Final Report

Nathaniel Pyo, Jimmy Cheung

11/14/2024

Project Scope

The aim of this project is to construct an algorithm that receives patient information and organizes a priority queue structured around two principles: severity and wait time. Patients classified as high severity will be treated first, but lower severity patients may be prioritized once a certain wait time threshold is reached.

The project's main objectives are:

- To enable hospital staff to provide effective and immediate care without bias.
- To prevent excessive delays and address patients' needs in a time-sensitive manner.
- To incorporate a timer threshold override that adjusts prioritization based on severity automatically.

The expected outcomes include a functioning C++ system, documentation, and final project report.

Project Plan

Timeline

The overall timeline for the project is divided into phases:

- **Week 1 (October 7 - October 13)**: Completed Iteration 2: timeline, scope, roles, skills/tools, repository.
- **Week 2 (October 14 - October 20)**: Began development of priority queue functionality testing, updated documentation.
- **Week 3 (October 21 - October 27)**: Completed basic skeleton code for algorithm with transparency for debugging, updated documentation for implemented functions
- **Week 4 (October 28 - November 3)**: Fixed edge case segmentation errors and finalized testing, Cleaned up output for readability.
- **Week 5 (November 4 - November 10)**: Detailed pseudocode for all functions in algorithm, documented test results section.
- **Week 6 (November 11 - November 17)**: Finalize deliverables, completed documentation.
- **Week 7 (November 18 - November 28)**: Project completion: presentation and report submission.

Milestones

Key milestones include:

- Project Scope and Plan (October 10).
- GitHub Repository Setup and Initial Development (October 10).
- Skeleton Algorithm with Base Functionality (October 25).
- Algorithm Filled out and Initial Testing (November 4).
- Finalize Algorithm with Testing (November 6).
- Finalize Report and Presentation (November 16).
- Submit Report and Presentation (November 17).

Team Roles

- **Nathaniel Pyo:** Constructed pseudocode and core implementation, conducted testing and debug.
- **Jimmy Cheung:** Documentation and analysis, algorithm walkthrough and technical brief.

  Both team members executed components crucial for the algorithm, documentation, and presentation. Roles were delineated but not exclusive. Any amount of pseudocode or implementation was supplemented with documentation and analysis, regardless of which member was responsible. In doing so, deadlines were met and workflow was streamlined.

Data Representation

All of the implementation for this project was constructed and modified through the project repository on GitHub, open to all members: `https://github.com/jimmykimo08/ECE2560_FinalProject`.

Across the priority queue class, there are 5 data members accessed across all algorithms. Priority queue automatically sorts and holds patient severity in integer form, and three vectors are created to store the order and start time point at which each patient's severity was added to the queue along with a patient ID for numbering. A variable to store the last ID was also necessary for reference as the ID vector is updated.

- Priority Queue: Primary data structure, holds integer values to represent severity level
- System Workflow: Patient arrival, Priority management, Queue display
- Data Flow and Structure: `pq`, `order`, `timers`, `patientID`, `lastPatient`
- Insertion, serving, synchronized erasing based on order and timers

Algorithm Design

Algorithm 1: Add Patients to Queue and Serve

```
Input: code integer code
Output: patients added and served
Begin Algorithm: nPatients
int numPatients = (rand from 0-3); int arr[numPatients];
if code == 0 then
    cout << "Patients added: ";
    for int i=0; i<numPatients; i++ do
        arr[i] = (rand from 1-3);
        timers.pushback(std::chrono::systemclock::now());
        order.pushback(arr[i]);
        patientID.pushback(lastPatient+i+1);
        cout << "P" << lastPatient+i+1 << "(" << arr[i] << ") ";
    end
    if !patientID.empty() then
        lastPatient = patientID.back();
    end
    cout << endl;
end
int proceed = -1;
if numPatients != 0 —— !pq.empty() then
    proceed = proceed = maxTime(timers[0]);
end
if proceed == -1 then
    if code == 0 then
        for int i=0; i<numPatients; i++ do
            pq.push(arr[i]);
        end
    end
    servePatient(proceed);
end
else
    servePatient(proceed);
    if code == 0 then
        for int i=0; i<numPatients; i++ do
            pq.push(arr[i]);
        end
    end
end
End Algorithm: nPatients
```

`nPatients` generates a random number of patients from 0-3 to approach the window. Each patient is then assigned a random severity from 1-3. As priority queues automatically sort and only preserve order for identical elements, the order at which each patient arrives is recorded, along with start time point. The patient is then assigned a unique ID based on arrival order. The algorithm accepts an operation code, which indicates whether the window is open or closed for the day. If the window is closed, the above process is bypassed as no new patients are to be added. Following, the algorithm calls function `maxTime`, which will check if a patient has been waiting past a set wait time threshold. If the max time is not passed, the patients are added to the priority queue and then function `servePatient` is called to serve the next patient. If max time passed, the corresponding patient's position in line is sent to `servePatient` for prioritization, and new patients are added afterwards.

Algorithm 2: Check if Max Time Threshold has been Reached

```
Input: start timepoint $str$
Output: output code indicating case
Begin Algorithm: maxTime
chrono::timepoint<std::chrono::systemclock> stp;
stp = chrono::systemclock::now();
chrono::duration<double> dur = stp - str;
double wT = dur.count();
if $wT > 3$ then
    for int i=0; i<timers.size(); i++ do
        if timers[i] == str then
            return i;
        end
    end
end
return -1;
End Algorithm: maxTime
```

`maxTime` receives a start time point, from which the current time point is used to determine the duration for which the patient has been waiting. If the duration is past a set threshold, the index of the element for which a match to the received start time point is found is returned. Note that the first patient in the arrival order vector will always have waited longer than any patients behind them, and the timers vector is synchronized to the order vector. If the duration is not past a set threshold, the base output code is -1.

