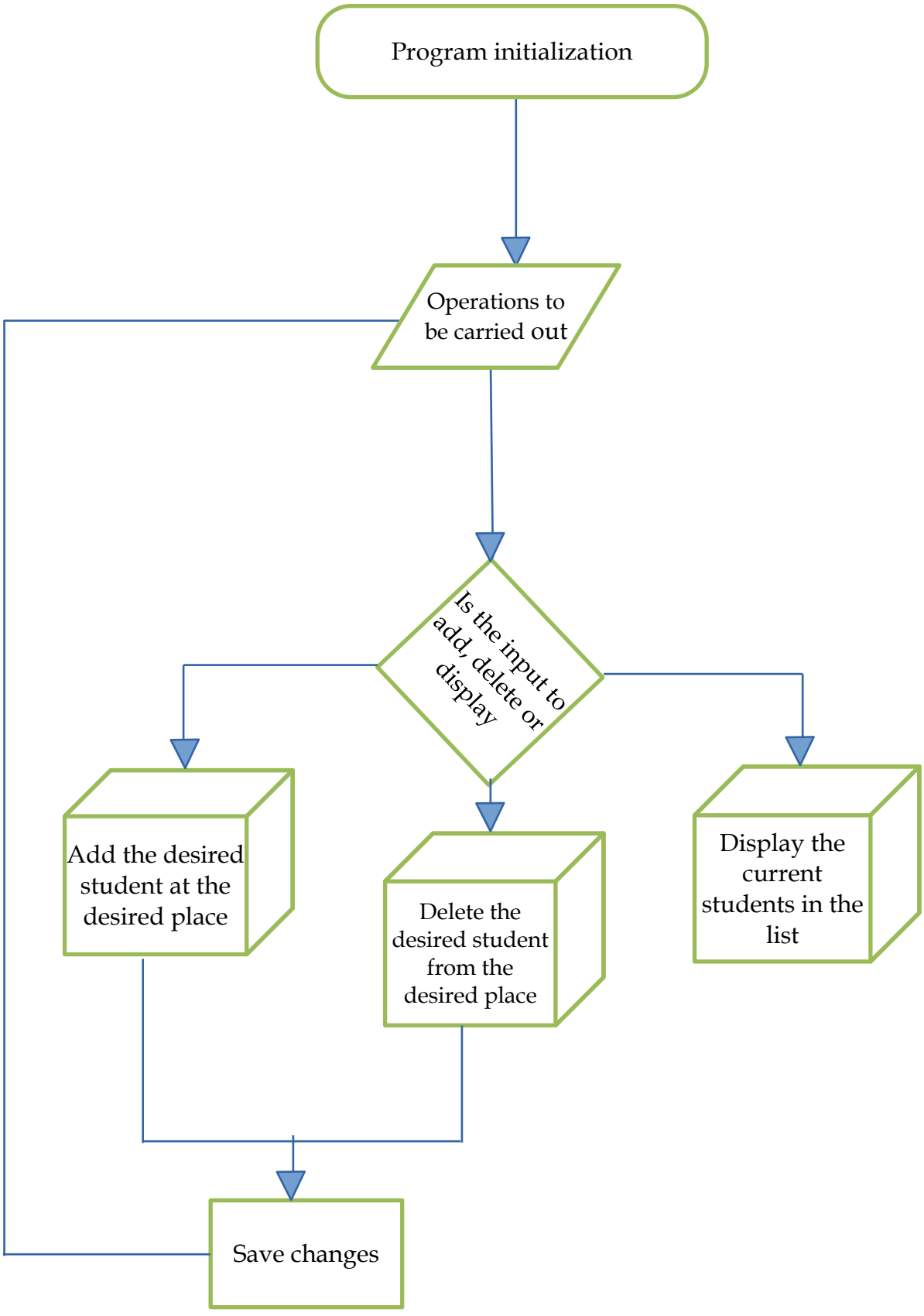
2. Adding new book at the end:

Prompt user for book name, ID and author
Check if the list is empty and if it is, set this new book as the head of the list
Otherwise, set the pointer of the last book in the list to this new book

3. Adding a new book at the middle:
Prompt the user for book name, ID and author
Prompt the user for the positioning of this new book
Set the pointer of the book before the inputted position to this new book
Check if there are more books after the new book
If there are more books, set the pointer of the new book to the book that the previous book was pointing to
If there are no more books, make the new book the last book

4. Delete book by book ID:
Prompt user to input ID
If the ID given by the user doesn't exist among the books, return book not found
If the ID given by the user exists among the books, delete the book and set the pointer of the previous book to the book next to the deleted book

The flowchart for this algorithm is very similar to the student registration mechanism:


Program initialization
Operations to be carried out
Is the input to add, delete or display
Add the desired student at the desired place
Delete the desired student from the desired place
Display the current students in the list
Save changes

