CMPE126 Final Project Deliverables

Abstract:

Project Statement: Comprehensive Application of Data Structures and Algorithms

Objective:
This project aims to integrate and apply the concepts of data structures and algorithms covered in CMPE126, including queues, stacks, linked lists, binary trees, hashing, STL, sorting, and searching algorithms. The project will allow students to creatively design and implement a solution in C++ while solidifying their understanding of the material.

Introduction:

For this project, the constraints were:

- At least three different data structures discussed in the course (e.g., stack, queue, linked list, binary tree).
- One or more sorting or searching algorithms.
- Efficient use of STL containers or algorithms.
- Big-O analysis of the major operations in the program.

Given these constraints, I decided to base my project around a car auction program, integrating a variety of data structures to handle data management and storing. Using a car auction program, I was able to efficiently integrate different data structures into my code as a way to store information. Using queues, a BST, and a linked-list, I was able to organize data for sorting and searching purposes. The car auction has 4 different functions, (1) the ability to add cars to the auction, (2) the ability to start the auction, (3) to exit the program, and (4) to search a vehicle based on its ID number.

Body:

When planning on this project, I didn't know how to initially start the project. In the planning phases, I determined the four functions of the program as listed above. I started the program with the Car.h file, as it would be the file where I can define a structure called 'bidder' to hold the bidders' information and a Car class, which would be the class dealing with adding vehicles to the auction and is responsible for getting all the relevant information of the vehicle in auction.

```
//operator overload to print
friend ostream& operator<<(ostream& os, const bidder& b) {
    os << b.name << ": $" << b.bid_amount;
    return os;
}
};
```

Figure 1 - Overloaded Operator

An important part of my structure definition that I want to highlight was overloading the << operator, so I could print out my own defined data type. I also decided to use a linked-list to add a list of bidders to the car class, and defined the linked-list as a template class incase I was to use it again in my code.

```
class Car {
private:
    string brand;
    string model;
    int year;
    int vin;
    double starting_bid;
    int count = 0;
    LinkedList<bidder> bidders;
```

Figure 2 - List of variables in the Car class, including the bidder linked-list

```
void addSorted(const T& value) {
    Node<T>* newNode = new Node<T>(value);

    if (!head || value.bid_amount > head->data.bid_amount) {
        newNode->next = head;
        head = newNode;
    } else {
        Node<T>* current = head;
        while (current->next && current->next->data.bid_amount > value.bid_amount) {
            current = current->next;
        }
        newNode->next = current->next;
        current->next = newNode;
    }
}
```

Figure 3 - Sorted addition of items into linked-list (By bid amount)

After implementing the basic getter functions, I focused my attention on my auction.h file, which has the main job of dealing with the class objects and storing them into a BST and a queue so I can then use them in an auction system. I created a template for the BST. Later, I decided to use the BST as a way to look up the ID number of specific vehicles because it has a best-case time complexity of O(log n) and a worst-case of O(n), and also because the storage of the ID number is going to be added dynamically.

```
void insert(BSTNode<T>*& node, const T& value) {
    if (!node) {
        node = new BSTNode<T>(value);
    } else if (value.getID() < node->data.getID()) {
        insert(node->left, value);
    } else if (value.getID() > node->data.getID()) {
        insert(node->right, value);
    }
}
```

Figure 4 - Binary Search Tree's insert method for ID numbers

For handling the vehicles in the auction, I went for a queue and utilized the queue STL. This was because I wanted to have a First-In-First-Out type of storage for the vehicles, similar to a real auction. The queue STL functions .pop() and .front() were extremely helpful for FIFO and managed the auction itself.

```
Car& getNextCar() {
    return carQueue.front();
}

void removeNextCar() {
    carQueue.pop();
}
```

Figure 5 - Usage of the .pop() and .front() functions

The final file I had to implement was my main file; in charge of the four commands listed in the introduction: (1) the ability to add cars to the auction, (2) the ability to start the auction, (3) to exit the program, and (4) to search a vehicle based on its ID number. These commands would be chosen through a menu which would prompt on compilation of the program.

```
int main() {
    Auction auction;
    string command;

    cout << "Welcome to the Car Auction Program!" << endl;

    while (true) {
        cout << "\nChoose an operation:" << endl;
        cout << "1. Add a car to the auction (enter 'add')" << endl;
        cout << "2. Start the auction (enter 'auction')" << endl;
        cout << "3. Exit the program (enter 'exit')" << endl;
        cout << "4. Search vehicle by ID# (enter 'ID')" << endl;
        cout << "Command: ";
        cin >> command;
```

