CS 2413 – Data Structures – Fall 2024 – Project Three
Due 11:59 PM, October 18, 2024

Description: This project's objective is to implement a novel templated queue data structure called NovelQueue in C++. This queue will simulate a CPU scheduling system, managing a collection of CPU jobs and their execution characteristics.

In real-world computing environments, operating systems use CPU scheduling algorithms to determine the order in which jobs (processes) are executed. This project simulates this concept by allowing students to manipulate a queue of CPU jobs with various attributes, including priority, job type, CPU time consumed, and memory consumption. By doing so, students will gain a practical understanding of how efficient CPU scheduling is critical for optimizing system performance and managing system resources.

The goal of this project is to provide students with hands-on experience in:

- Implementing and working with linked list-based data structures.
- Using templates in C++ to handle generic data.
- Applying sorting and searching algorithms within the context of job scheduling.
- Gaining a deeper understanding of how queue management impacts CPU scheduling efficiency.
- Practicing memory management by dynamically allocating and deallocating job records.

By designing and implementing the NovelQueue, students will explore how scheduling decisions can be made dynamically in response to job attributes, such as priority and resource consumption. This project mirrors key concepts in operating system design, particularly those related to process management, reinforcing both data structure knowledge and systems programming principles.

Data Structure to be Built

1. NovelQueue: A linked list-based queue that holds CPU job records with an additional array of pointers to queue nodes. The array is always kept in sorted order based on job IDs, allowing efficient searching and reordering.
2. Job Record Template: This will be a C++ class that defines a CPU job with the following fields:
    - Job ID: A unique identifier for each job.
    - Priority: Priority level of the job.
    - Job Type: An integer representing the type of job.
    - CPU Time Consumed: The total CPU time used by the job so far.
    - Total Memory Consumed Thus Far: The memory consumed by the job up to this point.

Data Structure Operations

Implement the following operations for NovelQueue:

1. Add (Enqueue) – 'A':
   - Format: A job_id priority job_type cpu_time_consumed memory_consumed
   - Description: Adds a new job to the queue. After insertion, update the sorted array of pointers to ensure it reflects the new job's position based on its job ID.
2. Remove (Dequeue) – 'R':
   - Format: R
   - Description: Removes the job at the front of the queue. Update the sorted array to reflect the removal and maintain its sorted order.
3. Modify – 'M':
   - Format: M job_id new_priority new_job_type new_cpu_time_consumed new_memory_consumed
   - Description: Searches for the job with the specified job_id, dequeues it, modifies its attributes as specified, and enqueues it back.
4. Change Job Values – 'C':
   - Format: C job_id field_index new_value
   - Description: Changes a specific field of the job with the given job_id. The field_index corresponds to different fields such as priority, job type, CPU time consumed, or memory consumed.
5. Promote:
   - Format: P job_id positions
   - Description: Promotes the job with job_id towards the front by the specified number of positions. If the number exceeds its current position, move it to the front.
6. Reorder Based on Attributes – 'O':
   - Format: O attribute_index
   - Description: Reorders the queue based on the specified attribute (e.g., CPU time consumed, priority). This operation returns a new instance of the reordered NovelQueue.
7. Display – 'D':
   - Format: D
   - Description: Displays all elements of the queue, including all fields of the job record.
8. Count – 'N':
   - Format: N
   - Description: Returns the number of elements in the queue.
9. List Jobs – 'L':
   - Format: L
   - Description: Iterates through the array of pointers in NovelQueue (sorted by job IDs) and prints each job's information using the CPUJob display method. Also, shows the position of each job in the queue.

Command	Format	Description
A	`A job_id priority job_type cpu_time_consumed memory_consumed`	Adds a job to the queue.
R	`R`	Removes the front job.
M	`M job_id new_priority new_job_type new_cpu_time_consumed new_memory_consumed`	Modifies a job's attributes.
C	`C job_id field_index new_value`	Changes a specific field of a job.
P	`P job_id positions`	Promotes a job within the queue.
O	`O attribute_index`	Reorders the queue based on a specified attribute.
D	`D`	Displays all elements in the queue.
N	`N`	Returns the number of elements in the queue.
L	`L`	Lists jobs in sorted order of job IDs and shows their positions.