As we can see, the flowchart is similar to the first one shown above. However, the code for those two are very different from one another as we will see below. Once again, the code has been commented pretty thoroughly to help anyone understand it. So without further ado, here's the code:

```
#include <iostream>
#include <string>
using namespace std;

struct BookNode {
    string book_name;
    int book_id;
    string author;
    BookNode* next;
    BookNode(string name, int id, string auth) {
        book_name = name;
        book_id = id;
        author = auth;
        next = nullptr;
    }
};

struct BookList {
    BookNode* head;
    BookList() {
        head = nullptr;
    }
    void add_book_at_front(string name, int id, string auth) {
        BookNode* new_book = new BookNode(name, id, auth);
        new_book->next = head;
        head = new_book;
        cout << "Book added successfully" << endl;
    }
    void add_book_at_end(string name, int id, string auth) {
        BookNode* new_book = new BookNode(name, id, auth);
        if (head == nullptr) {
            head = new_book;
            cout << "Book added successfully" << endl;
            return;
        }
        BookNode* last_book = head;
        while (last_book->next != nullptr) {
            last_book = last_book->next;
        }
        last_book->next = new_book;
        cout << "Book added successfully" << endl;
    }
    void add_book_at_middle(string name, int id, string auth, int position) {
        BookNode* new_book = new BookNode(name, id, auth);
        if (position == 1) {
            new_book->next = head;
            head = new_book;
            cout << "Book added successfully" << endl;
```

```
            return;
        }
        BookNode* current_book = head;
        for (int i = 1; i < position - 1; i++) {
            if (current_book == nullptr) {
                cout << "Position out of range" << endl;
                return;
            }
            current_book = current_book->next;
        }
        new_book->next = current_book->next;
        current_book->next = new_book;
        cout << "Book added successfully" << endl;
    }
    void delete_book_by_id(int id) {
        BookNode* current_book = head;
        if (current_book != nullptr && current_book->book_id == id) {
            head = current_book->next;
            delete current_book;
            cout << "Book deleted successfully" << endl;
            return;
        }
        BookNode* prev_book = nullptr;
        while (current_book != nullptr && current_book->book_id != id) {
            prev_book = current_book;
            current_book = current_book->next;
        }
        if (current_book == nullptr) {
            cout << "Book ID not found" << endl;
            return;
        }
        prev_book->next = current_book->next;
        delete current_book;
        cout << "Book deleted successfully" << endl;
    }
    void display_books() {
        BookNode* current_book = head;
        if (current_book == nullptr) {
            cout << "No books in the store" << endl;
        }
        while (current_book != nullptr) {
            cout << "Book name: " << current_book->book_name << endl;
            cout << "Book ID: " << current_book->book_id << endl;
            cout << "Author: " << current_book->author << endl;
            current_book = current_book->next;
        }
    }
};

int main() {
    BookList books;
    int choice, id, position;
    string name, author;
    while (true) {
        cout << "-----------------------------" << endl;
        cout << "          BOOK STORE          " << endl;
```

```
cout << "-----------------------------" << endl;
cout << "1. Add book at front" << endl;
cout << "2. Add book at end" << endl;
cout << "3. Add book at position" << endl;
cout << "4. Delete book by ID" << endl;
cout << "5. Display all books" << endl;
cout << "6. Exit" << endl;
cout << "Enter your choice: ";
cin >> choice;
switch (choice) {
case 1:
    cout << "Enter book name: ";
    cin.ignore();
    getline(cin, name);
    cout << "Enter book ID: ";
    cin >> id;
    cout << "Enter author name: ";
    cin.ignore();
    getline(cin, author);
    books.add_book_at_front(name, id, author);
    break;
case 2:
    cout << "Enter book name: ";
    cin.ignore();
    getline(cin, name);
    cout << "Enter book ID: ";
    cin >> id;
    cout << "Enter author name: ";
    cin.ignore();
    getline(cin, author);
    books.add_book_at_end(name, id, author);
    break;
case 3:
    cout << "Enter book name: ";
    cin.ignore();
    getline(cin, name);
    cout << "Enter book ID: ";
    cin >> id;
    cout << "Enter author name: ";
    cin.ignore();
    getline(cin, author);
    cout << "Enter position: ";
    cin >> position;
    books.add_book_at_middle(name, id, author, position);
    break;
case 4:
    cout << "Enter book ID to delete: ";
    cin >> id;
    books.delete_book_by_id(id);
    break;
case 5:
    books.display_books();
    break;
case 6:
    cout << "Exiting program" << endl;
    return 0;
```

```
        default:
            cout << "Invalid choice" << endl;
        }
    }
    return 0;
}
```

Conclusion

In this mini project, we tried to create a bookstore management system that records and saves student's names, IDs and books they borrowed to track the circulation of books in the campus. We also created the bookstore's book management that lets user to add books by their names give them IDs and remember the authors. Both of these mechanisms have options to add and delete based on the user's likings. Both mechanisms have options to display their contents at once. That said, though this is an efficient program to run as a console application, much more work is needed to make this into a useful application or website to actually use it in the campus. Due to this reason, the option to log out the system cannot be done because a console application would be closed by simply pressing the "esc" key on the keyboard. Other than that, this project has been helpful to us in order to learn more about the implementation of data structure in different types of scenarios.