Figure 6.1 - Menu

```
Welcome to the Car Auction Program!

Choose an operation:
1. Add a car to the auction (enter 'add')
2. Start the auction (enter 'auction')
3. Exit the program (enter 'exit')
4. Search vehicle by ID# (enter 'ID')
Command:
```

Figure 6.2 - Output upon compilation

(1) Adding cars to the auction

To add cars to the auction, the program will take user input of all relevant information, then store them in both the BST and queue by referencing the .addCar() function in the auction.h file

```
if (command == "add") {
    string brand, model;
    int year, ID;
    double starting_bid;

    cout << endl;
    cout << "Enter car brand: ";
    cin >> brand;
    cout << "Enter car model: ";
    cin >> model;
    cout << "Enter car year: ";
    cin >> year;
    cout << "Enter car ID: ";
    cin >> ID;
    cout << "Enter starting bid: ";
    cin >> starting_bid;
    cout << endl;

    auction.addCar(brand, model, year, ID, starting_bid);
}
```

Figure 7.1 - User input for vehicles

```
void addCar(const string& brand, const string& model, int year, int ID, double starting_bid) {

    Tbrand = brand;
    Tmodel = model;
    Tyear = year;

    Car newCar(brand, model, year, ID, starting_bid);
    carQueue.push(newCar);
    carBST.insert(newCar);
    cout << "Car added: " << year << " " << brand << " " << model << " (VIN: " << ID << ")" << endl;
}
```

7.2 - .addCar() function

After adding a vehicle, the vehicle will be displayed before prompting for any more additions of vehicles to ensure the user can see what they are inputting at each step. The menu prompt will show up every loop so the user can choose from the options displayed after adding vehicles.

```
Enter car brand:
Toyota
Enter car model:
Corolla
Enter car year:
2023
Enter car ID:
1098772
Enter starting bid:
20000

Car added: 2023 Toyota Corolla (ID: 1098772)

Choose an operation:
1. Add a car to the auction (enter 'add')
2. Start the auction (enter 'auction')
3. Exit the program (enter 'exit')
4. Search vehicle by ID# (enter 'ID')
Command:
```

Figure 7.3 - Example of add function

(2) Starting the car auction

The hardest function and the main function of the program was the actual auctioning of the vehicles. Upon calling the “start the auction” option from the menu, the code will output the vehicles on the car auction list and start the process of asking for the bidders information. For sake of simplicity, the conditions for selling a vehicle is 5 bids, with the 5th bid being the one that it is sold to. Of course, there is also a ‘done’ command that can be used to override the 5 count rule.

```
else if (command == "auction") {
    cout << "\nStarting the auction!" << endl;

    while (true) {
        auction.printQueue();
        if (auction.isQueueEmpty()) {
            cout << "All vehicles have been sold. Auction is over." << endl;
            return 0; // End program
        }

        Car& currentCar = auction.getNextCar();
        cout << "Auctioning vehicle: " << currentCar.getYear() << " " << currentCar.getBrand() << " "
            << currentCar.getModel() << " (ID: " << currentCar.getID() << ")" << endl;

        while (currentCar.getCount() < 5) {
            cout << "Enter bidder's name (or 'done' to stop bidding on this car): ";
            string bidderName;
            cin >> bidderName;
            if (bidderName == "done") break;

            cout << "Enter bid amount: ";
            double bidAmount;
            cin >> bidAmount;
            cout << endl;

            if (currentCar.addBid(bidderName, bidAmount)) {
                cout << "Bid accepted!" << endl;
                cout << endl;
            } else {
                cout << "Bid too low, Invalid!" << endl;
            }
        }

        currentCar.printBidders();
        cout << "This vehicle is sold!" << endl;

        auction.removeNextCar();  // Remove the car from auction after sale
    }
}
```

