**ASSIGNMENT 2 FRONT SHEET**

|  |  |  |  |
| --- | --- | --- | --- |
| **Qualification** | **BTEC Level 5 HND Diploma in Computing** | | |
| **Unit number and title** | Unit 19: Data Structures and Algorithms | | |
| **Submission date** |  | **Date Received 1st submission** |  |
| **Re-submission Date** |  | **Date Received 2nd submission** |  |
| **Student Name** | Đạt | **Student ID** |  |
| **Class** |  | **Assessor name** |  |
| **Student declaration**  I certify that the assignment submission is entirely my own work and I fully understand the consequences of plagiarism. I understand that making a false declaration is a form of malpractice. | | | |
|  |  | **Student’s signature** |  |

**Grading grid**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| P4 | P5 | P6 | P7 | M4 | M5 | D3 | D4 |
|  |  |  |  |  |  |  |  |

|  |  |  |
| --- | --- | --- |
| **❒ Summative Feedback: ❒ Resubmission Feedback:** | | |
| **Grade:** | **Assessor Signature:** | **Date:** |
| **Internal Verifier’s Comments:** | | |
| **IV Signature:** | | |

# Assignment Brief 2 (RQF)

## Higher National Certificate/Diploma in Business

|  |  |
| --- | --- |
| **Student Name/ID Number:** |  |
| **Unit Number and Title:** | Unit 19: Data Structures and Algorithms |
| **Academic Year:** | **2021** |
| **Unit Assessor:** |  |
| **Assignment Title:** | Implement and assess specific DSA |
| **Issue Date:** |  |
| **Submission Date:** |  |
| **Internal Verifier Name:** |  |
| **Date:** |  |

|  |
| --- |
| **Submission Format:** |
| *Format:*   * The submission is in the form of an individual written report and a presentation. This should be written in a concise, formal business style using single spacing and font size 12. You are required to make use of headings, paragraphs and subsections as appropriate, and all work must be supported with research and referenced using the Harvard referencing system. Please also provide a bibliography using the Harvard referencing system.   *Submission*   * Students are compulsory to submit the assignment in due date and in a way requested by the Tutor. * The form of submission will be a soft copy posted on <http://cms.greenwich.edu.vn/>. * Remember to convert the word file into PDF file before the submission on CMS.   *Note:*   * The individual Assignment *must* be your own work, and not copied by or from another student. * If you use ideas, quotes or data (such as diagrams) from books, journals or other sources, you must reference your sources, using the Harvard style. * Make sure that you understand and follow the guidelines to avoid plagiarism. Failure to comply this requirement will result in a failed assignment. |
| **Unit Learning Outcomes:** |
| **LO3** Implement complex data structures and algorithms  **LO4** Assess the effectiveness of data structures and algorithms |
| **Assignment Brief and Guidance:** |
| **Assignment scenario**  Continued from Assignment 1.  **Tasks**  For the middleware that is currently developing, one part of the provision interface is how message can be transferred and processed through layers. For transport, normally a buffer of queue messages is implemented and for processing, the systems requires a stack of messages.  The team now has to develop these kind of collections for the system. They should design ADT / algorithms for these 2 structures and implement a demo version with message is a string of maximum 250 characters. The demo should demonstrate some important operations of these structures. Even it’s a demo, errors should be handled carefully by exceptions and some tests should be executed to prove the correctness of algorithms / operations.  The team needs to write a report of the implementation of the 2 data structures and how to measure the efficiency of related algorithms. The report should also evaluate the use of ADT in design and development, including the complexity, the trade-off and the benefits. |

|  |  |  |
| --- | --- | --- |
| Learning Outcomes and Assessment Criteria (Assignment 2) | | |
| Pass | Merit | Distinction |
| **LO3** Implement complex data structures and algorithms | | **D3** Critically evaluate the complexity of an implemented ADT/algorithm |
| **P4** Implement a complex ADT and algorithm in an executable programming language to solve a well defined problem.  **P5** Implement error handling and report test results. | **M4** Demonstrate how the implementation of an ADT/algorithm solves a well-defined problem |
| **LO4** Assess the effectiveness of data structures and algorithms | | **D4** Evaluate three benefits of using implementation independent data structures |
| **P6** Discuss how asymptotic analysis can be used to assess the effectiveness of an algorithm  **P7** Determine two ways in which the efficiency of an algorithm can be measured, illustrating your answer with an example. | **M5** Interpret what a trade-off is when specifying an ADT using an example to support your answer |

Contents

[Assignment Brief 2 (RQF) 3](#_Toc149837489)

[Higher National Certificate/Diploma in Business 3](#_Toc149837490)