Algorithm 3: Match Arrival Order Placement with Severity Queue

```
Input: none
Output: index for first patient with highest severity
Begin Algorithm: matchOrderPrio
for int i=0; i<order.size(); i++ do
    if order[i] == pq.top() then
        return i;
    end
end
return 0;
End Algorithm: matchOrderPrio
```

`matchOrderPrio` simply searches the arrival order array and returns the first index at which there is a match for the top element in the priority queue. This function is called in `servePatient` in order to determine which patient should be erased from the arrival order, timers, and ID vectors once served.

Algorithm 4: Find Index of Patient in Severity Queue

```
Input: placement of patient in arrival order line index
Output: position of patient in priority queue
Begin Algorithm: matchPrioOrder
priorityqueue<int> copy = pq; int pos = 0; while !copy.empty() do
    if copy.top() == order.at(index) then
        return pos;
    end
    copy.pop(); pos += 1;
end
return 0;
End Algorithm: matchPrioOrder
```

Similarly to `matchOrderPrio`, `matchPrioOrder` does somewhat of the reverse operation in which the priority queue is traversed until the first match between the arrival order vector element at the received index and the priority queue. The position in the priority queue is then returned. This function is used when the wait time threshold has been reached and a specific patient in line needs to be served before the top patient in the priority queue.

Algorithm 5: Print Served Patient and Remove from Lines

```
Input: integer code code
Output: patient served based on highest severity or past max wait time
Begin Algorithm: servePatient
if pq.empty() then
    cout << "Waiting for customers..." << endl;
    return;
end
if code == -1 then
    int ind = matchOrderPrio();
    cout << "Serving patient P" << patientID[ind] << " with severity: " << pq.top() << endl;
    order.erase(order.begin()+ind);
    timers.erase(timers.begin()+ind);
    patientID.erase(patientID.begin()+ind);
    pq.pop();
end
else
    cout << "Max wait time reached. Serving patient P" << patientID[code] << " with severity: ";
    cout << order[code] << " first." <<endl;
    priorityqueue<int> copy = pq;
    priorityqueue<int, vector<int>, greater<int>> temp;
    while !copy.empty() do
        temp.push(copy.top());
        copy.pop();
    end
    int pos = matchPrioOrder(code);
    for int i=0;i<pq.size() - pos;i++ do
        temp.pop();
    end
    for int i=0;i<pos+1;i++ do
        pq.pop();
    end
    while !temp.empty() do
        pq.push(temp.top());
        temp.pop();
    end
    order.erase(order.begin()+code);
    timers.erase(timers.begin()+code);
    patientID.erase(patientID.begin()+code);
end
End Algorithm: servePatient
```

`servePatient` has two cases depending on received code, similarly to `nPatients`. The first case is called when wait time threshold has not been passed. Function `matchOrderPrio` is called to find the index to erase in the vectors once the patient has been served and popped from the priority queue. The patient ID being served is printed to the output window along with their severity, and their information is removed from all active data structures.

If the wait time threshold has been passed, the patient ID is served first and printed to the output window with their severity. In order to remove the patient from the priority queue while preserving preceding information, the priority queue is copied to a reverse priority queue. Function `matchPrioOrder` is then called to determine the position in the priority queue of the patient being served. The priority queue and reverse priority queue are both popped up until and including the patient, and then the elements of the reverse are added back. The served patient's information is then removed from all active data structures.

Algorithm 6: Display Patient Line

```
Input: none
Output: print patient line in arrival order with severity
Begin Algorithm: dispOrder
if order.empty() then
    cout << "Line is empty." << endl;
end
else
    cout << "Arrival Order: ";
    for int i = 0; i < patientID.size(); i++ do
        cout << "P" << patientID[i] << "(" << order[i] << ") ";
    end
    cout << endl;
end
End Algorithm: dispOrder
```

`dispOrder` simply prints the state of the line in an intuitive format to the output window. The patient ID is first, followed by the patient severity in parenthesis. The order of this line is by arrival order and not by treatment priority, but treatment is operated by treatment priority.

Algorithm 7: Driver Code: Window Operation

```
Input: none
Output: output code 0
Begin Algorithm: main
chrono::timepoint<std::chrono::systemclock> open = chrono::systemclock::now();
double close;
PriorityQueue patientLine;
srand((unsigned)time(0));
while close < 5 do
    patientLine.nPatients(0);
    patientLine.dispOrder();
    chrono::timepoint<std::chrono::systemclock> curtime = chrono::systemclock::now();
    chrono::duration<double> dur = curtime - open;
    close = dur.count();
    thisthread::sleepfor(chrono::seconds(1));
end
cout << "Window has closed, no new patients." << endl;
while !patientLine.isEmpty() do
    patientLine.nPatients(1);
    //patientLine.dispLine();
    patientLine.dispOrder();
    thisthread::sleepfor(chrono::seconds(1));
end
return 0;
End Algorithm: main
```

The driver code runs the window operations, along with the opening and closing time window. The operation window is kept open for a set time, for which during new patients are added to the line. The basic method of operations is as follows: patients are generated and added to the line, patients are served by severity unless the wait time threshold has been reached, and a slight delay is implemented for readability. The output window sees the following: any new patients generated with patient ID and severity, the current patient being served, and the state of the line afterwards.

Once the window has closed, `nPatients` is called with a different code, indicating the new patients should not be generated. Should any patients remain in line after the window is closed, they will also be treated by severity unless the wait time threshold has been reached.