Figure 8.1 - Code for the ‘Auction’ function

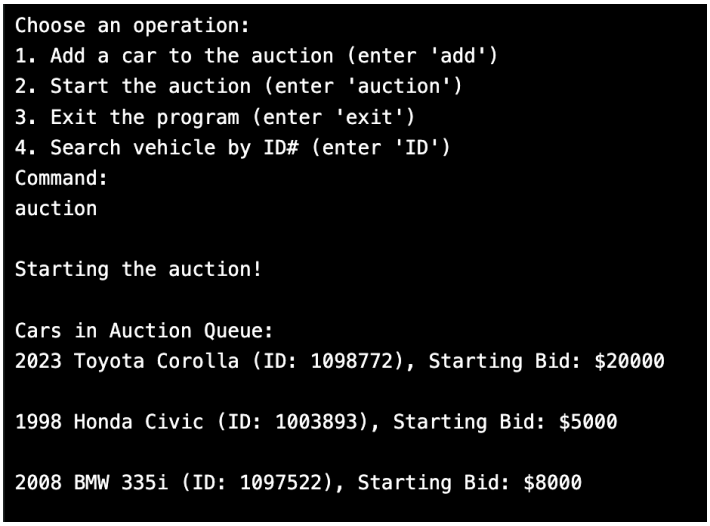

Figure 8.2 - Initial output prompt of the 'Auction' option

To add a bid to the auction queue, a call will be made to the .addBid() function in the Car.h file, which is done by creating a car object to store bidder information per vehicle in the queue.

```
Car& currentCar = auction.getNextCar();
cout << "Auctioning vehicle: " << currentCar.getYear() << " " << currentCar.getBrand() << " "
        << currentCar.getModel() << " (ID: " << currentCar.getID() << ")" << endl;
```

Figure 9.1 - Car object created

```
bool addBid(const string& name, double bid_amount) {
    // Stop bidding when a match is found or if the count exceeds 5
    if (count >= 5) {
        return false;
    }

    // Accept the bid if it's greater than or equal to the starting bid
    if ((bidders.getFront() == nullptr || bid_amount > bidders.getFront()->data.bid_amount) &&
        bid_amount >= starting_bid) {
        bidders.addSorted(bidder(name, bid_amount));
        count++;
        return true;
    }

    return false;
}
```

Figure 9.2 - .addBid() function

In the auctioning phase, there are three possible outcomes - (1) stopping the bid count early with the keyword ‘done’, (2) having all 5 bids, (3) having an invalid bid (i.e. too low).

Case (1):

For the first case, as seen in Figure 8.1, if the keyword ‘done’ is input, the loop will just break and move on to the next car.

```
Auctioning vehicle: 2023 Toyota Corolla (ID: 1098772)
Enter bidder's name (or 'done' to stop bidding on this car):
Timothy
Enter bid amount:
20300

Bid accepted!

Enter bidder's name (or 'done' to stop bidding on this car):
done
Bidders for vehicle: 2023 Toyota Corolla --- 1098772
Timothy: $20300
This vehicle is sold!
```

Figure 10.1 - Case 1 Result

Case (2):

For the second case, if 5 bids have been placed, the last bid (highest bid) will show up first on the results, and the prompt ‘This vehicle is sold!’ will be displayed at the end.

Figure 10.2 - Case 2 Result

Case (3):

For the third case, if a bid is placed and it is either below the initial bid or previous bid, it will not be counted towards the 5 count of bids, and a prompt will show up to indicate that the bid placed is invalid and not an option.

