# Print Job Scheduling with Priority Queue Using Max-Heap

Gormery K. Wanjiru

October 30, 2024

## Print Job Scheduling System

This program schedules print jobs by priority, using a max-heap to ensure higher-priority jobs are processed first. Users can add jobs, view the job with the highest priority, update priorities, and process jobs.

## Components of the System

1. **Priority Queue with Max-Heap**: - Jobs added to the max-heap are ordered by priority, so the job with the highest priority is at the top. - Includes methods for adding jobs, processing the highest-priority job, and updating job priorities.

2. **Adding Jobs**: - Users can add jobs with a name and priority, and the system shows the job with the highest priority after each addition.

3. **Updating Job Priority**: - Users can update any job's priority, triggering a reorganization of the max-heap to maintain order.

## How to Use It

To use the print job scheduling program, follow these steps:

1. **Compile and Run**
   - First, compile the program using a C++ compiler (e.g., `g++ main.cpp -o main`).
   - Run the executable (e.g., `./main`) to start the program.
2. **User-Friendly Interface**
   - Once the program is running, a menu interface will appear, guiding you through the options.
   - The menu provides choices to add jobs, update priorities, process the highest-priority job, and view the job queue.
3. **Navigating the Menu**
   - The program will prompt you to input your choice for each operation. Simply follow the prompts to:
     - Add new print jobs with a name and priority.
     - Update priorities for existing jobs.
     - Process and remove the highest-priority job.
     - View the full job queue and see which job has the highest priority.
   - After each operation, the program will display the current state of the queue, reflecting any changes made.

The interface ensures a smooth experience by handling input validation and displaying relevant error messages if needed.

# Code Structure

## Main Priority Queue and Print Job Structure

```
// Priority Queue and Print Job Structure
#include <iostream>
#include <vector>
#include <string>
#include <unordered_map>

struct PrintJob {
    std::string name;
    int priority;
    PrintJob(const std::string& name, int priority) : name(name), priority(
        priority) {}
};

class PriorityQueue {
private:
    std::vector<PrintJob> heap;
    std::unordered_map<std::string, int> jobMap; // Maps names for fast access
    bool verbose;

    void heapifyUp(int index);
    void heapifyDown(int index);
public:
    PriorityQueue(bool verbose = true) : verbose(verbose) {}
    void insertJob(const std::string& name, int priority);
    void processJob();
    void updatePriority(const std::string& name, int newPriority);
    void displayHighestPriorityJob();
    void displayQueue();
};
```

# Testing results

Following the example scenario, we tested each feature in sequence. Below are the expected and actual outputs for each step, with the result noted as pass or fail.

1. **Insert "Thesis" with Priority 4**
    - **Expected:** `"Next job: Thesis (Priority: 4)"`
    - **Actual:** `"Next job: Thesis (Priority: 4)"`
    - **Result:** PASS
2. **Insert "Project" with Priority 5**
    - **Expected:** `"Next job: Project (Priority: 5)"`
    - **Actual:** `"Next job: Project (Priority: 5)"`
    - **Result:** PASS
3. **Insert "Report" with Priority 3**
    - **Expected:** `"Next job: Project (Priority: 5)"`
    - **Actual:** `"Next job: Project (Priority: 5)"`
    - **Result:** PASS
4. **Insert "Assignment" with Priority 2**
    - **Expected:** `"Next job: Project (Priority: 5)"`

- **Actual:** "Next job: Project (Priority: 5)"
- **Result:** PASS

5. **Display Job with Highest Priority**
   - **Expected:** "Next job: Project (Priority: 5)"
   - **Actual:** "Next job: Project (Priority: 5)"
   - **Result:** PASS
6. **Process Job with Highest Priority (Project)**
   - **Expected:** "Processing: Project (Priority: 5)"
   - **Actual:** "Processing: Project (Priority: 5)"
   - **Result:** PASS
7. **Insert "Invoice" with Priority 6**
   - **Expected:** "Next job: Invoice (Priority: 6)"
   - **Actual:** "Next job: Invoice (Priority: 6)"
   - **Result:** PASS
8. **Update Priority of "Thesis" to 7**
   - **Expected:** "Updated Thesis to priority 7"
   - **Actual:** "Updated Thesis to priority 7. Queue contains: [Thesis (Priority: 7)] [Invoice (Priority: 6)] [Report (Priority: 3)] [Assignment (Priority: 2)]"
   - **Result:** PASS

## Testing Code for Automated Sequence

```
#define TESTING
#include <iostream>
#include <functional>
#include <sstream>
#include <string>
#include "main.cpp" // Include the main.cpp file containing PriorityQueue

// Helper function to capture output of specific actions
std::string captureOutput(std::function<void()> func) {
    std::ostringstream buffer;
    std::streambuf* old = std::cout.rdbuf(buffer.rdbuf());
    func();
    std::cout.rdbuf(old);
    return buffer.str();
}

// Helper function to print test results
void printTestResult(const std::string& testDescription, const std::string&
     actualOutput, const std::string& expectedOutput, bool condition) {
    std::cout << "Test: " << testDescription << "\n";
    std::cout << "Expected: " << expectedOutput << "\n";
    std::cout << "Actual: " << actualOutput << "\n";
    std::cout << (condition ? "Result: PASS" : "Result: FAIL") << "\n\n";
}

// Function to perform the example test sequence
void runExampleTest() {
    PriorityQueue queue;
    bool condition;
    std::string output;
```

```
        // Step 1: Insert "Thesis" with Priority 4
        output = captureOutput([&]() { queue.insertJob("Thesis", 4); });
        condition = (output.find("Next job: Thesis (Priority: 4)") != std::
            string::npos);
        printTestResult("Insert 'Thesis' with priority 4", output, "Next job:
            Thesis (Priority: 4)", condition);

        // Step 2: Insert "Project" with Priority 5
        output = captureOutput([&]() { queue.insertJob("Project", 5); });
        condition = (output.find("Next job: Project (Priority: 5)") != std::
            string::npos);
        printTestResult("Insert 'Project' with priority 5", output, "Next job:
            Project (Priority: 5)", condition);

        // the other tests ...
    }

    int main() {
        runExampleTest();
        return 0;
    }
```

## Testing Results discussion

The testing sequence confirmed the system processes jobs by priority and manages priority updates correctly. Each function worked as expected.

## Conclusion

The priority queue works as intended, scheduling and processing print jobs based on priority. The max-heap ensures efficient management of jobs, and tests confirm that job insertion, priority updates, and processing operations behave as expected.

## References