Time Complexity

nPatients

Initialize random number of patients: 1
If statement to check if window open: 1
Assign random severity for each patient: n
Update timer, order, ID, and disp each patient: 4n
Update last patient ID to use for next patient ID reference: 2
If new patients or line is not empty: 1
Call maxTime to check threshold, which has complexity O(n): n
If wait time not reached and window open: 2
For all patients, add to priority queue, push is O(log(n)): n*log(n)
Call servePatient, which is O(n*log(n)): n*log(n)
If wait time reached, call servePatient with different code: n*log(n) + 1
If window open, add new patients after: n*log(n) + 1
Total: 4n*log(n) + 6n + 9
Time Complexity: O(n*log(n))

maxTime

Intialize and obtain current time point: 2
Calculate duration from input time point to current: 2
If wait time is past threshold, return first match in timer array: n + 1
Else, return code -1: 1
Total: n + 6
Time Complexity: O(n)

matchOrderPrio

Iterate over order vector: n
Return index at first match with priority queue top element: n + 1
Return 0 if no match: 1
Total: 2n + 2
Time Complexity: O(n)

matchPrioOrder

Copy priority queue for iteration and init. index: 2 Iterate over copied priority queue: n
Return first match between copied pq top element and order at index: 1
Pop copied pq element for iteration, pop is O(log(n)): n*log(n) Update position index: n
If no match, return 0: 1
Total: n*log(n) + 2n + 4
Time Complexity: O(n*log(n))

servePatient

Case for empty severity queue: 3
Base case, severity prioritization: 1
Call match function and remove order timer and ID data at index: n + 3
Serve patient with highest priority and pop from pq: log(n) + 1 Second case if wait time threshold passed,

print exception: 3
Copy severity queue and reverse: 2n*log(n) + 2
Call match and pop reverse queue until and including match: 2n*log(n)
Pop severity queue until and including match: n*log(n)
Add reverse queue elements back into severity queue: 2n*log(n)
Remove matching data attributes from order timer and ID vectors: 3
Total: 7n*log(n) + n + log(n) + 16
Time Complexity: O(n*log(n))

dispOrder

Base case if order vector is empty: 2
If not empty, iterate over vector and print ID and severity: n
Total: n + 2
Time Complexity: O(n)

main

Init. time points, patientLine, random seed: 4
While loop when window is open, this check happens worst case 7 times from testing: 7
Call nPatients with code 0 indicating open window: 7*n*log(n)
Call dispOrder to print remaining line: 7*n
Obtain current time point and calculate duration: 7*3
Timed delay for readability: 7*1
Print window closed message: 1
While patients are still in line: $k < n$
Call nPatients with code 1 to indicate closed window: k*n*log(n)
Call dispOrder to print remaining line: k*n
Timed delay for readability: k*1
Exit code 0: 1
Total: (k+7)n*log(n) + (k+7)n + (k+41)
Time Complexity: O(n*log(n))

Results

Below are the results for a typical output window. While the window is added, anywhere from 0-3 patients are added with priorities 1-3. The patient with the highest severity is treated first unless a patient has been waiting for longer than a set wait time threshold, in which they are served first.

```
Patients added: P1(1)
Serving patient P1 with severity: 1
Line is empty.
Patients added:
Waiting for customers...
Line is empty.
Patients added: P2(3) P3(1) P4(1)
Serving patient P2 with severity: 3
Arrival Order: P3(1) P4(1)
Patients added: P5(2) P6(1)
Serving patient P5 with severity: 2
Arrival Order: P3(1) P4(1) P6(1)
```

```
Patients added:
Serving patient P3 with severity: 1
Arrival Order: P4(1) P6(1)
Patients added: P7(1)
Max wait time reached. Serving patient P4 with severity: 1 first.
Arrival Order: P6(1) P7(1)
Window has closed, no new patients.
Max wait time reached. Serving patient P6 with severity: 1 first.
Arrival Order: P7(1)
Serving patient P7 with severity: 1
Line is empty.
```

More examples are provided in Appendix B, which contains output windows for severity vs. arrival order and other cases that may not be depicted in this example.

Discussion

There were two key findings that made the implementation streamlined: input order is only preserved for priority queues when elements are identical, and logically, the patient who was added first will have waited for the longest time. In response to the first finding, the matching function operates by finding the first match between a priority queue and other data attributes in order to accurately erase all attributes pertaining to patients that are already served without erasing data of patients in line. For the second finding, the wait time threshold check and resultant time complexity of the overall algorithm was simplified by only needing the check the total wait time of the patient that arrived first in line. Even if other patients may have passed the wait time threshold, the patient first in line would have waited for longer, and according the project objectives priority was given on a wait time basis instead of severity once the wait time threshold was passed.

Limitations to the algorithm come in the form of needing to access 4 different data structures across all functions. Including the integer variable to keep track of the last patient ID number, while some data structures were non-essential to core functionality and only added for cleaner and readable output, the arrival order vector was essential for preserving arrival order to compensate for the priority queue's lack of doing so. These data structures are also dynamically modified, being added and taken from throughout.

Conclusion

The algorithm is optimized for time efficiency by utilizing priority queues. Upon investigation of priority queues, the time complexity of pushing or popping an element is O(log(n)). Essentially then, any base operations with priority queue elements will be O(log(n)), and dealing with array elements to be iterated over, the minimal complexity is O(n*log(n)). This holds consistent with typical sorting algorithms, in which the auto-sort capabilities of priority queues provide an efficient method for implementations that require sorting. For the hospital patient management system, the use of priority queues allowed for intuitive and time efficient implementation of prioritization, and the override functionalities for the wait time threshold did not significantly add to the time complexity. Next steps may be to identify potential clients, implement a GUI for more accessible output, and expansion to include other functionalities, including additional prioritizations or overrides.

Testing Results

Much of early testing was employed to ensure that the patient information being recorded and erased when necessary was consistent across all data structures. A faulty index or a mismatch between these would result in a mis-prioritization of patients and other errors that would halt operation. The conclusive results led to the implementations of functions `matchOrderPrio` and `matchPrioOrder` to better track the indices being called and sent across each function. The display functions originally printed both the arrival order and the priority queue to allow for a manual tracing of each patient's severity and order in the arrival line. When a patient with x severity was served, correct functioning led to the same severity being removed from the order line.

Extensive testing was employed in order to ensure proper functioning when the max time threshold was reached. At first, the wrong patient would be served, or there would be inconsistencies across the data structures. Small changes in iteration logic, like erasing the priority queue and reverse priority queue up until and including the served patient, largely fixed the problem. Additionally, the original code in `nPatients` would iterate through the entire timers vector to check each time instance for whether or not the max time was passed. However, not only did this lead to decrease efficiency, but small errors in logic led to output results that suggested the max time was never passed. To restore proper functioning, only the first element in the timers vector was sent to `maxTime` under the impression that if the earliest patient to arrive has not passed the threshold, then the others have not either. If the threshold has been reached, then the earliest patient naturally would have been waiting the longest, so they will be prioritized.