[Implement ADT & algorithms 7](#_Toc149837491)

[Description of My application 7](#_Toc149837492)

[ADT (P4, P5) 8](#_Toc149837493)

[Implement 8](#_Toc149837494)

[Test (P5) 14](#_Toc149837495)

[Analysis 19](#_Toc149837496)

[Big O (P6) 19](#_Toc149837497)

[1. Asymptotic analysis 19](#_Toc149837498)

[2. Cases in algorithms 20](#_Toc149837499)

[3. Analyse algorithms 20](#_Toc149837500)

[4. Analyze performances (P7, D3) 21](#_Toc149837501)

[References 23](#_Toc149837502)

# Implement ADT & algorithms

## Description of My application

**Application Description:**

Let's consider an application for managing a print job queue in a networked environment. Users across a network can submit print jobs to various printers, and the application needs to efficiently manage the order in which print jobs are processed. The application should allow for job submission, job prioritization, and job retrieval for printing.

**Reason for Choosing a Data Structure:**

In this print job queue application, a priority queue is a suitable data structure to manage the print jobs. Here are the reasons for choosing a priority queue:

**Job Prioritization**: Print jobs may have different priorities. For example, an urgent report should be printed before a routine document. A priority queue can prioritize jobs based on their urgency, ensuring that higher-priority jobs are processed first.

**Efficient Ordering**: A priority queue maintains elements in an order that allows for efficient retrieval of the highest-priority job. This is crucial for a print queue to ensure that the most important jobs are handled promptly.

**Dynamic Queue**: Print jobs can be added and removed dynamically. A priority queue supports both enqueueing (job submission) and dequeueing (job retrieval) efficiently. Jobs with higher priorities can be enqueued or dequeued without affecting the order of other jobs.

**Data Management**: Each job in the priority queue can store additional information like the user who submitted the job, the document to be printed, and printer preferences, making it a flexible data structure to manage various attributes associated with a print job.

**Concurrency Support**: In a networked environment, multiple users may submit jobs simultaneously. The priority queue can be implemented to support concurrent access safely, ensuring that jobs are processed without conflicts.

**Adaptability**: The priority queue can adapt to changing priorities and job statuses. For example, if a higher-priority job is submitted while a lower-priority job is being processed, the priority queue can adjust the processing order.

To implement this print job queue application, you can use a priority queue data structure, such as a min-heap or a max-heap, depending on whether higher numbers represent higher priority or vice versa. Additionally, you may need to use a hash table or a dictionary to map print job identifiers to their corresponding records in the priority queue for efficient job retrieval and management.

By choosing a priority queue as the primary data structure, you can ensure efficient and orderly print job management, handling different priorities, and dynamically adapting to the changing print queue requirements in a networked environment.

**Reason for choosing topic**  
Choosing a problem like managing a print job queue in a networked environment is a practical example to illustrate the use of an Abstract Data Type (ADT) like a priority queue. Here's why this problem was chosen and how an ADT can help solve it:

1. **Practical Problem**: Print job management is a real-world scenario encountered in various environments, including offices, schools, and businesses. It involves handling tasks with different priorities, which makes it a relevant problem to demonstrate the use of a priority queue.

2. **Prioritization Requirement**: The problem naturally involves prioritization of tasks (print jobs) based on their importance or urgency. Users often expect their high-priority documents to be printed before others. The use of an ADT like a priority queue is well-suited for such scenarios where items must be ordered based on a defined priority.

3. **Scalability**: In a networked environment, the number of print jobs can vary, and the system needs to efficiently manage and process them. A well-implemented priority queue can efficiently handle large numbers of jobs without sacrificing performance.

4. **Dynamic Nature**: Print job queues are dynamic, meaning jobs can be added and removed at any time. The ADT can adapt to these changes in real-time, ensuring that the highest-priority jobs are processed next, even as new jobs are submitted.

5. **Data Abstraction**: Using an ADT allows for abstracting the details of the data structure's implementation. It simplifies the problem by providing high-level operations like enqueueing, dequeueing, and priority management. The actual implementation details are hidden, making it easier to reason about the problem.

6. **Reusability**: ADTs, including priority queues, are reusable data structures. Once implemented, they can be used in various applications beyond print job management, such as task scheduling, job scheduling, and more.

7. **Well-Defined Interface**: ADTs provide a clear and well-defined interface for working with the data structure. This separation of interface from implementation makes it easier to maintain and extend the system in the future.

