Scheduling Assistance Program for a Pediatric Office

CSC 212: Data Structures and Abstractions

Professor Jonathan Schrader

Group 5

Sarah Dzwil, Darren Medeiros, Kat Toolan, Peter Card

Abstract

Our group created a program meant for a pediatric office which schedules new appointments throughout the day, taking into account urgency of care, and avoids scheduling appointments for the same doctor at the same time. We use data structures including interval trees and priority queues, and algorithms such as radix sort and bucket sort to schedule non-overlapping appointments based on patient priorities and print a sorted list of all patients at the end of the day. In this paper, we will discuss our program including how we use our data structures and sorting algorithms, strengths and weaknesses of our program, and use cases.

Table of Contents

Introduction

We imagined a scenario at a pediatric office where appointments need to be scheduled throughout the day to accommodate walk-in patients with different urgency levels. These walk-in appointments are in addition to other patients who already have appointments scheduled by the start of the day. We built a program that schedules new appointments throughout the day, making sure no appointments with the same doctor are scheduled at the same time. Additionally, we print out statistics of new appointments added for tracking purposes. We output each doctor's appointments for the day, sorted by start time, into text files to be used as their schedules. We also output a sorted list of all the patients that came into the office today. To accomplish the goals of our program, we used data structures including interval trees and priority queues, and algorithms such as radix sort and bucket sort. We recognize our program's current usage, strengths, and weaknesses and its potential for future features, improvements, and use cases.

Discussion of Data Structures and Algorithms

Aside from basic data structures, our program makes use of two data structures and two sorting algorithms. Namely, interval trees efficiently store patient appointments with specified time intervals. Priority queues manage the order of patient processing based on priority criteria, ensuring timely handling of urgent cases. Radix sort and Bucket sort are utilized for efficient sorting of patient data by last and first names, enhancing data presentation and analysis. These data structures and sorting algorithms are discussed in detail below.

Interval Tree

The inventor of interval trees is unknown, but many believe they were invented at the same time as segment trees. Segment trees were invented in 1977 by Jon Bentley as a solution to Klee's rectangle problems (Wikipedia, 2023). An interval tree is a self-balancing tree data structure that holds intervals instead of nodes like many other tree data structures. Some applications for interval trees include finding roads on a computerized map inside a rectangular viewpoint and finding all visible elements inside a three-dimensional scene (GeeksforGeeks, 2023). Since segmentation and interval trees are often confused, let's explain the differences. Segment trees and interval trees both store intervals, but segment trees are mainly used for queries at a given point while interval trees are used for optimizing overlapping queries in a given interval (Engibaryan, 2024; Anand, n.d.).

An interval tree has the same structure as a self-balancing binary tree and has similar operations, such as insertion, deletion, and searching. To insert a new node into an interval tree, you would start at the root node, descending left if lesser and right if greater until it hits a leaf position. The time complexity for this operation is O(log(n)) (GeeksforGeeks, 2023; npm, n.d.).

The search function starts at the root node and goes down the tree, following the same path as insertion until it hits a leaf node. If the node's interval matches, it is returned, otherwise, it returns a null pointer. The time complexity is O(min(n, k * log n)) with k being the number of intervals in the output list (GeeksforGeeks, 2023; npm, n.d.; cmu, n.d.).

For deletion, we first search the tree to find the requisite node. If the node is a leaf node, then it's removed without any rebalancing. If the node has one child, then the node and child switch places then the node is deleted. If the node has two children then it's a bit more complex. The program would first have to find the minimum interval in the node's right subtree, replace the interval in the node with the minimum interval, and then remove the minimum interval from the right subtree. The time complexity of deletion is O(log(n)) (GeeksforGeeks, 2023; npm, n.d.).

The benefit of an interval tree is that it can store intervals without overlap. Each interval is a pair of integers [a,b], where a < b. These intervals are called elementary intervals, indicating the start and end points on the integer line. Each node has an elementary interval and can avoid overlap with its child nodes (GeeksforGeeks, 2023; Anand, n.d.).

Priority Queue

Queues have been used for a very long time, and the first written record of queues comes from Agner Krarup Erlang. Agner developed the system to deal with the incoming calls at the Copenhagen Telephone (Wikipedia, 2024b).

A priority queue is just an applied queue where elements have priority—in other words, how urgent each element is. The higher the priority an element has the sooner it should be dealt with. As such, priority queues share many operations with regular queues, including enqueue,

dequeue, peek, rear, full, empty, and size. These operations are required for every queue to work (GeeksforGeeks, 2023a; GeeksforGeeks, 2023b).

The enqueue algorithm adds an element to the queue. The algorithm first checks if the queue is full. If so, the programmer may choose to return an error or resize the queue. In a queue, the element goes to the next available space, but in a priority queue, the element is inserted to the right of the first element with higher priority. The time complexity of this operation is O(1) (GeeksforGeeks, 2023a; GeeksforGeeks, 2023b; Plessis, 2018).

The next operation is dequeue. The dequeue operation removes and returns the first element in the queue. The algorithm first checks if the queue is empty. If so, the programmer may choose to return an error or return a null pointer. If the queue isn't empty, then the program returns the head of the queue. This implementation is the same for a priority queue and a queue. The time complexity of this operation is O(1).