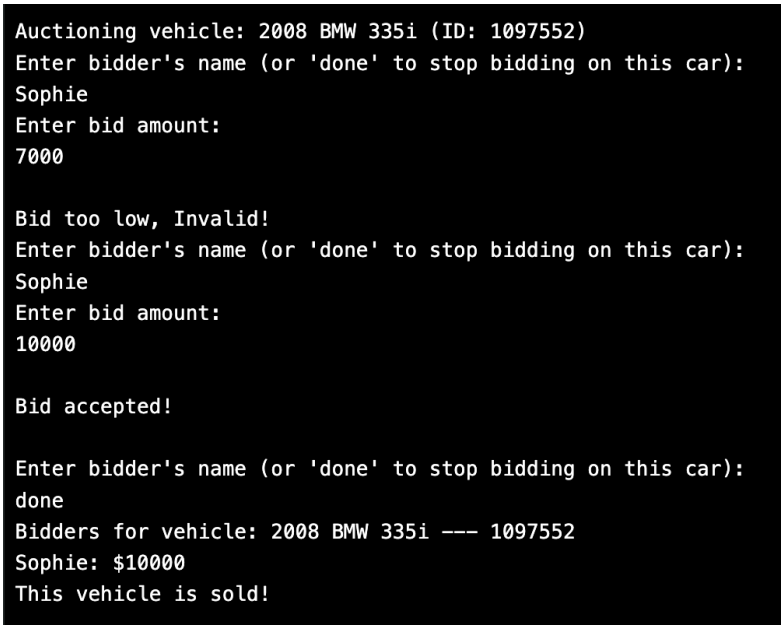

Figure 10.3 - Case 3 Result

When the queue is finally finished, a simple message will display and the program will exit.

```
Cars in Auction Queue:
No cars in the queue.
All vehicles have been sold. Auction is over.
```

Figure 10.4 - Exit Message

(3) Exiting the Program

This function is very simple; upon typing ‘exit’ a simple message will prompt ‘Exiting the program. Goodbye!’ and then break out of the loop and end the program.

(4) Searching by ID

When the ID search option is chosen from the menu, the program will call the .searchByID() function from the auction.h file. This function then calls .search() function within the binary search tree in order to match the ID that is searched.

```
    void searchByID(int ID) {
        if (carBST.search(ID)) {
            cout << "Car with ID " << ID << " found in the auction system: " << Tyear << " " <<
Tbrand << " " << Tmodel << endl;
        } else {
            cout << "Car with ID " << ID << " not found." << endl;
        }
    }
```

Figure 11.1 - .searchByID() function in auction.h file

```
bool search(BSTNode<T>* node, int ID) const {
    if (!node) return false;
    if (node->data.getID() == ID) return true;
    if (ID < node->data.getID()) return search(node->left, ID);
    return search(node->right, ID);
}
```

Figure 11.2 - .search() function in BST.h file

```
Choose an operation:
1. Add a car to the auction (enter 'add')
2. Start the auction (enter 'auction')
3. Exit the program (enter 'exit')
4. Search vehicle by ID# (enter 'ID')
Command:
ID
Enter the ID of the vehicle you would like to look up:
1097522
Car with ID 1097522 not found.

Choose an operation:
1. Add a car to the auction (enter 'add')
2. Start the auction (enter 'auction')
3. Exit the program (enter 'exit')
4. Search vehicle by ID# (enter 'ID')
Command:
ID
Enter the ID of the vehicle you would like to look up:
1097552
Car with ID 1097552 found in the auction system: 2008 BMW 335i
```

Figure 11.3 - Example of ID option

For my program the Big-O analysis / Time complexity varies depending on the options chosen from the menu. For the addition of cars in the queue, the time complexity is O(1), while for the BST, it would be O(log n). For the car auctions, because of the conditional loop of up to 5 bids placed, this would be a constant operation of O(1). However, the while loop inside of the conditional loop that handles the checking of bidders abides by a O(n) time complexity, where n is the amount of bidders. Removal of cars from the queue is an O(1) operation. Similarly, searching using a BST for an ID number is also an O(log n) operation.

Conclusion:

This project has proven to be very difficult and challenges the creativity to produce a program in order to display different uses of data structures. For me, one of the main struggles I faced was implementing the auction system itself and figuring out where to start the project. I had an issue with solving the addition and storage of bidders, as I would run into an issue where the comparisons made were incorrect or the loop would infinitely go on. Also, I wasn't sure how I

wanted to create the BST and linked-list class, which is why I decided to create them as templates, so it would be easier to apply them later on in my program. I also ran into the issue where I would struggle to keep “in scope”, like calling methods from other classes and so on. Overall, I found the project to be very difficult but fun, and I also learned different ways to integrate the data structures we learned this semester into our programs.