DATA SRUCTURE PROJECT REPORT

Shortest Job First (SJF) CPU Scheduling using a Linked List–based Priority Queue

Submitted by

<JOSHUA SAMUEL>

Registration Number: 12408527

Course Code: CSM-228

Under the Guidance of

Mr. Aman Kumar

Delivery and Student Success

School of Computer Science and Engineering

ACKNOWLEDGEMENT

I wish to express my Sincere gratitude to **Mr. Aman Kumar** for his unwavering support and assistance throughout my project. I also extend my thanks to our friends for providing me with the opportunity to work on a project titled " Shortest Job First Scheduling " . Their guidance and insights were instrumental in the successful completion of this project.

<Joshua Samuel>
<12408527>

Contents

1.INTRODUCTION 5

2.**OBJECTIVES AND SCOPE OF THE PROJECT** 6

- Does not consider process arrival times (all processes arrive at time 0) 7
- Non-preemptive implementation only 7
- Console-based interface (no GUI) 7
- Designed for educational demonstration purposes 7

3.APPLICATION TOOLS 7

4.METHODOLOGY/ALGORITHM IMPLEMENTATION 8

5. SCREENSHOTS OF EXECUTION 12

6.SUMMARY 23

6.1 Project Achievements 23

This project successfully implemented the Shortest Job First CPU scheduling algorithm using a custom-built linked list data structure in Java. The key achievements include: 23

1. **Complete Implementation**: Fully functional SJF scheduler from scratch 23
2. **Data Structure Application**: Practical use of linked list for priority queue 23
3. **Algorithm Understanding**: Deep comprehension of CPU scheduling concepts 23
4. **Problem-Solving Skills**: Application of theoretical knowledge to practical problem 23

6.2 Learning Outcomes 23

Through this project, several important concepts were reinforced: 23

1. **Linked List Operations**: Insertion, deletion, traversal with sorting 23
2. **SJF Algorithm**: Understanding of optimal scheduling technique 23
3. **Time Calculations**: Waiting time and turnaround time computations 23
4. **System Simulation**: Modeling real-world OS behavior 23
5. **Java Programming**: Object-oriented design and implementation 23

6.3 Future Enhancements 23

The current implementation can be extended in several ways: 23

1. **Add Arrival Times**: Implement SJF with different arrival times 23
2. **Preemptive SJF**: Implement Shortest Remaining Time First (SRTF) 24
3. **GUI Interface**: Develop graphical visualization of scheduling 24
4. **Comparative Analysis**: Include other algorithms (Round Robin, Priority) 24
5. **Real-time Simulation**: Add process arrival during execution 24

6.4 Final Remarks 24

This project demonstrates the practical application of data structures in solving real-world operating system problems. The SJF algorithm, while theoretical in some aspects, provides valuable insights into efficient resource management. The linked list implementation offers a clear, understandable approach to priority queue management, making it an excellent educational tool for understanding both data structures and operating system concepts. 24

7.BIBLIOGRAPHY 24

7.ANNEXURE 25

1.INTRODUCTION

1.1 CPU Scheduling in Operating Systems

CPU scheduling is a fundamental function of modern operating systems that determines which process gets access to the CPU at any given time. Efficient scheduling algorithms are crucial for optimizing system performance, maximizing CPU utilization, and ensuring fair resource allocation among processes.

1.2 Shortest Job First (SJF) Algorithm

The Shortest Job First (SJF) scheduling algorithm is a non-preemptive scheduling technique that selects the process with the smallest execution time (burst time) for execution first. This algorithm is proven to be optimal for minimizing the average waiting time of processes, making it one of the most efficient scheduling algorithms in theory.

1.3 Project Overview

This project implements the SJF scheduling algorithm using a custom-built **Linked List-based Priority Queue** in Java. Instead of using Java's built-in collection classes, we design and implement our own linked list data structure to store and manage process entities. Each process is represented as a node containing Process ID, Process Name, and Execution Time as specified in the problem statement.

2.OBJECTIVES AND SCOPE OF THE PROJECT

2.1 Objectives

The main objectives of this project are:

The primary objectives of this project are:

1. To understand and implement the Shortest Job First CPU scheduling algorithm
2. To design and implement a linked list data structure from scratch
3. To create a priority queue that automatically sorts processes by execution time
4. To calculate and analyze waiting time and turnaround time for processes
5. To demonstrate the practical application of Data Structures in solving operating system problems
6. To develop a console-based simulation of CPU scheduling

2.2 Scope of the Project

The Scope of this project are:

- Implementation of non-preemptive SJF scheduling
- Custom linked list implementation without using Java Collections
- Process entity with ID, Name, and Execution Time
- Automatic sorting based on execution time
- Calculation of waiting time and turnaround time
- User-friendly console interface

Limitations:

- Does not consider process arrival times (all processes arrive at time 0)
- Non-preemptive implementation only
- Console-based interface (no GUI)
- Designed for educational demonstration purposes

3.APPLICATION TOOLS

• Java Programming Language:
The project is implemented using Java, a general-purpose, object-oriented programming language. Java is well-suited for console-based applications and provides strong support for data structures, memory management, and platform independence, making it ideal for implementing scheduling algorithms.

• Linked List Data Structure:
A linked list is used to store and manage process information dynamically. Each node in the linked list represents a process containing attributes such as process ID, burst time, waiting time, and turnaround time. The linked list allows easy insertion, deletion, and sorting of processes based on burst time.

• CPU Scheduling Algorithm (Shortest Job First):
The Shortest Job First (SJF) scheduling algorithm is used to determine the execution order of processes. The algorithm selects the process with the smallest burst time first, helping to minimize average waiting time and improve CPU efficiency.

• **Sorting Technique:**
Sorting logic is applied to the linked list to arrange processes in ascending order of burst time. This sorting ensures correct implementation of the SJF scheduling policy.