The peek operation returns the element at the front of the queue. This operation doesn't remove the element and is implemented the same for a queue and a priority queue. The time complexity of this operation is O(1) (GeeksforGeeks, 2023a; GeeksforGeeks, 2023b; Plessis, 2018).

The rear operation returns the element at the end of the queue without removing the element. Two outputs can occur: an error message if the queue is empty or the rear value if a value does exist. The time complexity of this operation is O(1) (Plessis, 2018).

The isEmpty operation is a boolean operation that checks if the queue is empty. If the queue is empty, then the operation returns true and false if there are elements inside the queue. The time complexity is O(1) (Plessis, 2018).

The last operation that can be performed is the size operation which outputs the number of elements in the queue. If the queue is empty then the operation returns an error or a zero based on the programmer's choice. The time complexity is O(1) (Plessis, 2018).

Radix Sort

Radix sort was first published in a paper by Herman Hollerith in 1887 for use with his tabulating machines. When radix sort was released, many people thought it was impractical because of the need to allocate multiple buckets of an unknown size. Radix sort got its breakthrough in 1954 when Harold H. Seward from MIT developed the first memory-efficient computer algorithm. Seward's algorithm required a linear scan, which generated the number of buckets (Wikipedia, 2024c; GeeksforGeeks, 2024).

By definition, radix sort is a linear sorting algorithm that sorts elements by processing the elements digit by digit. Radix sort is a non-comparative sorting algorithm. This means that it only reads the list elements through a single abstract comparison operation and it uses the concept of place values. There are two variations of radix sort, least significant digit radix sort and most significant digit radix sort (Yadav, 2022; GeeksforGeeks, 2024).

Least significant digit (LSD) radix sort starts from the least significant/rightmost digit and terminates at the most significant digit of each element in the array. The run time is O(nw), where n is the number of elements, and w is the average element length. The sort iterates through each element, inserting them into buckets. If an element does not have a digit at the position being sorted, the algorithm treats it as a 0. This continues until the longest element(s) have been sorted, and results in a list sorted from large to small numbers (Yadav, 2022).

Most significant digit (MSD) radix sort sorts the elements in lexicographic order which doesn't preserve the order of the original array. The algorithm takes the most significant digit of each element in the array as a key, which is then sorted into buckets. The algorithm then moves to the next significant digit and sorts that until every digit has gone through and has been sorted. A recursive function is the most straightforward way to implement this sorting method (Yadav, 2022).

Bucket Sort

Bucket sort has an unclear history since some people claim to have invented it in 1955 but people argue that the methods which are used in bucket sort have been around since 1945. No matter when bucket sort was developed, it wasn't commonly used until the 1970s and then exploded in popularity in 1995 when computing speed increased. This is when computer scientists and developers realized the use of bucket sort (Allan, 2024).

Bucket sort was derived from a counting sort method which had been around for many years beforehand, and is quite simple in its implementation. The sort divides elements into buckets which can be easily sorted by any other sorting algorithm. Bucket sort works best in tandem with another sorting algorithm. To sort an array using bucket sort you would first have to know the size of the array, in this context, we will call it "i". After a user knows the "i" then each element in an array would be multiplied by "i" to learn which bucket the element would go into. All items are then converted into integers to be put into buckets. The elements need to be converted into integers since buckets can only be in whole numbers, converting the items into integers forces them to be whole numbers. In many cases, some items in the original array may be in the same bucket which is ok. In cases in which that occurs a linked list would be

implemented to store multiple items in the same bucket. Once items are in the bucket a sorting algorithm will be used (GeeksforGeeks, 2024a).

Bucket sort works great for almost every sorting algorithm. After the sorting occurs the buckets are then put back into the original array. The time complexity of bucket sort is O(n^2), where "n" is the number of elements in the original array. The space complexity of bucket sort is O(n+k). K is the number of buckets and "n" is the number of elements in the original array (Hebert & Tsai, 2013).

Discussion of our Program

Overview of Microcosm

Our group developed a program to be used at a pediatric office to handle a day's worth of patients, including patients with appointments and walk-ins. Our program assumes the office opens at 8am and closes at 8pm, taking walk-ins from 8am to 7pm who get appointments during the following hour if there is room. Patients with scheduled appointments are read in at the beginning of the day as their appointments are added into our data structures. At the start of each day, we output a schedule for each doctor containing the appointments they have today.

Walk-in patients without appointments enter each hour. Their personal information and reasons for visiting are recorded and added into our records. The urgency of care needed is taken into account when determining who to give walk-in appointments to. Patients enter with a triage level of between 1 and 5, with 1 being the highest priority. To schedule a walk-in appointment, we try to match a patient with a doctor of their preferred gender, checking that the doctor does not already have an appointment that would conflict with a new appointment. If there are no scheduling conflicts, the appointment is added. If there are scheduling conflicts with the first doctor checked, other doctors' availability are checked. If no doctors have any room for more walk-in appointments, all remaining walk-in patients will be turned away since there are no more appointment slots open for the hour. At the end of each hour, an updated list of the day's appointments for each doctor is output. This is meant to be the most up-to-date schedule for each doctor. This walk-in scheduling process repeats each hour until the end of the day. Every hour, an update of new appointments made and walk-ins turned away is given to the terminal in order to clearly track where the terminal is in the program. Note that appointments will never be removed

from the interval trees to insert another appointment as this would imply an appointment is moved by the office, which would make patients upset if done to their appointments.

