**Appointment Scheduling System**

**By Adrian Mora, Xuan Ho, Mahmood Abuzaina**

**Abstract**

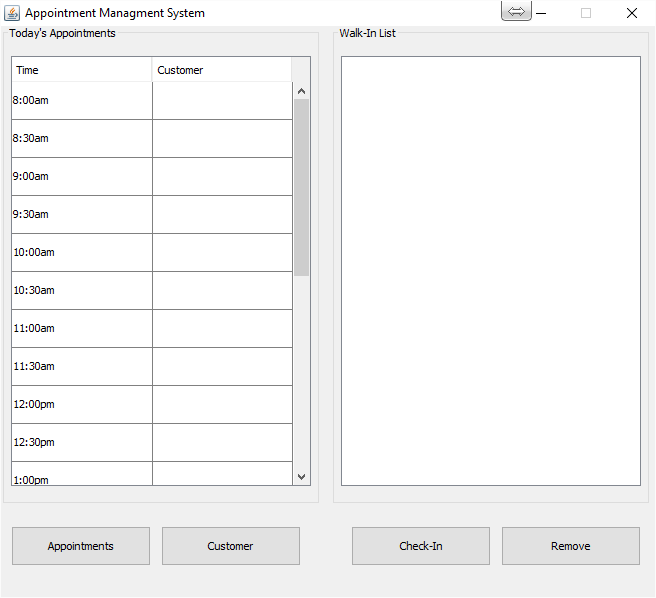
Our project is an appointment management system that was initially made specifically for barber shops, but along the way it became a management system that any businesses that has customers with appointments can use in their everyday activities. We have used the data structure and algorithm concepts of Linked List, Queue, Enqueue, Dequeue, Stacks, Binary Search Tree, Quick Sort, and Hash function that we learned in class to create that system. Queue is used for walk-in customers to determine who will be next to be serviced, as in first come first serve, and it will also determine who will get the next available appointment time. We implemented the binary search tree to search through customer’s name and find their appointment times as well as any other relevant information about them. Quick Sort algorithm is implemented for customer’s names. A concept we have covered that is not part of the class is the UI, where we used Java swing as our graphical interface. As of now, the plan is to leave it as a software, but it could be integrated into a mobile app where the customers can pick an available appointment, and the barber would get updates on his phone and the system immediately.

**Introduction**

In our appointment management system, we have a excellent GUI built using Java swing that includes four buttons and two display windows in the main graphical interface. The four buttons are called Check-In, Appointment, Customer, and Remove with each having their own functionality when clicked. The two display windows are the Appointment window and the Check-In window. The Check-In window show a queue linked list of people came to the business as a walk-in customer. The appointment window displaying a list of appointment time slots of that current day with the names of the people who had appointment, for that day, next to the time slot.

**Body**

The first thing the program does is load a text file that we named “Customers.” In this file, all relative information about the customer is stored; we store the name, phone number, appointment time, and the time when they booked the appointment. After this file gets loaded, the main method creates the GUI of the main menu of the program. After the GUI is created, the information that were stored onto the text file will be displayed.



The Check-In button calls the add method in our queue of walk-in customers. Since our queue uses a linked list, adding an item to the queue is an operation that is O(1) since there is no traversal required (item gets inserted at the beginning of the list). On the other hand, the remove button has to remove the item at the very end, so since we are using a linked list queue, it would require O(n) time because it needs to traverse all the nodes in the linkedlist. Fortunately, the number of walk-in customers that will be added to the list will be relatively low, which means that performance will not be an issue and a queue with a linked list as the data structure seemed to be the best fit.

class btnCheckIn\_Action implements ActionListener{

public void actionPerformed (ActionEvent e){

// create a panel containing labels and buttons

JPanel checkInJPan = new JPanel(new GridLayout(2, 2));

JLabel checkinCustName = new JLabel("Enter Walk-In Customer Name: ");

JTextField jtfCheckInName = new JTextField();

// add the components to the panel

checkInJPan.add(checkinCustName);

checkInJPan.add(jtfCheckInName);

checkInJPan.setVisible(true);

// add the panel into the JOptionPane

JOptionPane.showMessageDialog(null, checkInJPan, "please enter the details", JOptionPane.INFORMATION\_MESSAGE);

// create the walkin customer object

Customer WalkIn = new Customer(jtfCheckInName.getText(), new SimpleDateFormat("HH:mm:ss", Locale.ENGLISH).format(new Date()));

// add object to the queue

queWalkIn.add(WalkIn);

tree.add(WalkIn);

// create the header of the table

walkinList.setText(String.format("%-3s%-20s%-15s%n", "#","NAME", "CHECK-IN TIME"));

walkinList.append("------------------------------------");

walkinList.append("\n");