Data Structure Constraints

1. pointers must always be sorted based on job IDs whenever jobs are added, removed, or modified.
2. **Operations like promotion and job attribute changes should utilize the sorted array to enhance efficiency. Use binary search on the sorted array to search jobs and ensure optimal performance.**
3. Thoroughly test all the operations to ensure the correct functionality of the NovelQueue.

Your Project Implementation: As part of this project, three classes are CPUJob Class, Queue class (linked list), and the NovelQueue. Here, I will give the required fields for each of the classes. You have the freedom to add more fields and methods.

```
class CPUJob {
public:
    int job_id;            // Unique identifier for the job
    int priority;          // Priority level of the job (1-10)
    int job_type;          // Job type (1-10)
    int cpu_time_consumed; // Total CPU time consumed by the job
    int memory_consumed;   // Total memory consumed thus far
};
```

```
template <class DT>
class Queue {
public:
    DT* JobPointer;           // Pointer to a job (e.g., CPUJob)
    Queue<DT>* next;          // Pointer to the next node in the queue
};
```

```
template <class DT>
class NovelQueue {
public:
    Queue<DT>* front;         // Pointer to the front of the queue
    Queue<DT>** NodePtrs;     // Array of pointers to Queue nodes
    int size;                 // Number of elements in the queue)
};
```

Programming Objectives:

1. All code must be in C++.You will create a class given below with appropriate fields.
2. The class will have several methods whose prototypes are provided to you.
3. All input will be read via redirected input. That is, you will not open a file inside the program.
4. The class structure is shown below (you are responsible for fixing any syntax errors).
5. The structure for your main program is also provided. You will use that for your project.

The Main Program

Your program must execute the main program. We may change the main program but will ensure that we only call the methods that you have been asked to create.

```
#include <iostream>
using namespace std;

int main() {
    int n;  // Number of commands
    cin >> n;  // Read the number of commands

    // Instantiate a NovelQueue for CPUJob pointers
    NovelQueue<CPUJob*>* myNovelQueue = new NovelQueue<CPUJob*>();

    char command;  // Variable to store the command type

    // Variables for job attributes
    int job_id, priority, job_type, cpu_time_consumed, memory_consumed;

    // Variables for modifying a job
    int new_priority, new_job_type, new_cpu_time_consumed;
    int new_memory_consumed;
    int field_index, new_value;
    // Variable for the number of positions in the 'Promote' command
    int positions;
    int attribute_index;  // Variable for the 'Reorder' command

/************** Read each command Process ***************//
```

```
    for (int i = 0; i < n; ++i) {
        cin >> command;  // Read the command type

        switch (command) {
            case 'A': {  // Add (Enqueue)
                cin >> job_id >> priority >> job_type;
                cin >> cpu_time_consumed >> memory_consumed;
                CPUJob* newJob = new CPUJob(job_id, priority, job_type,
                                    cpu_time_consumed, memory_consumed);
                (*myNovelQueue).enqueue(newJob);
                break;
            }
            case 'R': {  // Remove (Dequeue)
                CPUJob* removedJob = (*myNovelQueue).dequeue();
                if (removedJob) {
                    cout << "Dequeued Job: ";
                    (*removedJob).display();
                    delete removedJob;  // Clean up memory after use
                }
                break;
            }
            case 'M': {  // Modify
                cin >> job_id >> new_priority >> new_job_type;
                cin >> new_cpu_time_consumed >> new_memory_consumed;
                (*myNovelQueue).modify(job_id, new_priority, new_job_type,
                             new_cpu_time_consumed, new_memory_consumed);
                break;
            }
            case 'C': {  // Change Job Values
                cin >> job_id >> field_index >> new_value;
                (*myNovelQueue).change(job_id, field_index, new_value);
                break;
            }
            case 'P': {  // Promote
                cin >> job_id >> positions;
                (*myNovelQueue).promote(job_id, positions);
                break;
            }
            case 'O': {  // Reorder
                cin >> attribute_index;
                NovelQueue<CPUJob*>* reorderedQueue =
                           (*myNovelQueue).reorder(attribute_index);
                cout << "Reordered Queue:" << endl;
                (*reorderedQueue).display();
                break;
            }
            case 'D': {  // Display
                (*myNovelQueue).display();
                break;
            }
            case 'N': {  // Count
                cout << "Number of elements in the queue: " <<
                       (*myNovelQueue).count() << endl;
                break;
            }
            case 'L': {  // List Jobs
```

```
                (*myNovelQueue).listJobs();
                break;
            }
            default:
                cout << "Invalid command!" << endl;
        }
    }

    delete myNovelQueue;  // Clean up the NovelQueue after all operations
    return 0;
}
```