At the end of the day, a final list of all the patients that came into the office today are output, sorted by name, with information on whether or not they were seen at the office or turned away. Using the idea behind radix sort, sorting by the least significant digit first, we sort by first name and then again by last name. This results in our printed and sorted list of patients being ordered by last name, then first name. The percentage of walk-ins sent elsewhere are output to the terminal for a quick summary of success of walk-ins.

Table 1: Input/Output Summary for Different Stages of the Day

Start of the day (7am)	
Input	Run with ./main <date> with date read in through command line argument as YYYY-MM-DD (ex: “2024-04-15”) Files: <date>_scheduled.txt (Last Name, First Name, Gender, Doctor, Start Time)
Structures/ Algorithms	Interval trees- 1 per doctor, which shows that doctor’s appointments
Output	Terminal: “Start of day. <#> appointments scheduled. Daily schedules made.” Files: One file for each doctor containing a list of appointments they have today (including patient name and appointment start and end times)

Each hour from 8am to 7pm	
Input	Terminal: Get the user to press enter to begin the hour. Files: Walk in patients for 8am, 9am, …, 7pm (1 file per hour)
Structures/ Algorithms	Priority queue- walk-in patients added to priority queue to get appointments. Priority is based on urgency of care. Interval trees- check doctor interval trees (checking gender preference first) for if a new appointment can be made at a certain time. If the new appointment doesn't overlap with existing appointments, add in. Otherwise, keep checking until no more options are left and send patient elsewhere.
Output	Terminal: "New hour: <hour>. <#> patients walked in. <#> walk-ins got an appointment. <#> walk-ins turned away. Daily schedules updated." Files: Update files for each doctor containing a list of appointments they have today (including patient name and appointment start and end times)
End of day (8pm)	
Structures/ Algorithms	Radix sort (w/ bucket sort) to sort patient names
Output	Files: Output to .txt file a list of all patients that were in the hospital, sorted by last name and first name (to accomplish this in the program sort by first name first then sort by last name; this uses the idea behind

	radix sort). Each row is “<last name>, <first name>, <appointment or walk-in>”. Terminal: “End of day. <#> patient records written to file. <%> of walk-ins sent elsewhere.”

Implementation

The first data structure used in the program is interval trees. This program creates interval trees for each doctor who is scheduled for the day. Currently, there are only three doctors: Jenny, Taylor, and Paul. This creates three corresponding interval trees which are stored in a map for easy access later in the code. At this stage, the interval trees are empty, and will be filled later in the code.

```
int readScheduleFilename (std::string date, std::vector<Patient>& vec, std::map<std::string, IntervalTree>& doctorTrees) {

    std::string filename = generateScheduledFilename(date);

    std::ifstream input;
    input.open(filename);
    if (input.fail()) {
        std::cerr << "Can't open file." << std::endl;
        return 1;
    }

    std::string line, lname, fname, gender, doctor, strStart;
    int start, end;
    int patientCount = 0; // keep count of patients added

    std::getline(input, line); // get header out
    while (std::getline(input, line)) {
        std::stringstream streamline(line);
        // parse elements of line
        // ex: Doe,John,Male,Jenny,0900
        getline(streamline, lname, ',');
        getline(streamline, fname, ',');
        getline(streamline, gender, ',');
        getline(streamline, doctor, ',');
        getline(streamline, strStart, ',');
        start = std::stoi(strStart);
        end = start + 29;

        // create patient node
        Patient currentPatient(fname, lname, gender, "Appointment");

        // add patient to linked list and vector
        vec.push_back(currentPatient);

        // add patient to interval tree for their doctor
        if (doctorTrees[doctor].insertInterval(lname, fname, start, end)) {
            //std::cout << "Appointment inserted." << std::endl;
```

The interval trees will be partially filled in the "readScheduleFilename" function. The "readScheduleFilename" takes in three inputs: the date, the vector filled with patients, and doctorTrees, which is the map that holds the doctor's interval tree.

```
std::string generateScheduledFilename(std::string date) {
    return "../Input/" + date + "_scheduled.txt";
}
```

The function will call the "generateScheduledFilename" which takes in the date and creates a file that matches the date. The file that is outputted from "generateScheduledFilename" contains all the patient's data. The code following the function after will read in the patient's last name, first name, gender, doctor assigned, and the appointment start time. Once the file is read and the information is obtained; a node is created for each patient. The node for each patient is inserted into a doctor's interval tree. The output of the function is the number of patients who are scheduled for the day.

```
struct Patient {
    std::string firstName;
    std::string lastName;
    std::string gender;
    std::string type;
    Patient* next;

    // Constructors
    Patient(const std::string& firstName, const std::string& lastName,
            const std::string& gender, const std::string& type, Patient* next);
    Patient(const std::string& firstName, const std::string& lastName,
            const std::string& gender, const std::string& type);
};
```