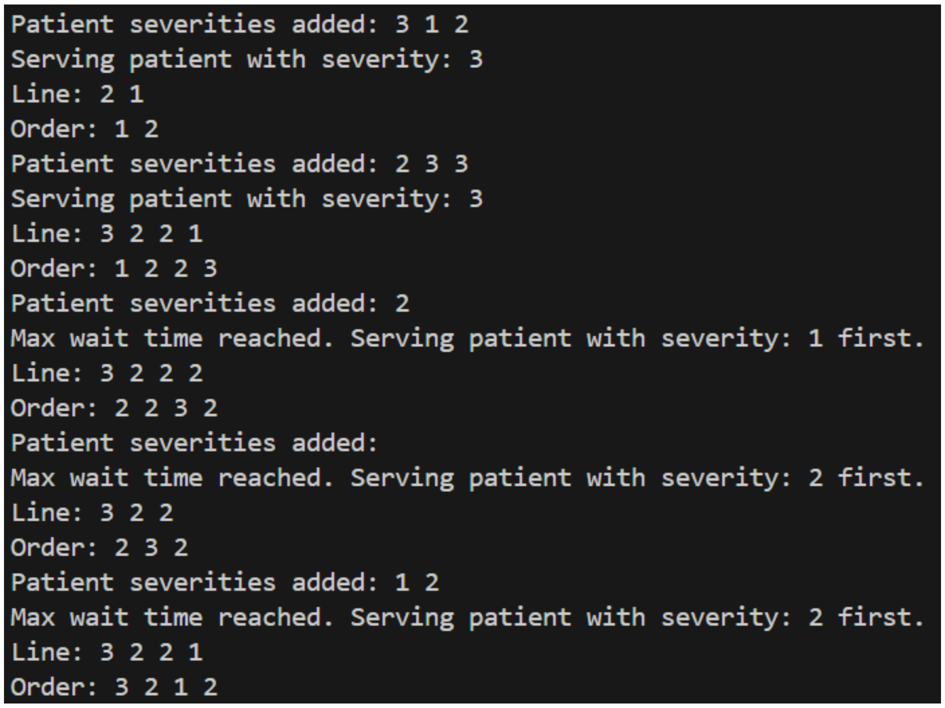
Following variable and index tracking, the most common error was in encountering segmentation faults once standard operation was ensured. Most oversights were exposed when the `nPatients` generated no new patients for the very first pass, or when the line became empty during one of the passes due to all patients being served and no new patients being generated on the next pass. Default cases for empty data structures and expanding logic to only call certain functionalities for non-empty structures largely fixed issues.

References

Priority queue functionality and basic logic was studied and analyzed prior to testing to better formulate effective implementation. Additionally, the references were used to understand the time complexity of pushing and popping elements from priority queues. The following information on priority queues served useful for this project:

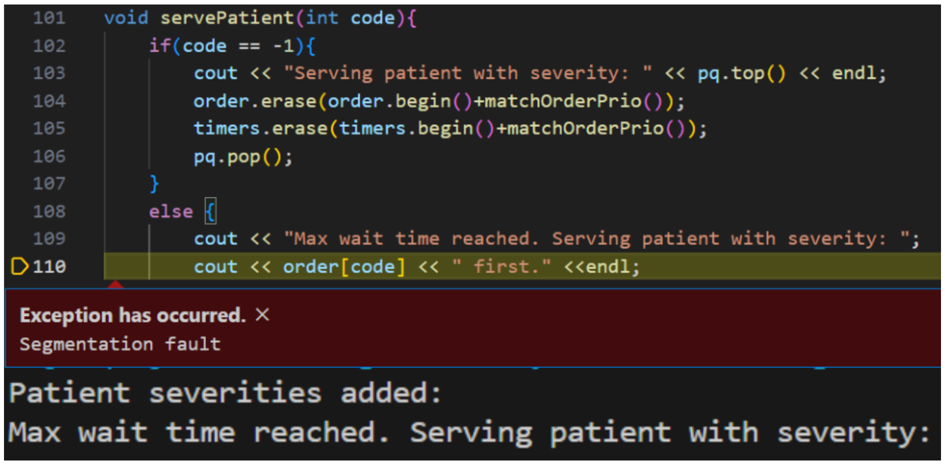
GeeksforGeeks. (2024, October 11). Priority queue in C++ Standard Template Library (STL). GeeksforGeeks. https://www.geeksforgeeks.org/priority-queue-in-cpp-stl/

Appendix A: Test Figures and Cases



```
Patient severities added: 3 1 2
Serving patient with severity: 3
Line: 2 1
Order: 1 2
Patient severities added: 2 3 3
Serving patient with severity: 3
Line: 3 2 2 1
Order: 1 2 2 3
Patient severities added: 2
Max wait time reached. Serving patient with severity: 1 first.
Line: 3 2 2 2
Order: 2 2 3 2
Patient severities added:
Max wait time reached. Serving patient with severity: 2 first.
Line: 3 2 2
Order: 2 3 2
Patient severities added: 1 2
Max wait time reached. Serving patient with severity: 2 first.
Line: 3 2 2 1
Order: 3 2 1 2
```

Figure 1: Iteration 3/4 Output for Variable Tracing

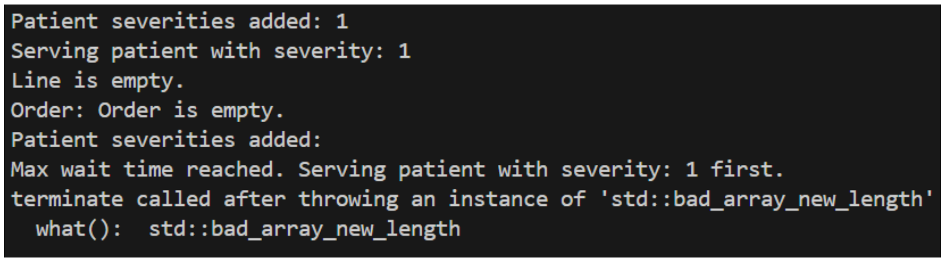


```
void servePatient(int code){
    if(code == -1){
        cout << "Serving patient with severity: " << pq.top() << endl;
        order.erase(order.begin()+matchOrderPrio());
        timers.erase(timers.begin()+matchOrderPrio());
        pq.pop();
    }
    else {
        cout << "Max wait time reached. Serving patient with severity: ";
        cout << order[code] << " first." <<endl;
```

```
Exception has occurred. ×
Segmentation fault
```