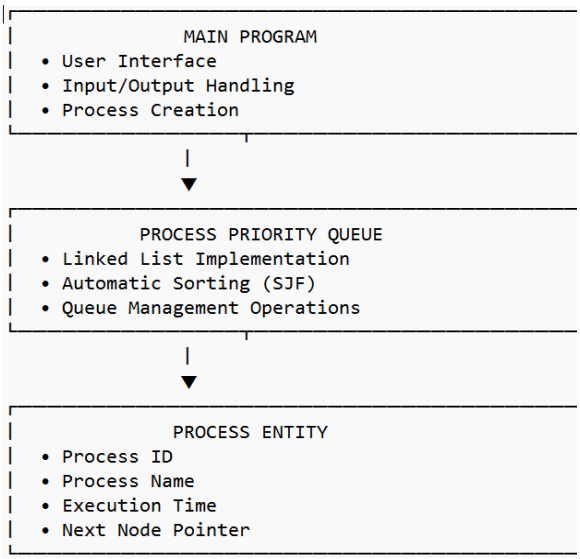
• **Console-Based User Interface:**
The program uses a text-based, menu-driven interface for user interaction. Users can enter process details and view calculated waiting time and turnaround time through standard input and output.

• **Java Development Kit (JDK):**
The Java Development Kit is used to compile and run the Java program. It provides the necessary tools such as the Java compiler (javac) and Java Virtual Machine (JVM).

• **Integrated Development Environment (IDE):**
Development and testing of the project are carried out using IDEs such as IntelliJ IDEA, Eclipse, or Visual Studio Code, which provide features like syntax highlighting, debugging, and error detection.

4.METHODOLOGY/ALGORITHM IMPLEMENTATION

4.1 System Architecture



4.2 Data Structure Design

4.2.1 PROCESS CLASS

```
class Process {
    int id;           // Unique identifier for process
    String name;      // Descriptive name of process
    int execTime;     // CPU burst time in milliseconds
    Process next;     // Reference to next process in linked list
}
```

4.2.2 ProcessQueue class

```
class ProcessQueue {
    Process head;     // Reference to first process in queue

    // Core operations:
    void addProcess(Process p)     // Add process with SJF sorting
    Process getNextProcess()       // Retrieve shortest process
    boolean isEmpty()              // Check if queue is empty
    void display()                 // Show all processes in queue
}
```

4.3 Algorithm Implementation

Algorithm: Shortest Job First (SJF) Scheduling Using Linked List

4.3.1 SJF Scheduling Algorithm

Input: Set of processes with execution times
Output: Execution order with waiting and turnaround times

Steps:

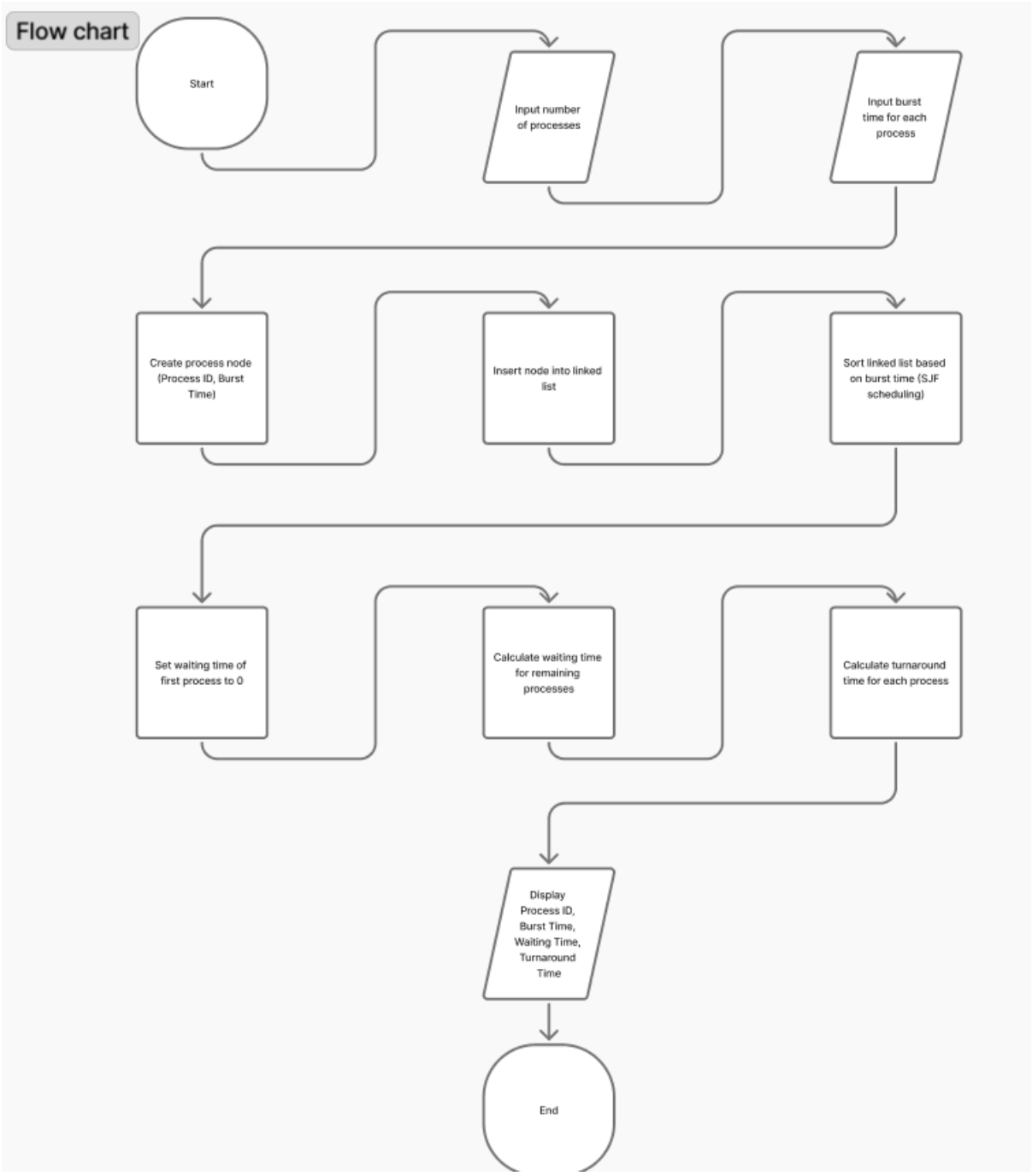
1. **Input Phase**: Collect process details from user
2. **Insertion Phase**: Add each process to queue (auto-sorts by SJF)
3. **Execution Phase**: Process each job in sorted order
4. **Calculation Phase**: Compute performance metrics
5. **Output Phase**: Display scheduling results

4.3.2 Mathematical Formulas

1. **Waiting Time Calculation**:
   - First process: $WT_1 = 0$
   - Subsequent processes: $WT_i = WT_{i-1} + BT_{i-1}$
     Where BT_{i-1} is burst time of previous process
2. **Turnaround Time Calculation**:
   - $TAT_i = WT_i + BT_i$
     Where BT_i is burst time of current process
3. **Average Waiting Time**:
   - $AWT = (\Sigma\ WT_i) / n$
4. **Average Turnaround Time**:
   - $ATT = (\Sigma\ TAT_i) / n$

→Where TAT=Turn Around Time && WT=Waiting time && BT=burst time.