Each patient node has a place for the patient's last name, first name, gender, and the type of each patient. The node class for patients is used for both walk-ins and scheduled patients, which is why the type of patients needs to be stored. The type of patients will also be written to the output file.

```cpp
void IntervalTree::printAppointments(const std::string& doctorName, const std::string& date) const {
    std::string fileName = "../Output/" + date + "_appointments_" + doctorName + ".txt";
    std::ofstream outFile(fileName);

    if (!outFile.is_open()) {
        std::cerr << "Failed to open file: " << fileName << std::endl;
        return;
    }

    outFile << "Appointments for Dr. " << doctorName << ":\n";

    std::vector<Appointment> sortedIntervals = intervals;
    std::sort(sortedIntervals.begin(), sortedIntervals.end(), [](const Appointment& a, const Appointment& b) {
              return a.startTime < b.startTime;
          });

    for (const auto& interval : sortedIntervals) {
        // Had to slave away to do this one !!!
        // Get hours and minutes
        int startHours = interval.startTime / 100;
        int startMinutes = interval.startTime % 100;

        // Get hours and minutes in end
        int endHours = interval.endTime / 100;
        int endMinutes = interval.endTime % 100;

        // output to file that is nice adn good !
        outFile << interval.firstName << " (" << interval.lastName << "): Start time: "
                << std::setw(2) << std::setfill('0') << startHours << ':'
                << std::setw(2) << std::setfill('0') << startMinutes
                << ", End time: "
                << std::setw(2) << std::setfill('0') << endHours << ':'
                << std::setw(2) << std::setfill('0') << endMinutes << "\n";
    }

    outFile.close();
```

The patients who are scheduled for the day are then written to each doctor's file. This brings the code back out to the main function and the number of appointments is outputted into the terminal. Afterward the function "printAppointments" is called; the function is part of the Interval tree class, so it can easily access the data in the interval trees. "printAppointments" creates an output file for each doctor with the scheduled patients for the day. The output file will follow the format "date + "_appointments_" + doctorate + ".txt". The date is the date that the user inputted, and the "doctorName" is one of the doctor's names. The function will then begin to write the file. The file will be named with this format: "Appointments for Dr. "doctorName" with doctorName being the doctor's name whose file is being written. The rest of the file will be the patient's last name, first name, start and end time for the appointment.

```
std::vector<int> readWalkInFilename (std::string date, int time, std::vector<Patient>& vec, std::map<std::string, IntervalTree>& doctorTrees) {

    std::string filename = generateWalkInFilename(date, time);

    std::ifstream input;
    input.open(filename);
    if (input.fail()) {
        std::cerr << "Can't open file." << std::endl;
        std::vector<int> emptyVector;
        return emptyVector;
    }

    std::string line, lname, fname, gender, strTriageLevel;
    int triageLevel;
    int patientCount = 0; // keep count of patients added
    PriorityQueue pq;

    std::getline(input, line); // get header out
    while (std::getline(input, line)) {
        std::stringstream streamline(line);
        // parse elements of line
        // ex: Medeiros,Darren,Male,5
        getline(streamline, lname, ',');
        getline(streamline, fname, ',');
        getline(streamline, gender, ',');
        getline(streamline, strTriageLevel, ',');
        triageLevel = std::stoi(strTriageLevel);

        // create patient node
        Patient currentPatient(fname, lname, gender, "Walk in");

        // add patient to linked list and vector
        vec.push_back(currentPatient);

        // add patients to priority queue
        pq.enqueue(lname, fname, gender, triageLevel);
```

```
// Function to create the filename for the input file of walk-in patients
std::string generateWalkInFilename(std::string date, int time) {
    return "../Input/" + date + "~" + std::to_string(time) + "_walk_ins.txt";
}
```

The code now begins to increment through each hour of the working day. The current time will be outputted, and the "readWalkInFilename" will be called. The "readWalkInFilename" takes the date, the time, the vector which contains all the patients' names, and the map containing all the doctor's interval trees. Once inside the function, then the "generateWalkInFilename" is called. The "generateWalkInFilename" just outputs the correct walk-in file name based on the time and the date. The file is then read, collecting the walk-in patient's first name, last name, gender, and triage level. A new patient node is created, which stores everything except the triage level.

```
void PriorityQueue::enqueue(std::string lastName, std::string firstName, std::string gender, int triage_level){

    // if nothing in list, add
    if (this->head == nullptr) {
        this->head = new PQNode(lastName, firstName,gender, triage_level);
        this->size++;
    }
    // if things in list
    else {
        if (triage_level < (this->head)->triage_level) { // if higher priority than 1st element
            PQNode* newNode = new PQNode (lastName, firstName, gender, triage_level, this->head);
            this->head = newNode;
            this->size++;
            return;
        }
        // if higher priority than rest of the elements
        PQNode* currentNode = this->head;
        while (currentNode->next != nullptr) {
            // runs until we are at the tail
            if (triage_level < (currentNode->next)->triage_level) {
                currentNode->next = new PQNode(lastName, firstName, gender, triage_level, currentNode->next);
                this->size++;
                return;
            }
            currentNode = currentNode->next; // move over to next node if condition not met
        }

        // add to end (was not higher priority than anything before the tail)
        // currentNode is now the tail after leaving the while loop
        currentNode->next = new PQNode(lastName, firstName, gender, triage_level); // points to nullptr
        this->size++;
    }
}
```