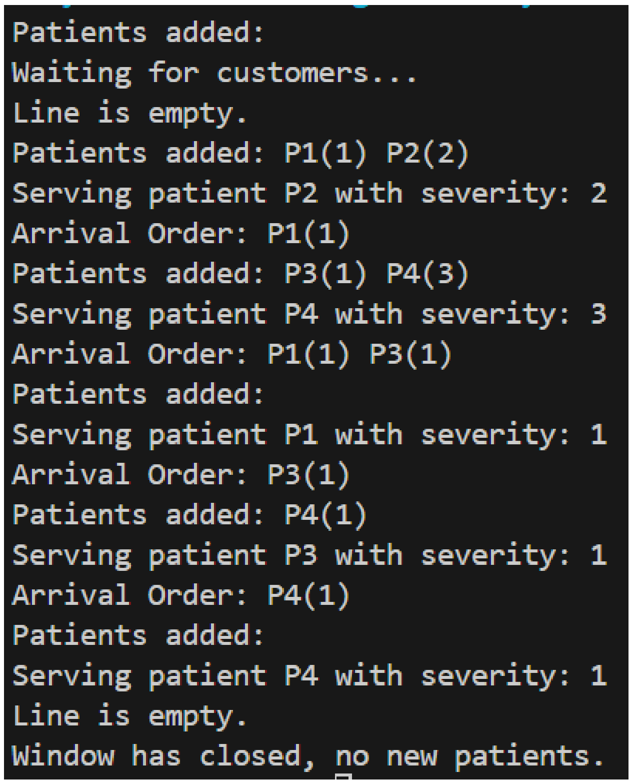
```
Patient severities added:
Max wait time reached. Serving patient with severity:
```

Figure 2: Error Case 1: No Patients Added at Start



```
Patient severities added: 1
Serving patient with severity: 1
Line is empty.
Order: Order is empty.
Patient severities added:
Max wait time reached. Serving patient with severity: 1 first.
terminate called after throwing an instance of 'std::bad_array_new_length'
  what():  std::bad_array_new_length
```

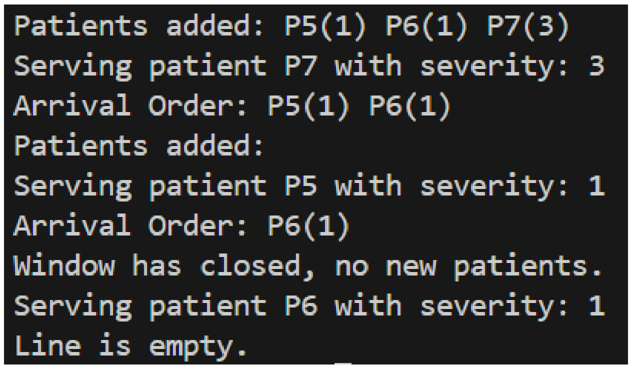
Figure 3: Error Case 2: Line is Empty During Operation



```
Patients added:
Waiting for customers...
Line is empty.
Patients added: P1(1) P2(2)
Serving patient P2 with severity: 2
Arrival Order: P1(1)
Patients added: P3(1) P4(3)
Serving patient P4 with severity: 3
Arrival Order: P1(1) P3(1)
Patients added:
Serving patient P1 with severity: 1
Arrival Order: P3(1)
Patients added: P4(1)
Serving patient P3 with severity: 1
Arrival Order: P4(1)
Patients added:
Serving patient P4 with severity: 1
Line is empty.
Window has closed, no new patients.
```

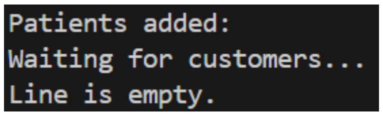
Figure 4: Error Case Fix Implementation Output

Appendix B: Output Functionality Screenshots



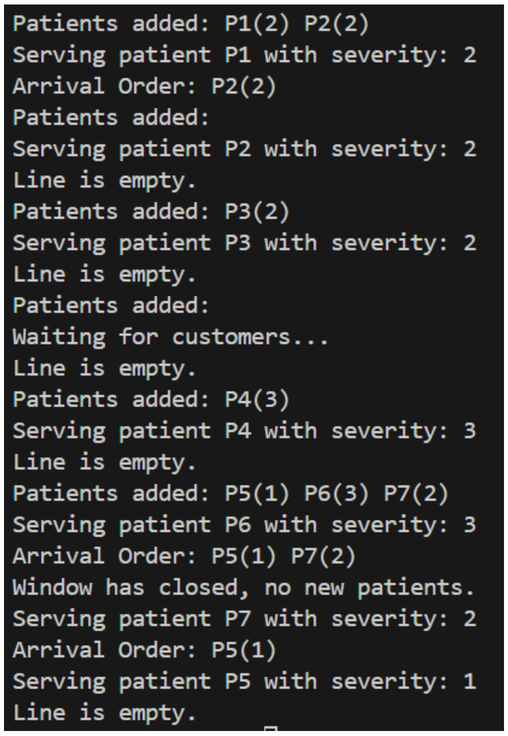
```
Patients added: P5(1) P6(1) P7(3)
Serving patient P7 with severity: 3
Arrival Order: P5(1) P6(1)
Patients added:
Serving patient P5 with severity: 1
Arrival Order: P6(1)
Window has closed, no new patients.
Serving patient P6 with severity: 1
Line is empty.
```