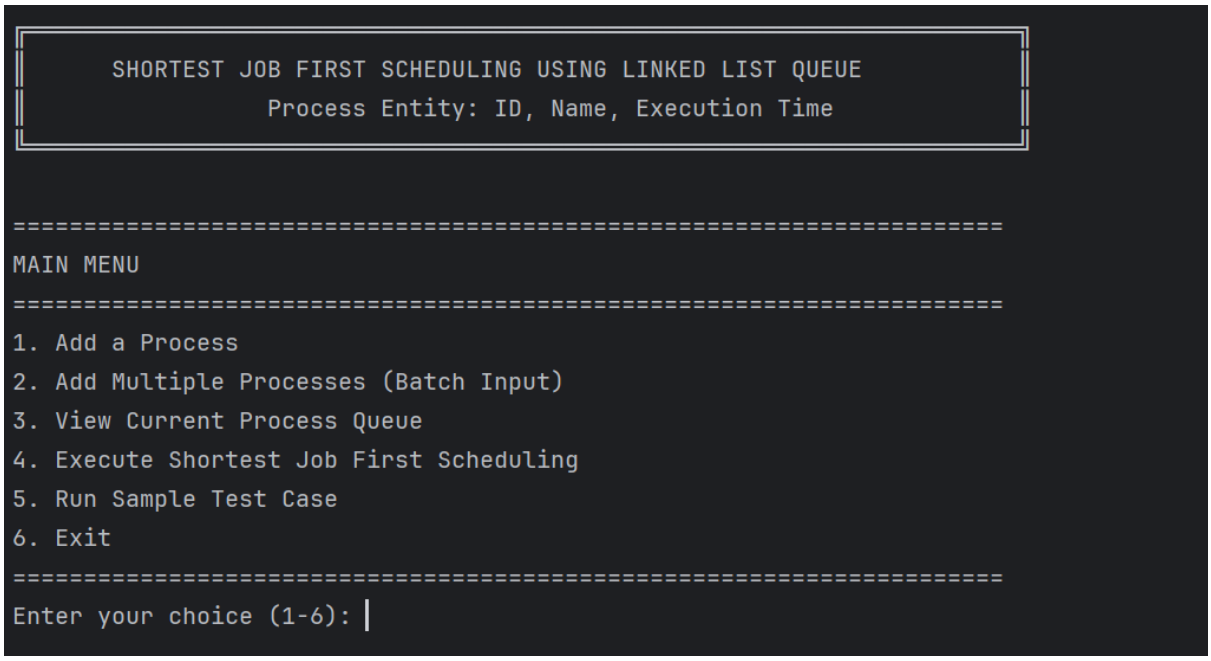
4.5 Flow of Execution


Flow chart
Start
Input number of processes
Input burst time for each process
Create process node (Process ID, Burst Time)
Insert node into linked list
Sort linked list based on burst time (SJF scheduling)
Set waiting time of first process to 0
Calculate waiting time for remaining processes
Calculate turnaround time for each process
Display Process ID, Burst Time, Waiting Time, Turnaround Time
End

5. SCREENSHOTS OF EXECUTION

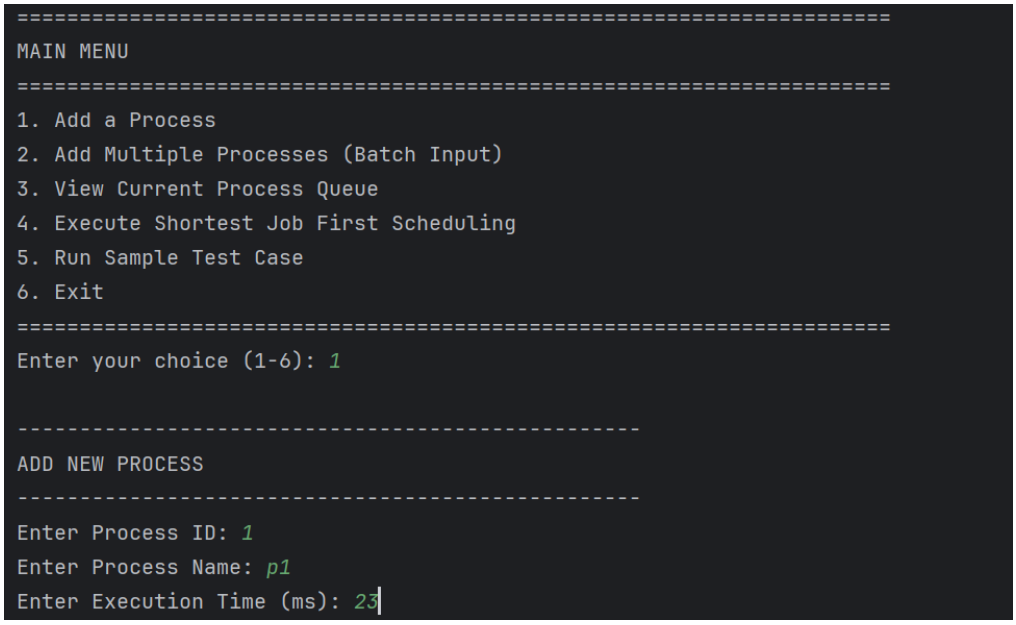
5.1 Screenshots of output and its function

Figure 1: Program Main Menu Interface



This screenshot shows the main menu of the SJF Scheduler program. The menu displays six options for user interaction: adding single processes, adding multiple processes in batch, viewing the current process queue, executing the SJF scheduling algorithm, running a sample test case, and exiting the program. This demonstrates the structured and user-friendly interface design.The numbers displaying from 1-6 has their own job and function which will be shown in the following screenshots.