Then, the patient's name is placed into a priority queue based on their triage level, with smaller numbers given higher priority. The priority queue is then read, putting patients into the doctor interval tree. Since there are limited spaces in the interval tree, the priority queue is used to give priority to patients who need to see the doctor soon. Since the interval tree is being used the patients will not have any overlap with the doctors, so the doctor can spend the correct amount of time with each patient.

```
bool spotsLeftInHour = true;
while ( (spotsLeftInHour == true) && (pq.get_size() != 0) ) {

    PQNode* currentPatient = pq.top();

    if (currentPatient->get_gender() == "Male") {

        if (doctorTrees["Paul"].insertInterval(currentPatient->get_lastName(), currentPatient->get_firstName(), (time*100), (time*100)+14)) {
            pq.dequeue();
            walkInsAdded++;
            //std::cout << "Appointment inserted." << std::endl;
        } else if (doctorTrees["Paul"].insertInterval(currentPatient->get_lastName(), currentPatient->get_firstName(), (time*100)+15, (time*100)+29))
            pq.dequeue();
            walkInsAdded++;
            //std::cout << "Appointment inserted." << std::endl;
        } else if (doctorTrees["Paul"].insertInterval(currentPatient->get_lastName(), currentPatient->get_firstName(), (time*100)+30, (time*100)+44))
            pq.dequeue();
            walkInsAdded++;
```

When trying to make an appointment for a patient in the priority queue, a few things could happen. If there are no more available times, then the patients get turned away. If there are still spaces available, then the patient gets placed into a temporary node, then inserted based on

gender. If the patient is male, then the male doctor will be checked first for any availability if he has any available time. If the male doctor is available, then the patient is placed with him otherwise, the patient is placed with a female doctor.

After the priority queue has been read, the terminal will output the number of patients who are walk-ins, those who were able to gain an appointment, and how many patients had to be declined. The doctor's file is updated for the hour for each patient they could see. The time is then incremented by one, which corresponds to one hour.

```
--------------------
New hour: 8:00am.
--------------------
 * 4 patients walked in.
 * 4 walk-in patients successfully got an appointment.
 * 0 walk-in patients were turned away.
 * Daily schedules have been updated for each doctor. They are available in the output folder.

Press the ENTER key to continue to the next hour:
--------------------
```

The program will continue to read the walk-in files after the user hits enter. If all doctor's appointments are filled for the day, then all patients will be turned away. The program will continue to increment until the time variable equals 20, which correlates to 8:00 pm, closing time! The patient vector is then put into the radixSortByFirstName function and the radixSortByLastName function.

```
// Radix sort by first name
void radixSortByFirstName(std::vector<Patient>& arr) {
    // Find the maximum length of first names
    int maxLength = 0;
    for (const Patient& p : arr) {
        maxLength = std::max(maxLength, static_cast<int>(p.firstName.size()));
    }

    // Perform radix sort for each digit in the first name (starting from the most significant digit)
    for (int digit = maxLength - 1; digit >= 0; --digit) {
        radixSortByDigit(arr, digit);
    }
}

// Radix sort by last name
void radixSortByLastName(std::vector<Patient>& arr) {
    // Find the maximum length of first names
    int maxLength = 0;
    for (const Patient& p : arr) {
        maxLength = std::max(maxLength, static_cast<int>(p.lastName.size()));
    }

    // Perform radix sort for each digit in the first name (starting from the most significant digit)
    for (int digit = maxLength - 1; digit >= 0; --digit) {
        radixSortByDigit2(arr, digit);
    }
}
```

Both of the functions sort the vector alphabetically using Most Significant Digit (MSD) radix sort. The functions use bucket sort to facilitate radix sort. The number of buckets is set as 256 since the program assumes ASCII characters which can allow symbols, numbers, lower and uppercase letters. Both functions sort from the leftmost character, terminating at the rightmost character and treating spaces as the smallest character. For example, Al would come before Alex.

```
void radixSortByDigit2(std::vector<Patient>& arr, int digit) {
    const int NUM_BUCKETS = 256; // Assuming ASCII characters

    // Create buckets
    std::vector<std::vector<Patient>> buckets(NUM_BUCKETS);

    // Distribute elements into buckets based on the digit
    for (const Patient& p : arr) {
        int bucketIndex = (digit < p.lastName.size()) ? static_cast<unsigned char>(p.lastName[digit]) : 0;
        buckets[bucketIndex].push_back(p);
    }

    // Collect elements back into the array
    int index = 0;
    for (auto& bucket : buckets) {
        for (const Patient& p : bucket) {
            arr[index++] = p;
        }
    }
}
```

Both radix sort functions utilize the same logic, but call different functions to sort the firstName elements and the lastName elements in the vector of patients. Using the idea behind radix sort, sorting by the least significant digit first, we sort by first name and then again by last name. This results in our printed and sorted list of patients being ordered by last name, then first name.

```
// Function to write sorted objects to an output file
void writeNamesToFile(const std::vector<Patient>& arr, const std::string& date) {
    std::string filename = "../Output/" + date + "_sorted_patients.txt";

    std::ofstream outputFile(filename);
    if (!outputFile.is_open()) {
        std::cerr << "Error: Unable to open file " << filename << std::endl;
        return;
    }

    for (const Patient& p : arr) {
        outputFile << p.lastName << ", " << p.firstName << " (" << p.type << ")" << std::endl;
    }

    outputFile.close();
}
```

The final function which is called is "writeNamesToFile". This function takes the date and the patient vector as inputs. It creates a file, which follows this naming format: "date + "_sorted_patients.txt", where the date is the date inputted from the function. The patients are written into the file, where each row includes the patient's last name, first name, and if the patient was a walk-in or a scheduled patient that day.

```
--------------------
End of day at 8:00pm.
--------------------
 * 92 patient records written to day's patient records.
 * 13% of walk-ins were sent elsewhere.
--------------------
```

At the end of the day at 8pm, two statistics are output to the terminal. These are the total number of patients who were seen that day and the percentage of walk-in patients that had to be turned away. The program terminates after this completes.

