Print Job Scheduling with Priority Queue Using Max-Heap

Gormery K. Wanjiru

October 29, 2024

1 Introduction

This project implements a print job scheduling system using a priority queue based on a max-heap in C++. The system allows users to add print jobs with different priorities and processes them in descending order of priority. The project demonstrates the application of a max-heap for task prioritization and includes structured test outputs to verify functionality.

2 System Design

This system following the assignment is broken down into three core tasks:

1. **Task 1: Priority Queue Implementation** - Develop a max-heap-based priority queue to manage and process print jobs by priority.
2. **Task 2: Job Addition and Display** - Create a user interface for job addition that displays the job with the highest priority after each addition.
3. **Task 3: Priority Update** - Enable priority updates for existing jobs, and reordering them within the max-heap as needed.

3 Algorithm and Implementation Details

3.1 Max-Heap Priority Queue

The max-heap data structure ensures that higher-priority print jobs (higher integer values) are processed before lower-priority ones. The program includes methods for inserting jobs, updating job priority, and processing jobs based on priority, with some internal functions to maintain heap properties after every operation.

```
// Priority Queue Implementation
#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; // Map for quick job lookup
    bool verbose; // Flag for output control in testing

    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();
};
```

3.2 Testing Sequence

The testing sequence follows the specified scenario we got in the assignment. it also just captures the output at each step to verify correct functionality.

```
Step 1: Insert "Thesis" with Priority 4
Expected: Next job: Thesis (Priority: 4)
Actual: Next job: Thesis (Priority: 4)
Result: PASS

Step 2: Insert "Project" with Priority 5
Expected: Next job: Project (Priority: 5)
Actual: Next job: Project (Priority: 5)
Result: PASS

Step 3: Insert "Report" with Priority 3
Expected: Next job: Project (Priority: 5)
Actual: Next job: Project (Priority: 5)
Result: PASS

Step 4: Insert "Assignment" with Priority 2
Expected: Next job: Project (Priority: 5)
Actual: Next job: Project (Priority: 5)
Result: PASS

Step 5: Display Highest Priority Job
Expected: Next job: Project (Priority: 5)
Actual: Next job: Project (Priority: 5)
Result: PASS

Step 6: Process Job with Highest Priority
Expected: Processing: Project (Priority: 5)
Actual: Processing: Project (Priority: 5)
Result: PASS

Step 7: Insert "Invoice" with Priority 6
Expected: Next job: Invoice (Priority: 6)
Actual: Next job: Invoice (Priority: 6)
Result: PASS

Step 8: Update "Thesis" Priority to 7
Expected: Updated Thesis to priority 7
Actual: Updated Thesis to priority 7
Result: PASS
```

4 Testing Code

Below is the code for testing for the output above, which automates each step and validates expected outcomes. The function `printTestResult` compares captured output to expected results and displays pass/fail status.

```cpp
// 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;

    // Test Steps
    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);

    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);

    // Additional steps continue...
}

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