Figure 2: Single Process Input Screen

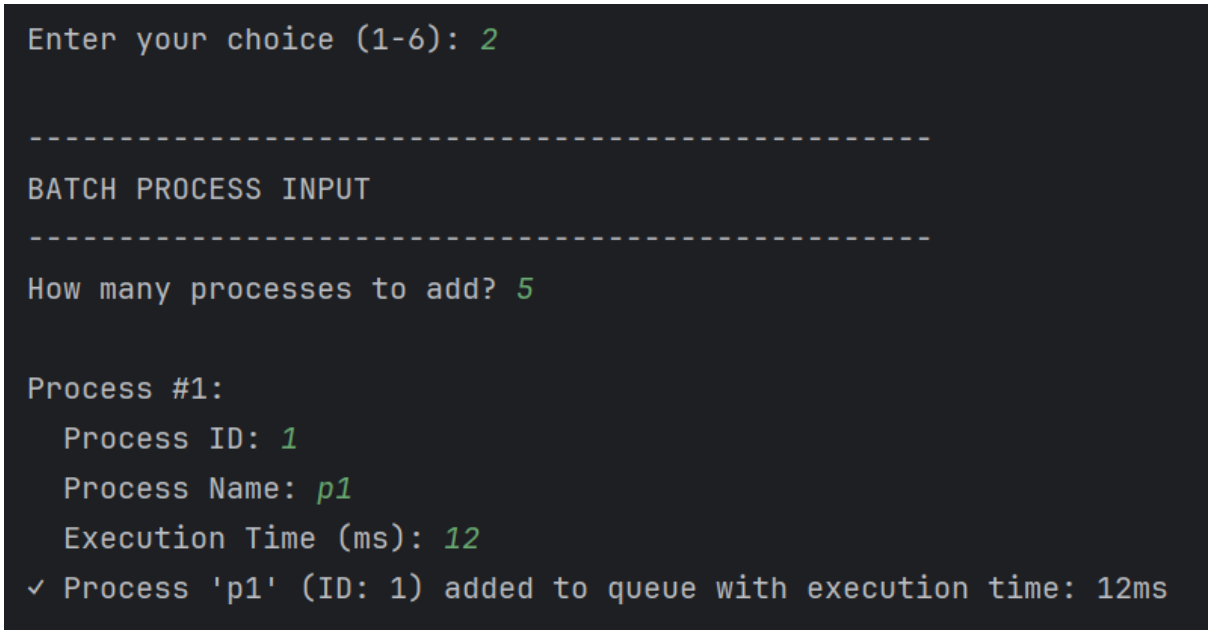


This screenshot captures the interface for adding a single process to the scheduler. The user has selected option 1 and is entering details for process p1 with execution time 23ms. The program prompts for Process ID, Process Name, and Execution Time, showing individual process entry functionality.

Figure 3: Process Addition Confirmation

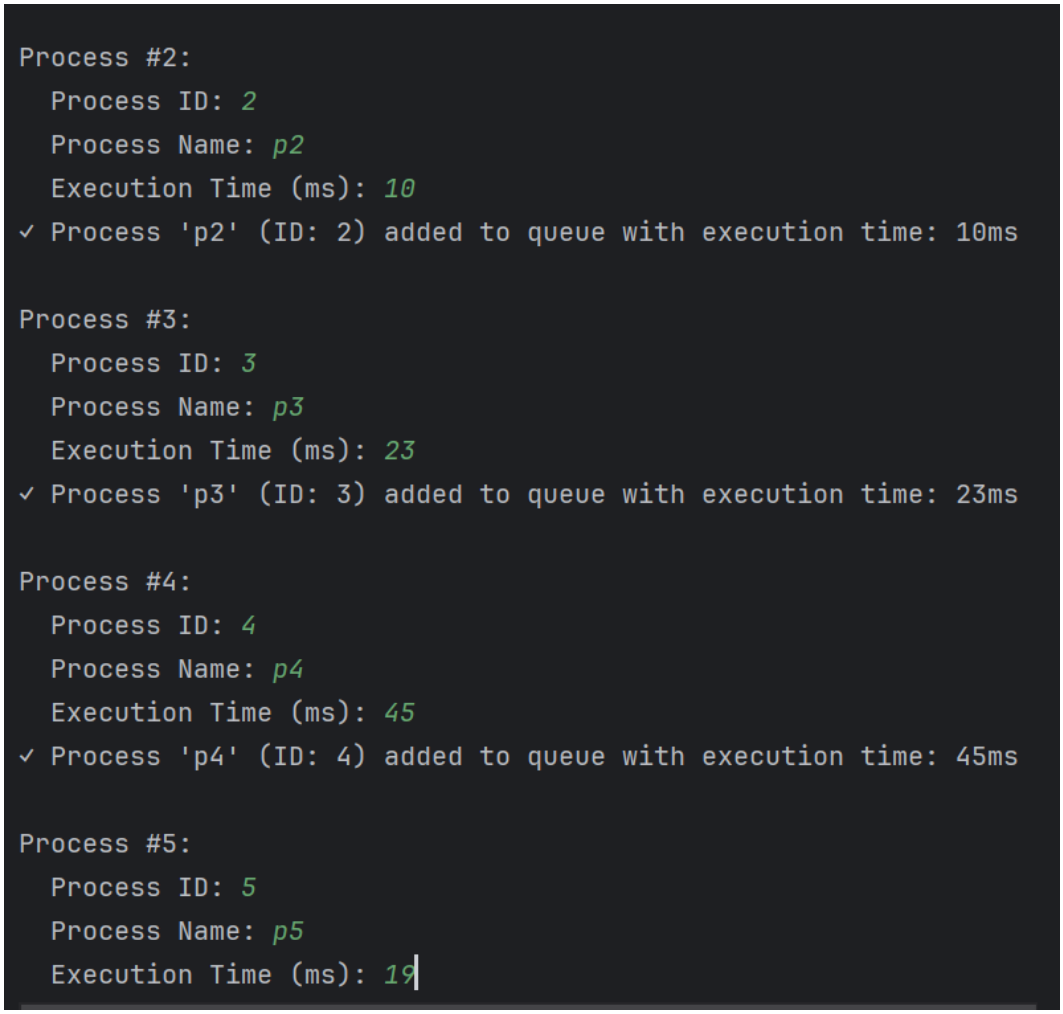
This screenshot shows the confirmation message after successfully adding process p1 to the queue. The message confirms that process p1 (ID: 1) with execution time 23ms has been added to the priority queue. The checkmark symbol provides clear visual feedback of successful operation.