Usage

Our program can be used to solve multiple challenges presented in a situation that requires scheduling. We create schedules for each doctor that can be referenced by the doctors throughout the day to see what appointments and patients they have. This helps the schedules stay clear, which is important especially as schedules are changing throughout the day from new walk-in appointments. Another problem our program tackles is it dynamically schedules for new walk-ins. As the day goes on, our program keeps track of appointments, adding in new appointments where it can but making sure no appointments overlap.

Another problem the program solves is with storing patient information and keeping track of counts of patients seen or sent away throughout the day. Storing information is important for offices to keep track of data about their patients and who they are seeing, such as patient records,

as well as for analyzing their doctor, space, and time utilizations. If it is found that no walk-in patients are being turned away, more appointments could be made ahead of time to better fill the schedule. If most or all walk-in patients are getting turned away, this could indicate to management that they are scheduling too many appointments ahead of time or that they might have enough demand to hire another doctor to address the high demand for appointments.

The program user could be a front desk receptionist using it to schedule appointments for a pediatric office. Before running the program, an input file must be added by the program user to reflect the patients that have appointments that day. This start of day input file should be named “YYYY-MM-DD_scheduled.txt”. Once this file is completed, the program is run by navigating to the Programming folder and running “./main YYYY-MM-DD” in the terminal. Before the program attempts to read in files for the new hour, the user must press enter on their keyboard when prompted. Before pressing enter, the user should complete the input file for that hour, making sure it is saved and correctly formatted. These hourly files for walk-ins are required from 8am to 7pm. They should be named “YYYY-MM-DD-HH_walk_ins.txt”, where if the time is 7am, the hour, HH, is just 7, and if the time is 4pm, the hour, HH, is 16 for military time. After 7pm passes, the program will complete the actions for that hour and finally the end of day actions, as already described above.

Overall, this program can be used in a variety of different real-world situations, since the program can do a lot and be applied to a real-life pediatrician's office. The first part of our program creates doctors' schedules, which can plan appointments for multiple doctors each day and prevent any appointments from overlapping. This is important so a doctor can spend the correct amount of time with a patient without any rush or worry of overlapping appointments. The interval tree used in our program allows us to update the doctors' schedules for the day,

showing times when the doctor is free. It can also easily allow any walk-in patients to be inserted into a doctor's schedule in real-time or be turned away if the doctor has no more space for the hour. The program also has the feature of printing each doctor’s interval tree every hour so that the doctors' schedules are visible to all, allowing for better planning and up-to-date schedules. This is useful since a person who reviews the outputs can decide if another doctor needs to be hired based on the number of successful walk-ins and appointments for each day. Our program also stores the information of each patient who enters the office, which can be useful in analyzing patient trends and could be used to improve the pediatrician's office.

Use Cases

The implications of usage in real-life situations are important for motivation as to why a program has been built. We designed our program with a pediatric office in mind, but other medical offices could also make use of this easily. A pediatric office or other medical offices would find this useful for making schedules for doctors and scheduling any walk-in patients that may occur. A hospital could use this program to schedule patients based on how soon they are going to die. It uses a similar concept of urgency of care needed, but the stakes at a hospital would be higher. Since a priority queue is used for walk-ins, a hospital could use this to implant a system so that doctors are going to see patients first who are closer to death and patients who are fine last. A hospital might not have any appointments to be read in that are scheduled, and instead all of their patients would be considered walk-ins.

Apart from these, other situations where scheduling is required could make use of our program, with some tweaking. A metric for priority isn’t required for our program to work, as all priorities could be set to the same value so that the priority queue operates just like a normal

queue, which would give people appointments in the same order that they entered the queue. Instead of each interval tree representing a doctor, each interval tree could represent a room in a building and our program could be used for room reservations instead. Any situation that involves scheduling could use our program as a framework.

Strengths

Modularity and Organization

Our program is organized into distinct folders for input, output, and programming files. This organization makes it easy to locate and manage different types of files. Additionally, we separated the program code into different files based on functionality (such as interval tree, priority queue, patient-related functions). This enhances modularity and maintainability, allowing for easier modification, testing, and reuse of our code. Furthermore, our output files are well-organized and formatted in a clear and understandable manner. They utilize a standardized format with clear headings and well-structured data, ensuring readability and ease of interpretation for users. This makes it simple to incorporate the output into spreadsheets or calendars for better aesthetics and analysis.

Efficient Sorting and Access

The structure of our code efficiently utilizes sorting algorithms like bucket sort and radix sort, ensuring that patient sorting is performed only when necessary. Despite current test files containing a small number of patients, these algorithms are well-suited for large datasets, indicating strong scalability. This approach minimizes computational overhead and optimizes

sorting performance, enabling our program to handle increased data volumes efficiently. After sorting, looking up a name also takes less time, as binary search would take O(log n) time.