Figure 5: Severity Prioritization



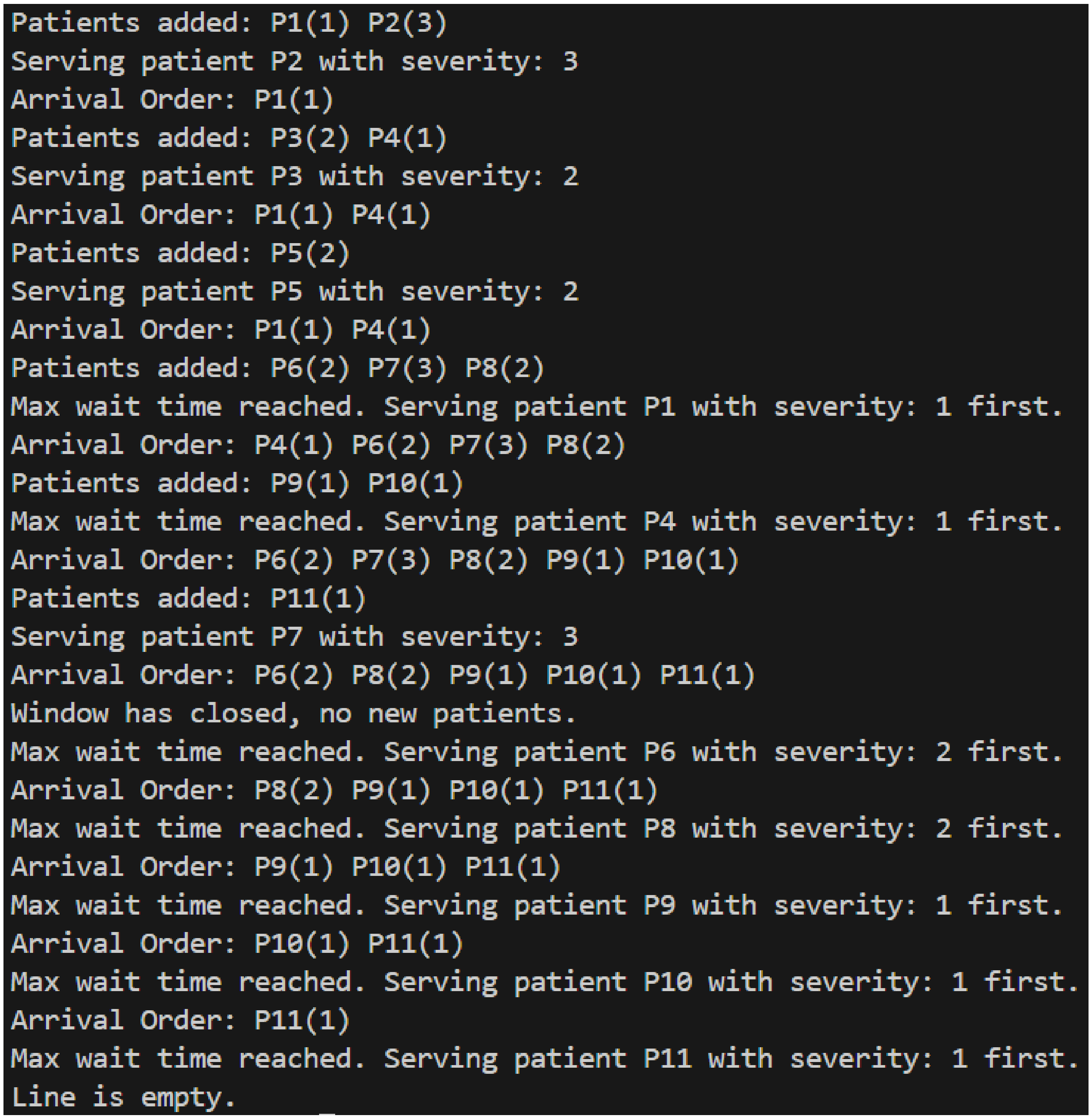
```
Patients added:
Waiting for customers...
Line is empty.
```