Figure 4: Batch Process Input Interface



This screenshot demonstrates the batch input mode for adding multiple processes simultaneously. The user has selected option 2 and is entering 5 processes. The first process p1 with execution time 12ms is shown being entered, showcasing the program's batch processing capability.

Figure 5: Batch Process Addition Progress



This screenshot continues the batch input process, showing the addition of processes p2 through p5. Each process receives individual confirmation: p2 (10ms), p3 (23ms), p4 (45ms), and p5 (19ms). This demonstrates organized handling of multiple process entries.

Figure 6: Sorted Process Queue Display

```
Enter your choice (1-6): 3

================================================================================
CURRENT PROCESS QUEUE (Sorted by Execution Time - Shortest Job First)
================================================================================
| Process ID | Process Name    | Execution Time | Waiting Time | Turnaround Time |
--------------------------------------------------------------------------------
| 2          | p2              | 10           ms | 0           ms | 0              ms |
| 1          | p1              | 12           ms | 0           ms | 0              ms |
| 5          | p5              | 19           ms | 0           ms | 0              ms |
| 1          | p1              | 23           ms | 0           ms | 0              ms |
| 3          | p3              | 23           ms | 0           ms | 0              ms |
| 4          | p4              | 45           ms | 0           ms | 0              ms |
================================================================================
Total processes in queue: 6
```

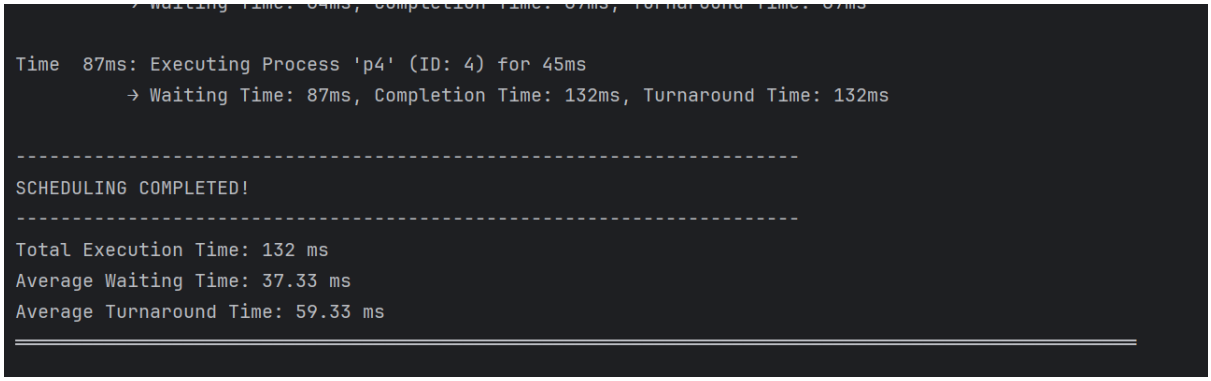
This screenshot presents the current process queue sorted by execution time in SJF order. The processes are automatically arranged in ascending order of execution time: p2(10ms), p1(12ms), p5(19ms), p1(23ms), p3(23ms), p4(45ms). This is the core SJF sorting in action.

Figure 7: SJF Execution Process (Part 1)

```
Enter your choice (1-6): 4

================================================================================
                SHORTEST JOB FIRST SCHEDULING EXECUTION
================================================================================
Starting SJF Scheduling at Time = 0ms
--------------------------------------------------------------------------------

Execution Order:
--------------------------------------------------------------------
Time   0ms: Executing Process 'p2' (ID: 2) for 10ms
         → Waiting Time: 0ms, Completion Time: 10ms, Turnaround Time: 10ms

Time  10ms: Executing Process 'p1' (ID: 1) for 12ms
         → Waiting Time: 10ms, Completion Time: 22ms, Turnaround Time: 22ms

Time  22ms: Executing Process 'p5' (ID: 5) for 19ms
         → Waiting Time: 22ms, Completion Time: 41ms, Turnaround Time: 41ms

Time  41ms: Executing Process 'p1' (ID: 1) for 23ms
         → Waiting Time: 41ms, Completion Time: 64ms, Turnaround Time: 64ms

Time  64ms: Executing Process 'p3' (ID: 3) for 23ms
         → Waiting Time: 64ms, Completion Time: 87ms, Turnaround Time: 87ms
```

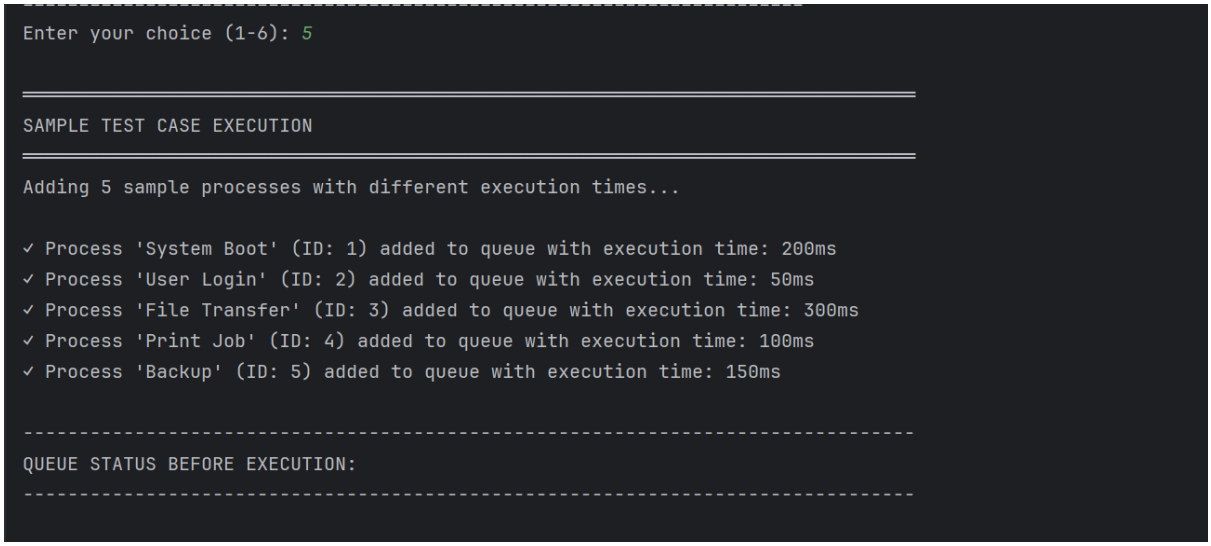
This screenshot captures the first part of SJF scheduling execution. Processes execute in sorted order: p2 at time 0ms (waiting time: 0ms), p1 at time 10ms (waiting time: 10ms), and p5 at time 22ms (waiting time: 22ms). The timeline shows how waiting time accumulates.