Efficient Memory Usage

Our program only stores patient information where needed, reducing memory usage. Initially, each patient and their information is stored in a vector of Patient instances. Appointments are directly slotted into interval trees, while walk-ins are passed to priority queue nodes, then slotted into interval trees (or turned away). The priority queue is then cleared, freeing memory to be reused. At the end of the day, the interval trees are written to text files, then cleared as well. The only memory usage remaining is our patient vector, which is sorted by name then printed for recordkeeping. Thus, our program dynamically allocates memory as required, releasing resources when they are no longer needed to avoid unnecessary memory consumption. This approach ensures that our program remains lightweight and responsive, even when dealing with large datasets or prolonged execution times.

Weaknesses

Error handling

Our program requires input files in a specific format to function, and cannot handle missing information. For example, if a walk-in patient is not listed with a triage level, the priority queue algorithm breaks and terminates the program instead of prompting for the level. Our current level of error handling is enough to demonstrate the program's usage, but would need to be reworked before deployment in an official setting.

Overload management

Our walk-in scheduling algorithm runs at the top of each hour, turning away patients who cannot be scheduled within that hour. This approach becomes problematic when faced with a large influx of walk-ins, as the majority are turned away rather than being scheduled for later in the day. This can lead to patient dissatisfaction, overcrowding, and bottlenecks within the healthcare facility.

Encryption

When working with medical records, privacy is crucial, as there are strict laws regarding the distribution and viewing of personal medical information. Our program does not encrypt any of the patient information after reading it in. This lack of encryption poses a significant security risk, as it means that sensitive patient data is transmitted and stored in plain text format, making it vulnerable to unauthorized access or interception by malicious actors. Without encryption, patient information is exposed to potential breaches, data leaks, or identity theft, undermining patient confidentiality and violating regulatory compliance standards such as HIPAA (Health Insurance Portability and Accountability Act) in the United States or GDPR (General Data Protection Regulation) in the European Union. Encrypting patient information would provide an additional layer of security, ensuring that sensitive data remains protected both during transmission and storage, thereby safeguarding patient privacy and maintaining regulatory compliance.

Hard-Coding

Currently, there is a section in our main function that is hard coded. When deciding which interval tree to insert the walk-ins, we have multiple if statements checking each doctor's tree, with the doctor names as parameters. This means that if we want to add or remove doctors, we would have to manually change the code, which could lead to mistakes. Additionally, it might be difficult to adjust the program to fit different clinic setups or workflows. Using a more flexible approach could help make the program easier to update and adapt to different situations.

Limitations

With every program, there are limitations. One limitation of our program is the current state of the program only allows scheduled patients and walk-in patients to be evaluated and updated every hour. We have a function for updating the time, so this could be updated to update the time more frequently, with some tweaking in our program to reflect this. Another limitation is that the walk-ins for a given hour need to be known at the start of the hour, as they are read in through a text file. In real life, walk-ins do not all come in at the same time. So, it is possible that walk-ins in our program would have to wait a full hour before getting added to the priority queue to potentially get an appointment. For example, if a walk-in came in at 9:01am, they could be added into the input file but then not read in until the program is run for the 10am hour. To improve this, shorter time intervals could be used so that the program schedules more frequently than just every hour. Alternatively, the program could be adjusted so that walk-ins are read in through the terminal so to avoid having an input file that a group of walk-ins are added into. For our program, we use the input file approach as this makes it easier to test and demonstrate our program.

Another limitation to the program is the lack of patient data being read and stored. Currently, we are only reading and storing the patient's name and doctor preference, which may not provide sufficient information for comprehensive patient management and analysis. Important medical information such as patient history, diagnoses, medications, allergies, and treatment plans are not being inputted or stored, limiting the program's ability to support informed decision-making and personalized care delivery. Adding variables to store a broader range of patient data would improve the quality of care and support more effective clinical decision-making processes.

Future

Finally, while developing this program, we realized that there are still many features that could be added. This section expounds upon several such features, describing their use cases, implementation, and real world usage.

One of the features would be to add the ability for doctors to call out sick and move their appointments around. In real-life situations, doctors are bound to get sick, and when this occurs you do not want people to have their appointments canceled. If a doctor called out sick, the patients they have scheduled for the day would be rearranged based off of their original appointment time or be informed that their time and/or doctor have been changed. If they were rescheduled by the system, the patient could accept or decline the new appointment. If the patient chose to decline, they would be able to make an appointment based on future availability. Likewise, we could also include options for the patients to cancel or reschedule their appointments themselves, for example if they get sick or have another conflict.

Another feature that could be added would be the ability to give the doctors more flexibility with their work schedule. The interval trees would just have ‘break’ intervals inserted for lunch times and other commitments. If a doctor needs to be off for the day or can only work for half of the day then a break slot would be inserted when the doctor can’t work. This method would also be used if a doctor has a planned day off. The interval tree would be filled with nodes that state ‘off’ instead of ‘break’. A similar method would be implanted if the doctor is sick. If the doctor calls out sick at the start of the day then the tree would be set with nodes that state ‘sick’. If the doctor goes home early midway through the work day, the appointments scheduled for times after the doctor leaves would be replaced with ‘sick’, and the patients who are scheduled for the day would be rescheduled for the same day with a different doctor or for a different day.

Another feature that could be added is for the program to run over multiple days, even years, keeping a record of past patients. Since most pediatric offices run seven days a week, the program should archive at the end of each week. To achieve this, the main function would be changed to output the date and then to only output that day's data. Allowing the program to run for multiple days can also allow patients to schedule appointments further in the future and see doctors available over the next few days following the original appointments.