// get all the items of the list

int i = 1;

for(Customer c : queWalkIn){

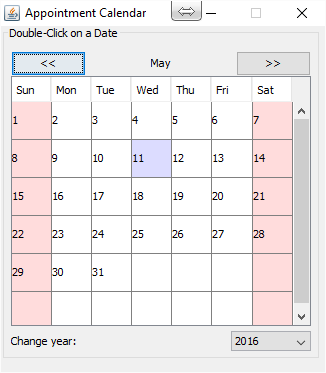
walkinList.append(String.format("%-3d%-20s%-15s%n",i,c.getName(), c.getTime()));

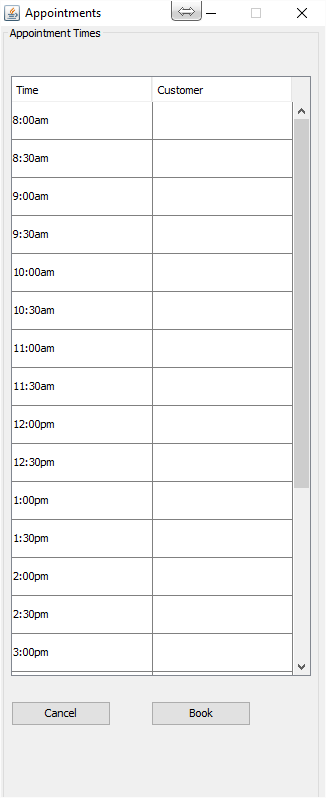
i++;

}

}

The other part of our application is for storing and retrieving a Customer object in date and time slots. Our application was designed to allow booking appointments at most 1 year in advance and to view past appointments up to 1 year ago from the current date. Each day during that 2 period is represented using an array of AppointmentDay objects. Inside each AppointmentDay object, we store an array of AppointmentTimeSlot objects, which represent different appointment slots in a day i.e. 8:00am - 8:30am. The array structure would not change in size, which meant that we did not have to worry about inserting new array elements, but rather place a Customer object inside of one of those time slots for a given date. We decided to use the hash function below, which would take a date, and it would return an index between 0 and 730. The 0th element would represent exactly 1 year ago and the index 729 would represent the exactly a year from the current date. We designed our hash function in a way that would avoid any collisions, which optimises performance. Originally, we were thinking of creating an array big enough to hold all the dates since the beginning of time, which we would then access by an index representing the date. For example, 5/11/2016 would be accessed by index 20160511. That meant that we would need to create an array with over 20 million elements! It would be really impractical in our application to allow booking appointments too far into the future and likewise to view appointments too far into the past, so we settled for the function below. The function below is a perfect hash function since there are no collisions and there is no wasted memory. The complexity of the function below is O(1) with some minimal overhead to derive the index value. Since we will be retrieving or storing data inside of the array elements, having complexity of (O1) makes our application have great performance.





public static long hash( Calendar key ){

int tableSize = 730; // table size required to store 2 years worth of dates

Calendar aYearAgo = Calendar.getInstance();

aYearAgo.add(Calendar.DAY\_OF\_MONTH, -365);

Date startDate = aYearAgo.getTime();

Date endDate = key.getTime();

long startTime = startDate.getTime();

long endTime = endDate.getTime();

long diffTime = endTime - startTime;

// diffDays stores the number of days between a year from today's date and the given date

long diffDays = diffTime / (1000 \* 60 \* 60 \* 24);

// if date is less than a year ago, return index 0

if (diffDays < 0){

return 0;

}

// if date is greater than a year ago, return last index

if (diffDays > tableSize){

return tableSize-1;

}

return diffDays;

}

We have implemented the binary search tree, where we add the customer objects into the tree. We then have a search method inside the bst package that would allow us to retrieve the correct customer object by comparing the names of the object. Here’s the method used for the search method:

public boolean Search(T entry)

{

//T result = null;

boolean found = false;

BinaryNodeInterface<T> currentNode = getRootNode();

while (!found && (currentNode != null) )

{

T currentEntry = currentNode.getData();

//System.out.println(currentNode.getData());

if (entry.compareTo(currentEntry) == 0)

{

result = currentEntry;

found = true;

}

else if (entry.compareTo(currentEntry) < 0)

currentNode = currentNode.getLeftChild();

else

currentNode = currentNode.getRightChild();

} // end while

**Conclusion**

One of the most challenging part of this project was to finalize a conceptual design. As future developers, we have learned that it is worth spending extra time creating a design and high level pseudocode of our algorithms before diving into the code. Even though most of our time was spent during the design phase, it felt like once we had finalized the design, the coding became really simple. Having a concept design upfront is what helped us choose the best data structure and algorithm possible to optimize performance and user experience. One of the good things of working as a group is that we all were able to exchange ideas and collaboratively agree on the best design possible. One of the challenging parts of working as a group was to work on code independently and then put it back together. Working as a group on this project definitely provided some experience of what it might be like to work with a group of developers at a company. Overall, we are really glad that we had the opportunity to use the tools that we have learned in class to create an application optimizing both the usage of space and runtime.