Figure 6: No New Patients Added While No Patients in Line



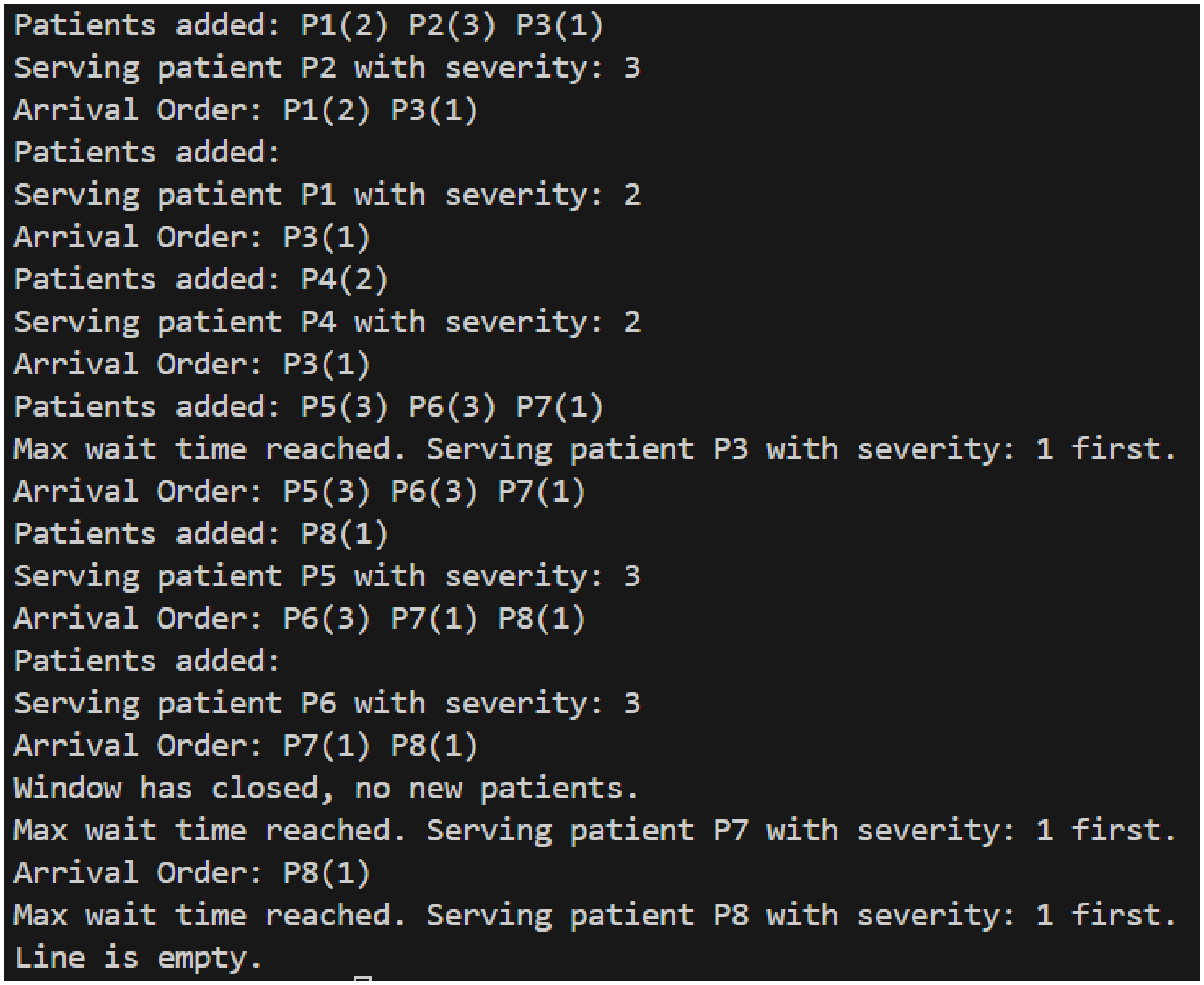
```
Patients added: P1(2) P2(2)
Serving patient P1 with severity: 2
Arrival Order: P2(2)
Patients added:
Serving patient P2 with severity: 2
Line is empty.
Patients added: P3(2)
Serving patient P3 with severity: 2
Line is empty.
Patients added:
Waiting for customers...
Line is empty.
Patients added: P4(3)
Serving patient P4 with severity: 3
Line is empty.
Patients added: P5(1) P6(3) P7(2)
Serving patient P6 with severity: 3
Arrival Order: P5(1) P7(2)
Window has closed, no new patients.
Serving patient P7 with severity: 2
Arrival Order: P5(1)
Serving patient P5 with severity: 1
Line is empty.
```

Figure 7: Wait Time Threshold Not Met



```
Patients added: P1(1) P2(3)
Serving patient P2 with severity: 3
Arrival Order: P1(1)
Patients added: P3(2) P4(1)
Serving patient P3 with severity: 2
Arrival Order: P1(1) P4(1)
Patients added: P5(2)
Serving patient P5 with severity: 2
Arrival Order: P1(1) P4(1)
Patients added: P6(2) P7(3) P8(2)
Max wait time reached. Serving patient P1 with severity: 1 first.
Arrival Order: P4(1) P6(2) P7(3) P8(2)
Patients added: P9(1) P10(1)
Max wait time reached. Serving patient P4 with severity: 1 first.
Arrival Order: P6(2) P7(3) P8(2) P9(1) P10(1)
Patients added: P11(1)
Serving patient P7 with severity: 3
Arrival Order: P6(2) P8(2) P9(1) P10(1) P11(1)
Window has closed, no new patients.
Max wait time reached. Serving patient P6 with severity: 2 first.
Arrival Order: P8(2) P9(1) P10(1) P11(1)
Max wait time reached. Serving patient P8 with severity: 2 first.
Arrival Order: P9(1) P10(1) P11(1)
Max wait time reached. Serving patient P9 with severity: 1 first.
Arrival Order: P10(1) P11(1)
Max wait time reached. Serving patient P10 with severity: 1 first.
Arrival Order: P11(1)
Max wait time reached. Serving patient P11 with severity: 1 first.
Line is empty.
```

Figure 8: Remaining Patients all past Threshold, Order Prioritization



```
Patients added: P1(2) P2(3) P3(1)
Serving patient P2 with severity: 3
Arrival Order: P1(2) P3(1)
Patients added:
Serving patient P1 with severity: 2
Arrival Order: P3(1)
Patients added: P4(2)
Serving patient P4 with severity: 2
Arrival Order: P3(1)
Patients added: P5(3) P6(3) P7(1)
Max wait time reached. Serving patient P3 with severity: 1 first.
Arrival Order: P5(3) P6(3) P7(1)
Patients added: P8(1)
Serving patient P5 with severity: 3
Arrival Order: P6(3) P7(1) P8(1)
Patients added:
Serving patient P6 with severity: 3
Arrival Order: P7(1) P8(1)
Window has closed, no new patients.
Max wait time reached. Serving patient P7 with severity: 1 first.
Arrival Order: P8(1)
Max wait time reached. Serving patient P8 with severity: 1 first.
Line is empty.
```