Sample Input

```
50
A 101 5 2 150 2000
A 102 3 1 100 1024
A 103 7 4 250 4096
A 104 2 3 300 2048
A 105 6 5 200 512
A 106 4 2 175 3072
A 107 8 3 400 1024
A 108 1 5 50 2048
A 109 5 1 225 8192
A 110 9 4 500 16384
M 105 8 5 220 1024
C 102 1 9
P 108 2
R
O 1
D
A 111 7 2 350 1024
A 112 3 4 180 4096
C 111 3 450
L
N
M 103 6 4 275 5120
A 113 2 3 100 256
P 107 5
C 108 2 7
O 4
D
A 114 4 1 125 2048
R
L
N
P 110 1
P 109 3
P 104 2
O 2
D
A 115 2 5 300 2048
```

```
A 116 5 2 175 1024
P 115 4
O 3
D
C 115 1 7
A 117 3 3 200 512
P 116 2
O 4
L
R
P 112 1
A 118 6 4 400 3072
O 1
D
N
```

Redirected Input: Redirected input provides you a way to send a file to the standard input of a program without typing it using the keyboard. To use redirected input in Visual Studio environment, follow these steps: After you have opened or created a new project, on the menu go to project, project properties, expand configuration properties until you see Debugging, on the right you will see a set of options, and in the command arguments type "< **input filename**". The < sign is for redirected input and the **input filename** is the name of the input file (including the path if not in the working directory).

Constraints

1. In this project, the only header you will use is #include <iostream> and #include <string> and using namespace std.
2. None of the projects are group projects. Consulting with other members of this class our seeking coding solutions from other sources including the web on programming projects is strictly not allowed and plagiarism charges will be imposed on students who do not follow this.

Project Submission Requirements:

1. **Code Development (75%):** Implement the provided classes and the required methods Your implementation must be fully compatible with the main program provided. Your code must run successfully with the main program and the corresponding input, demonstrating the correct functionality of all classes and its methods.

    - **LLM/AI Tool Usage:** You can use Large Language Models (LLMs) or AI tools, such as GitHub Copilot, to assist in writing and refining your classes and their methods. If you did not use LLM or AI tools to write your project, you still have to show for 2. below how you would have used it to find a solution to the project. **You need to make sure that you use the class structure provided to you as the basis, and failure to do that will result in zero points for this project.**

2. **LLM and GitHub Copilot Usage Documentation (15%):** If you choose to use LLM tools or GitHub Copilot, you must document your usage. This documentation (in PDF Format) should include:

   - **Prompts and Suggestions:** Provide the specific prompts or suggestions you used, such as "Generate a method for this method” or "How can I implement this method for a NovelQueue class?"
   - **Rationale:** Explain why you chose these prompts or suggestions and how they contributed to the development of your classes. For instance, you might describe how a particular suggestion helped you structure the program.
   - **Incremental Development:** Detail how you used the tools to build and refine your classes and methods incrementally. For example, you might start by generating a basic structure for the classes and then refine individual methods, ensuring each one integrates smoothly with the main program.

3. **Debugging and Testing Plan (10%):** Submit a comprehensive debugging and testing plan. This should include:

   - **Specific Tests:** Describe the tests you conducted on your class methods, such as checking that the compute method works correctly for various circuits.
   - **Issues and Resolutions:** Document any issues you encountered, such as handling zero values or optimizing for performance, and how you resolved them.
   - **Verification:** Explain how you verified that your classes work correctly with the provided main program. This could involve running a series of test cases the main program provides or creating additional test cases to ensure robustness.