To make the program more user-friendly, we could add a front end to the program which would make many things easier to view. On the website side of things, HTML, CSS, and JavaScript could be used to display all the features that are currently being used. In the case of an app, React, a web interface language, could be used. The front end would display a calendar for each doctor and their availability throughout each week. To protect the privacy of others and the doctor's privacy the calendar times would be listed as busy or free. The patient can schedule an

appointment in the ‘free’ time. When creating an appointment, the patient would have the ability to log in or sign up, lessening the risk of misused records. This interface would also be used by a receptionist. If a patient walks in, the receptionist could schedule them for any available doctor or put them into a queue.

A feature that is quite important to many pediatric offices is a late policy. This feature could be implanted into the code as a check-in policy and a late policy—if a patient has not checked in between 30 minutes before their appointment and up until the time of their appointment, then the appointment is canceled and a walk-in who is currently in the queue could take the patient's spot. The way this could be implanted would be that in the doctor’s interval trees would be a boolean which would be checked in. If the patient has not checked in before the appointment time, which sets the boolean to true behind the scenes, then the appointment is canceled.

Conclusion

Our group created a scheduling program for use at a pediatric office. It schedules new appointments throughout the day based on the urgency of care needed for each walk-in, for each hour, to allocate available appointment times to walk-in patients. New appointments are added so there are no scheduling conflicts, or no overlapping appointments for the same doctor and time. Our program takes text files as input for the appointments scheduled for the day, and input files for each hour of the walk-ins with their urgency levels. It outputs schedules for each doctor at the start of the day and every hour once walk-in appointments have been added. It outputs a final sorted list of patients who came into the office that day.

Our program used interval trees to prevent adding overlapping appointments. We used priority queues to take into account patients' urgency of care when determining which walk-ins will get appointments before no more are available for the hour they came in. We use radix sort and bucket sort when outputting a list of patients of the day, sorted by last name and first name.

To summarize, our program allows the user to create schedules and easily add walk-in appointments. The textbook user for this program is a front desk receptionist, who can also monitor current appointments and patient information. Strengths of our program include modularity and organization, efficient sorting and access, and efficient memory usage. Potential weaknesses of our program include error handling, overload management, encryption, and hard coding. The limitations of our program are that walk-in patients are only updated every hour, and that not enough information is being stored on each patient. In the future, we could implement break slots in the doctor's interval trees and add a late policy for patients. Finally, we could create a user interface for scheduling. We believe our program effectively solves the scheduling

problem presented in the given pediatric office scenario, and with alterations and new features added, can expand to even more use cases.

References

Interval Tree

Wikimedia Foundation. (2024, February 23). *Interval tree*. Wikipedia. https://en.wikipedia.org/wiki/Interval_tree

GeeksforGeeks. (2023, April 23). *Interval tree*. https://www.geeksforgeeks.org/interval-tree/#

Engibaryan, R. (2024, March 18). *Difference between segment trees, interval trees, range trees, and binary indexed trees*. Baeldung on Computer Science. https://www.baeldung.com/cs/tree-segment-interval-range-binary-indexed#:~:text=An%20interv

Anand, A. (n.d.). *Interval tree*. https://astikanand.github.io/techblogs/advanced-data-structures/interval-tree

Node-interval-tree. npm. (n.d.). https://www.npmjs.com/package/node-interval-tree

Interval trees. cmu. (n.d.). https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/intervaltrees.pdf

Priority Queue

Wikimedia Foundation. (2024b, April 5). *Queueing theory*. Wikipedia. https://en.wikipedia.org/wiki/Queueing_theory

GeeksforGeeks. (2023a, January). *What is priority queue: Introduction to priority queue*. https://www.geeksforgeeks.org/priority-queue-set-1-introduction/

GeeksforGeeks. (2023b, January). *Basic operations for queue in Data Structure*. https://www.geeksforgeeks.org/basic-operations-for-queue-in-data-structure/

Plessis, J. du. (2018, August 1). *Data Structure Stories: Stacks and queues*. Medium. https://medium.com/@duplessisjdp96/data-structure-stories-stacks-and-queues-1826a7e92026

Radix sort

Wikimedia Foundation. (2024c, April 14). *Radix sort*. Wikipedia. https://en.wikipedia.org/wiki/Radix_sort

Yadav, P. (2022, February 24). *Radix sort algorithm in data structure (with code in python, C++, java and C)*. Scaler Topics. https://www.scaler.com/topics/data-structures/radix-sort/

GeeksforGeeks. (2024, April 19). *Radix sort - data structures and algorithms tutorials*. https://www.geeksforgeeks.org/radix-sort/

Bucket sort

Allan. (2024, February 26). *Move over bubble & bin: Bucket sort is the fastest sort!*. tl;dv. https://tldv.io/blog/move-over-bubble-bin-bucket-sort-is-the-fastest-sort/

GeeksforGeeks. (2024a, March 27). *Bucket sort - data structures and algorithms tutorials*. https://www.geeksforgeeks.org/bucket-sort-2/

Hebert, J., Tsai, J. *Bucket sort*. prezi.com. (2013). https://prezi.com/2_cf_jf-pzv_/bucket-sort/