Figure 9: Typical Output Window

Appendix C: C++ Implementation

```
#include <iostream>
#include <queue>
#include <chrono>
#include <vector>
#include <thread>
using namespace std;

class PriorityQueue {
private :
// priority queue for severity, timer vector for wait time
// order vector for preserving patient arrival order
// patientID vector for easier identification in output
priority_queue<int> pq;
vector<chrono::time_point<std::chrono::system_clock>> timers;
vector<int> order;
vector<int> patientID;
int lastPatient = 0;
public :

bool isEmpty(){
    return pq.empty();
}

int maxTime(chrono::time_point<std::chrono::system_clock> str){
    // calculate duration from start point
    chrono::time_point<std::chrono::system_clock> stp;
    stp = chrono::system_clock::now();
    chrono::duration<double> dur = stp - str;
    double wT = dur.count();
    // if duration is past threshold, find the position in timers
    // where duration is past threshold and return index
    // this index will match with order vector
    if(wT > 3){
            for(int i=0; i<timers.size(); i++){
            if(timers[i] == str){
                return i;
            }
        }
        }
    return -1;
}

void nPatients(int code){
    // random number (0-3) of new patients added
    // form severity array with n new patients
int numPatients = (rand()%4);
        int arr[numPatients];
        // code 0 indicates window is open: add new patients
        if(code == 0){
        cout << "Patients added: ";
        for(int i=0; i<numPatients; i++){
        // random severity 1-3 for each new patient
        arr[i] = (rand()%3+1);
        // record start time in timer vector + order that patients arrive
        timers.push_back(std::chrono::system_clock::now());
        order.push_back(arr[i]);
        patientID.push_back(lastPatient+i+1);
        cout << "P" << lastPatient+i+1 << "(" << arr[i] << ") ";
        }
        if(!patientID.empty()){
            lastPatient = patientID.back();
        }
        cout << endl;
        }
        int proceed = -1;
        // if patients need to be added or line is not empty call maxTime
        if(numPatients != 0 || !pq.empty()){
            // proceed will be -1 if wait time is not reached or 0
            // only the first timer needs to be checked as timers stores
            // time points in order of arrival, if first timer is not
            // past wait time threshold the others are not either
            proceed = proceed = maxTime(timers[0]);
        }

        // proceed -1 indicates wait time not reached, add patients > serve
        if(proceed == -1){
            // if window is still open, add patients to pq
            if(code == 0){
            for (int i=0; i<numPatients; i++) {
                pq.push(arr[i]);
            }}
            servePatient(proceed);
        }
        // proceed 0 indicates wait time reached, serve > add new patients
        // note that proceed accepts index at which wait time passes
        // threshold, as timers vector is in order of arrival, the first
        // timer duration will always be the longest
        else {
            servePatient(proceed);
            if(code == 0){
            for (int i=0; i<numPatients; i++) {
                pq.push(arr[i]);}
            }
        }
}

int matchOrderPrio(){
    // pq generally does not preserve order, arrival order is only
    // preserved for two of the same element; if severity 3 is added while
    // 3 is already part of the pq, it will be added behind the existing
    // as such, the first match returns the correct index
for(int i=0; i<order.size(); i++){
    if(order[i] == pq.top()){
        return i;
    }
}
```

```
return 0;
}

int matchPrioOrder(int index){
    // similarly to matching the order vector element with the severity
    // being served in pq, the same process is done by incrementing
    // pos until the first match of severity the patient at given index
    priority_queue<int> copy = pq;
    int pos = 0;
while(!copy.empty()){
    if(copy.top() == order.at(index)){
        return pos;
    }
    copy.pop();
    pos += 1;
}
return 0;
}

void servePatient(int code){
    // base case for no patients to be served
    if(pq.empty()){
        cout << "Waiting for customers..." << endl;
        return;
    }
    // for usual case -1, serve patients off of severity
    // pq automatically sorts in decreasing order
    if(code == -1){
        int ind = matchOrderPrio();
        cout << "Serving patient P" <<
        patientID[ind] << " with severity: " << pq.top() << endl;
        // erase patient served from order and timers vectors
        // this is done by calling the match function
        order.erase(order.begin()+ind);
        timers.erase(timers.begin()+ind);
        patientID.erase(patientID.begin()+ind);
        pq.pop();
    }
    // if wait time has been reached, pq is modified by creating a new pq
    // in increasing order
    else {
        cout << "Max wait time reached. Serving patient P" <<
        patientID[code] << " with severity: ";
        cout << order[code] << " first." <<endl;
        priority_queue<int> copy = pq;
        priority_queue<int, vector<int>, greater<int>> temp;
        while(!copy.empty()){
        temp.push(copy.top());
        copy.pop();
    }
    // when wait time reached, code is assigned to index of the patient
    // for which wait time has been reached in the order and timers vectors
    // this will be 0 as the first arriving customer has waited the longest
    int pos = matchPrioOrder(code);
    // pop reverse pq up until and including patient being served
    for(int i=0;i<pq.size() - pos;i++){
        temp.pop();
    }
    // pop pq up until and including patient being served
    for(int i=0;i<pos+1;i++){
        pq.pop();
    }
    // add remaining reverse pq elements to pq
    while(!temp.empty()){
        pq.push(temp.top());
        temp.pop();
    }
    // erase patient from order and timers vectors
        order.erase(order.begin()+code);
        timers.erase(timers.begin()+code);
        patientID.erase(patientID.begin()+code);
    }
}

void dispLine(){
    // copy pq and display all elements in order
    priority_queue<int> copy = pq;
    if(copy.empty()){
        cout << "Line is empty." << endl;
    }
    else{
    cout << "Severity Order: ";
    while (!copy.empty()) {
        cout << copy.top() << " ";
        copy.pop();
        }
        cout << endl;}
}

void dispOrder(){
    // display patient arrival order
    if(order.empty()){
        cout << "Line is empty." << endl;
    }
    else{
        cout << "Arrival Order: ";
    for(int i = 0; i < patientID.size(); i++){
        cout << "P" << patientID[i] << "(" << order[i] << ") ";}
    cout << endl;
    }


}

};

// driver code
int main() {
    // initizalize window open time point and patientLine, random seed
    chrono::time_point<std::chrono::system_clock> open = chrono::system_clock::now();
    double close;
    PriorityQueue patientLine;
    srand((unsigned)time(0));
    while (close < 5){
        patientLine.nPatients(0);
        //patientLine.dispLine();
        patientLine.dispOrder();

    // run window for 5 seconds before closing window to new patients
    // update duration since open and add second delay
    chrono::time_point<std::chrono::system_clock> curtime = chrono::system_clock::now();
    chrono::duration<double> dur = curtime - open;
    close = dur.count();
    this_thread::sleep_for(chrono::seconds(1));
    }
    // once window has closed, call nPatients with code 1, indicating
    // that new patients should not be added to the pq
    cout << "Window has closed, no new patients." << endl;
    while(!patientLine.isEmpty()) {
        patientLine.nPatients(1);
        //patientLine.dispLine();
        patientLine.dispOrder();
        this_thread::sleep_for(chrono::seconds(1));
    }

    return 0;
}
```