Figure 8: SJF Execution Results (Part 2)



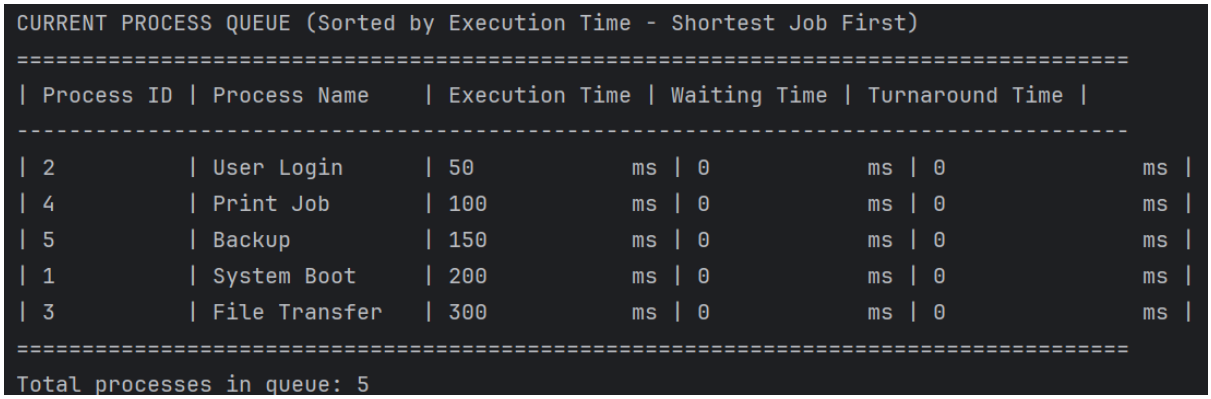
This screenshot shows completion of SJF execution with performance metrics. All 6 processes complete execution by time 132ms. The final calculations show average waiting time of 37.33ms and average turnaround time of 59.33ms, validating SJF effectiveness.

Figure 9: Sample Test Case Setup



This screenshot demonstrates the sample test case feature. Five realistic processes are automatically added: System Boot (200ms), User Login (50ms), File Transfer (300ms), Print Job (100ms), and Backup (150ms). This feature is useful for quick program demonstration.

Figure 10: Sample Test Queue Status



```
CURRENT PROCESS QUEUE (Sorted by Execution Time - Shortest Job First)
================================================================================
| Process ID | Process Name    | Execution Time | Waiting Time | Turnaround Time |
--------------------------------------------------------------------------------
| 2          | User Login      | 50           ms | 0           ms | 0              ms |
| 4          | Print Job       | 100          ms | 0           ms | 0              ms |
| 5          | Backup          | 150          ms | 0           ms | 0              ms |
| 1          | System Boot     | 200          ms | 0           ms | 0              ms |
| 3          | File Transfer   | 300          ms | 0           ms | 0              ms |
================================================================================
Total processes in queue: 5
```

This screenshot shows the sorted queue of sample processes before execution. The processes are correctly sorted by SJF: User Login(50ms), Print Job(100ms), Backup(150ms), System Boot(200ms), File Transfer(300ms). This confirms SJF sorting with descriptive names.

Figure 11: Sample Test Execution Results

```
Execution Order:
-----------------------------------------------------------------------
Time 132ms: Executing Process 'User Login' (ID: 2) for 50ms
          → Waiting Time: 132ms, Completion Time: 182ms, Turnaround Time: 182ms

Time 182ms: Executing Process 'Print Job' (ID: 4) for 100ms
          → Waiting Time: 182ms, Completion Time: 282ms, Turnaround Time: 282ms

Time 282ms: Executing Process 'Backup' (ID: 5) for 150ms
          → Waiting Time: 282ms, Completion Time: 432ms, Turnaround Time: 432ms

Time 432ms: Executing Process 'System Boot' (ID: 1) for 200ms
          → Waiting Time: 432ms, Completion Time: 632ms, Turnaround Time: 632ms

Time 632ms: Executing Process 'File Transfer' (ID: 3) for 300ms
          → Waiting Time: 632ms, Completion Time: 932ms, Turnaround Time: 932ms

-----------------------------------------------------------------------
SCHEDULING COMPLETED!
-----------------------------------------------------------------------
Total Execution Time: 932 ms
Average Waiting Time: 332.00 ms
Average Turnaround Time: 492.00 ms
```

This screenshot presents execution results of the sample test case. Total execution time is 932ms, with average waiting time of 332.00ms and average turnaround time of 492.00ms. This demonstrates SJF performance on realistic workload examples.

Figure 12: Program Termination

```
Enter your choice (1-6): 6

=======================================================================
Thank you for using SJF Process Scheduler!
Exiting program...
=======================================================================

Process finished with exit code 0
```

This final screenshot shows the program's graceful termination. The user selects option 6 to exit, and the program displays a thank you message with exit code 0, indicating successful completion and proper program closure.

5.2 Sample output

```
======================================================
    SHORTEST JOB FIRST CPU SCHEDULING SIMULATION
    Linked List based Priority Queue Implementation
======================================================

Enter number of processes: 3

------------------------------------------------------
ENTER PROCESS DETAILS
------------------------------------------------------

Process 1:
  Process ID: 101
  Process Name: System Boot
  Execution Time (ms): 200
✓ Added: System Boot (ID: 101, Time: 200ms)

Process 2:
  Process ID: 102
  Process Name: User Login
  Execution Time (ms): 50
✓ Added: User Login (ID: 102, Time: 50ms)
```

```
Process 3:
  Process ID: 103
  Process Name: File Transfer
  Execution Time (ms): 150
✓ Added: File Transfer (ID: 103, Time: 150ms)

======================================================
PROCESSES AFTER SJF SORTING
======================================================
ID      Process Name            Execution Time
--      ------------            --------------
102     User Login              50ms
103     File Transfer           150ms
101     System Boot             200ms
```

```
======================================================
EXECUTION IN SJF ORDER
======================================================

Execution Timeline:
----------------------------------------------------------------------
Time    Process         Exec Time       Wait Time       Turnaround
----    -------         ---------       ---------       ----------
0ms     User Login      50ms            0ms             50ms
50ms    File Transfer   150ms           50ms            200ms
200ms   System Boot     200ms           200ms           400ms

======================================================
SCHEDULING RESULTS SUMMARY
======================================================
Total Processes Executed: 3
Total Execution Time: 400ms
Average Waiting Time: 83.33 ms
Average Turnaround Time: 216.67 ms
```

5.2.1 Analysis of Sample Output

1. **Sorting Verified**: Processes automatically sorted as 50ms, 150ms, 200ms
2. **SJF Execution**: Shortest job (User Login) executes first
3. **Time Calculations**:
    - Waiting times: 0ms, 50ms, 200ms
    - Turnaround times: 50ms, 200ms, 400ms
4. **Performance Metrics**:
    - Average Waiting Time: 83.33ms
    - Average Turnaround Time: 216.67ms

5.2.2 Performance Analysis

5.2.2.1 Time complexity:

Operation	Complexity	Explanation
Insertion	O(n)	Need to find correct position in sorted list
Deletion	O(1)	Always remove from head
Sorting	$O(n^2)$	Insertion sort approach for n elements
Display	O(n)	Traverse entire list

5.2.2.2 Space complexity

- O(n): Linear space requirement
- Each process node stores: id, name, execTime, next pointer
- No additional arrays or data structures needed

6.SUMMARY

6.1 Project Achievements

This project successfully implemented the Shortest Job First CPU scheduling algorithm using a custom-built linked list data structure in Java. The key achievements include:

1. **Complete Implementation**: Fully functional SJF scheduler from scratch
2. **Data Structure Application**: Practical use of linked list for priority queue
3. **Algorithm Understanding**: Deep comprehension of CPU scheduling concepts
4. **Problem-Solving Skills**: Application of theoretical knowledge to practical problem

6.2 Learning Outcomes

Through this project, several important concepts were reinforced:

1. **Linked List Operations**: Insertion, deletion, traversal with sorting
2. **SJF Algorithm**: Understanding of optimal scheduling technique
3. **Time Calculations**: Waiting time and turnaround time computations
4. **System Simulation**: Modeling real-world OS behavior
5. **Java Programming**: Object-oriented design and implementation

6.3 Future Enhancements

The current implementation can be extended in several ways:

1. **Add Arrival Times**: Implement SJF with different arrival times

2. **Preemptive SJF**: Implement Shortest Remaining Time First (SRTF)
3. **GUI Interface**: Develop graphical visualization of scheduling
4. **Comparative Analysis**: Include other algorithms (Round Robin, Priority)
5. **Real-time Simulation**: Add process arrival during execution

6.4 Final Remarks

This project demonstrates the practical application of data structures in solving real-world operating system problems. The SJF algorithm, while theoretical in some aspects, provides valuable insights into efficient resource management. The linked list implementation offers a clear, understandable approach to priority queue management, making it an excellent educational tool for understanding both data structures and operating system concepts.

7.BIBLIOGRAPHY

- Up Grad
- GitHub
- OS class Notes

7.1 PROCESS ENTITY CLASS

```
public class ProcessEntity  {
    int processID;
    String processName;
    int executionTime;
    int waitingTime;
    int turnaroundTime;
    ProcessEntity next;

    ProcessEntity(int pid, String name, int execTime) {
        this.processID = pid;
        this.processName = name;
        this.executionTime = execTime;
        this.waitingTime = 0;
        this.turnaroundTime = 0;
        this.next = null;
    }

    void display() {
        System.out.printf("| %-10d | %-15s | %-13d | %-12d | %-15d |\n",
                processID, processName, executionTime, waitingTime,
turnaroundTime);
    }
}
```

7.2 PROCESS PRIORITY CLASS

```
public class ProcessPriorityQueue {
    private ProcessEntity front;
    private int size;

    public ProcessPriorityQueue() {
        front = null;
        size = 0;
```

```
    }


    public void enqueue(ProcessEntity process) {

        if (front == null || process.executionTime < front.executionTime) {
            process.next = front;
            front = process;
        } else {

            ProcessEntity current = front;
            while (current.next != null && current.next.executionTime <=
process.executionTime) {
                current = current.next;
            }
            process.next = current.next;
            current.next = process;
        }
        size++;
        System.out.println("✓ Process '" + process.processName + "' (ID: "
+ process.processID +
                ") added to queue with execution time: " +
process.executionTime + "ms");
    }


    public ProcessEntity dequeue() {
        if (isEmpty()) {
            System.out.println("Queue is empty!");
            return null;
        }
        ProcessEntity temp = front;
        front = front.next;
        size--;
        return temp;
    }
```

```
    public ProcessEntity peek() {
        return front;
    }


    public boolean isEmpty() {
        return front == null;
    }


    public int getSize() {
        return size;
    }


    public void displayQueue() {
        if (isEmpty()) {
            System.out.println("\nQueue is empty!");
            return;
        }

        System.out.println("\n" + "=".repeat(85));
        System.out.println("CURRENT PROCESS QUEUE (Sorted by
Execution Time - Shortest Job First)");
        System.out.println("=".repeat(85));
        System.out.println("| Process ID | Process Name    | Execution
Time | Waiting Time | Turnaround Time |");
        System.out.println("-".repeat(85));

        ProcessEntity current = front;
        int position = 1;
        while (current != null) {
            System.out.printf("| %-10d | %-15s | %-13d ms | %-12d ms | %-
15d ms |\n",
```

```
                current.processID, current.processName,
current.executionTime,
                current.waitingTime, current.turnaroundTime);
            current = current.next;
            position++;
        }
        System.out.println("=".repeat(85));
        System.out.println("Total processes in queue: " + size);
    }
}
```

7.3 SJF SCHEDULER CLASS

```
public class ProcessPriorityQueue {
    private ProcessEntity front;
    private int size;

    public ProcessPriorityQueue() {
        front = null;
        size = 0;
    }


    public void enqueue(ProcessEntity process) {

        if (front == null || process.executionTime < front.executionTime) {
            process.next = front;
            front = process;
        } else {

            ProcessEntity current = front;
            while (current.next != null && current.next.executionTime <=
process.executionTime) {
                current = current.next;
            }
            process.next = current.next;
            current.next = process;
        }
```

```
        size++;
        System.out.println("✓ Process '" + process.processName + "' (ID: "
+ process.processID +
                "') added to queue with execution time: " +
process.executionTime + "ms");
    }


    public ProcessEntity dequeue() {
        if (isEmpty()) {
            System.out.println("Queue is empty!");
            return null;
        }
        ProcessEntity temp = front;
        front = front.next;
        size--;
        return temp;
    }


    public ProcessEntity peek() {
        return front;
    }


    public boolean isEmpty() {
        return front == null;
    }


    public int getSize() {
        return size;
    }


    public void displayQueue() {
        if (isEmpty()) {
            System.out.println("\nQueue is empty!");
```

```
            return;
        }

        System.out.println("\n" + "=".repeat(85));
        System.out.println("CURRENT PROCESS QUEUE (Sorted by
Execution Time - Shortest Job First)");
        System.out.println("=".repeat(85));
        System.out.println("| Process ID | Process Name    | Execution
Time | Waiting Time | Turnaround Time |");
        System.out.println("-".repeat(85));

        ProcessEntity current = front;
        int position = 1;
        while (current != null) {
            System.out.printf("| %-10d | %-15s | %-13d ms | %-12d ms | %-
15d ms |\n",
                    current.processID, current.processName,
current.executionTime,
                    current.waitingTime, current.turnaroundTime);
            current = current.next;
            position++;
        }
        System.out.println("=".repeat(85));
        System.out.println("Total processes in queue: " + size);
    }
}
```

7.4 SJF PROCESS SCHEDULER

```
import java.util.*;
public class SJFProcessScheduler {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        SJFScheduler scheduler = new SJFScheduler();
        int choice;


        do {
```

```
System.out.println("\n" + "=".repeat(70));
System.out.println("MAIN MENU");
System.out.println("=".repeat(70));
System.out.println("1. Add a Process");
System.out.println("2. Add Multiple Processes (Batch Input)");
System.out.println("3. View Current Process Queue");
System.out.println("4. Execute Shortest Job First Scheduling");
System.out.println("5. Run Sample Test Case");
System.out.println("6. Exit");
System.out.println("=".repeat(70));
System.out.print("Enter your choice (1-6): ");

choice = scanner.nextInt();
scanner.nextLine();

switch (choice) {
    case 1:
        System.out.println("\n" + "-".repeat(50));
        System.out.println("ADD NEW PROCESS");
        System.out.println("-".repeat(50));

        System.out.print("Enter Process ID: ");
        int pid = scanner.nextInt();
        scanner.nextLine(); // Consume newline

        System.out.print("Enter Process Name: ");
        String name = scanner.nextLine();

        System.out.print("Enter Execution Time (ms): ");
        int execTime = scanner.nextInt();

        scheduler.addProcess(pid, name, execTime);
        break;

    case 2:
        System.out.println("\n" + "-".repeat(50));
```

```
                System.out.println("BATCH PROCESS INPUT");
                System.out.println("-".repeat(50));

                System.out.print("How many processes to add? ");
                int count = scanner.nextInt();
                scanner.nextLine(); // Consume newline

                for (int i = 1; i <= count; i++) {
                    System.out.println("\nProcess #" + i + ":");
                    System.out.print("  Process ID: ");
                    int batchPid = scanner.nextInt();
                    scanner.nextLine();

                    System.out.print("  Process Name: ");
                    String batchName = scanner.nextLine();

                    System.out.print("  Execution Time (ms): ");
                    int batchExecTime = scanner.nextInt();
                    scanner.nextLine();

                    scheduler.addProcess(batchPid, batchName,
batchExecTime);
                }
                System.out.println("\n✓ Successfully added " + count + "
processes!");
                break;

            case 3:
                scheduler.showQueueStatus();
                break;

            case 4:
                scheduler.executeSJF();
                break;

            case 5:
```

```
                runSampleTestCase(scheduler);
                break;

            case 6:
                System.out.println("\n" + "=".repeat(70));
                System.out.println("Thank you for using SJF Process
Scheduler!");
                System.out.println("Exiting program...");
                System.out.println("=".repeat(70));
                break;

            default:
                System.out.println("Invalid choice! Please enter 1-6.");
        }

        if (choice != 6) {
            System.out.print("\nPress Enter to continue...");
            scanner.nextLine();
        }

    } while (choice != 6);

    scanner.close();
}

private static void runSampleTestCase(SJFScheduler scheduler) {
    System.out.println("\n" + "═".repeat(80));
    System.out.println("SAMPLE TEST CASE EXECUTION");
    System.out.println("═".repeat(80));
    System.out.println("Adding 5 sample processes with different
execution times...\n");

    // Sample processes
    String[][] sampleProcesses = {
            {"P1", "System Boot", "200"},
            {"P2", "User Login", "50"},
```

```
            {"P3", "File Transfer", "300"},
            {"P4", "Print Job", "100"},
            {"P5", "Backup", "150"}
        };

        for (int i = 0; i < sampleProcesses.length; i++) {
            int pid = Integer.parseInt(sampleProcesses[i][0].substring(1));
            scheduler.addProcess(pid, sampleProcesses[i][1],
Integer.parseInt(sampleProcesses[i][2]));
        }

        System.out.println("\n" + "-".repeat(80));
        System.out.println("QUEUE STATUS BEFORE EXECUTION:");
        System.out.println("-".repeat(80));
        scheduler.showQueueStatus();

        System.out.println("\n" + "-".repeat(80));
        System.out.println("EXECUTING SJF SCHEDULING:");
        System.out.println("-".repeat(80));
        scheduler.executeSJF();

        System.out.println("\n" + "═".repeat(80));
        System.out.println("SAMPLE TEST COMPLETED!");
        System.out.println("═".repeat(80));
    }
}
```