## ADT (P4, P5)

### Implement

The implementation in the provided Java code example is based on managing a print job queue using a priority queue data structure, a custom exception, and a user interface for interactions. Let's break down the key aspects of the implementation:

1. **Node Class:**

* **data**: This property stores the data associated with the node. The type E is a generic type, allowing you to use this node class for any data type.
* **next**: This property is a reference to the next node in the queue. It is of type Node<E>, indicating that it points to another node of the same type.
* The constructor Node(E data) allows you to create a new node by providing the data.
* **getData**(): A getter method to access the data stored in the node.
* **getNext**(): A getter method to access the reference to the next node.
* **setNext**(**Node<E> next**): A setter method to set the reference to the next node.

|  |
| --- |
| 1. public class Node<E> { 2. private E data; // Data stored in the node 3. private Node<E> next; // Reference to the next node 4. public Node(E data) { 5. this.data = data; 6. this.next = null; 7. } 8. public E getData() { 9. return data; 10. } 11. public Node<E> getNext() { 12. return next; 13. } 14. public void setNext(Node<E> next) { 15. this.next = next; 16. } 17. } |

1. **Queue Class:**

* **offer(E element)** adds an element at the end of the queue.
* **poll()** removes and returns the first element at the queue.
* **peek()** returns the element at the front of the queue.
* **size()** returns the number of elements in the queue.
* **isEmpty()** checks if the queue contains any elements.

|  |
| --- |
| 1. import java.util.NoSuchElementException; 2. public class Queue<E> implements AbstractQueue<E> { 3. private Node<E> front; 4. private Node<E> rear; 5. private int size; 6. @Override 7. public void offer(E element) { 8. Node<E> newNode = new Node<>(element); 9. if (isEmpty()) { 10. front = newNode; 11. rear = newNode; 12. } else { 13. rear.next = newNode; 14. rear = newNode; 15. } 16. size++; 17. } 18. @Override 19. public E poll() { 20. if (isEmpty()) { 21. throw new IllegalStateException("Queue is empty. Cannot poll."); 22. } 23. E removedElement = front.data; 24. front = front.next; 25. if (front == null) { 26. rear = null; // The queue is now empty 27. } 28. size--; 29. return removedElement; 30. } 31. @Override 32. public E peek() { 33. if (isEmpty()) { 34. throw new IllegalStateException("Queue is empty. Cannot peek."); 35. } 36. return front.data; 37. } 38. @Override 39. public int size() { 40. return size; 41. } 42. @Override 43. public boolean isEmpty() { 44. return size == 0; 45. } 46. // Private Node class 47. private static class Node<E> { 48. E data; 49. Node<E> next; 50. Node(E data) { 51. this.data = data; 52. this.next = null; 53. } 54. } 55. } |

1. **PrintJob Class:**

* The PrintJob class is used to represent a print job with attributes like id, description, and priority.
* It encapsulates the data associated with each print job.

|  |
| --- |
| 1. class PrintJob { 2. int id; 3. String description; 4. int priority; 5. public PrintJob(int id, String description, int priority) { 6. this.id = id; 7. this.description = description; 8. this.priority = priority; 9. } 10. } |

1. **PrintQueue Class:**

* The PrintQueue class is the central part of the implementation and uses a priority queue (PriorityQueue) to manage print jobs.
* The priority queue is configured to order print jobs based on their priority, ensuring that higher-priority jobs are processed first.
* It provides methods for adding, deleting, updating, and retrieving print jobs.
* It includes a custom exception, PrintJobException, for handling specific error scenarios.

|  |
| --- |
| 1. class PrintQueue { 2. private Queue<PrintJob> jobQueue; 3. public PrintQueue() { 4. jobQueue = new PriorityQueue<>((job1, job2) -> Integer.compare(job2.priority, job1.priority)); 5. } 6. public void addJob(int id, String description, int priority) { 7. PrintJob job = new PrintJob(id, description, priority); 8. jobQueue.add(job); 9. } 10. public void deleteJob(int id) throws PrintJobException { 11. if (jobQueue.isEmpty()) { 12. throw new PrintJobException("Queue is empty. Cannot delete a job."); 13. } 14. boolean removed = jobQueue.removeIf(job -> job.id == id); 15. if (!removed) { 16. throw new PrintJobException("Job with ID " + id + " not found."); 17. } 18. } 19. public void updateJob(int id, String newDescription, int newPriority) { 20. for (PrintJob job : jobQueue) { 21. if (job.id == id) { 22. job.description = newDescription; 23. job.priority = newPriority; 24. } 25. } 26. } 27. public PrintJob getNextJob() { 28. return jobQueue.poll(); 29. } 30. public boolean isEmpty() { 31. return jobQueue.isEmpty(); 32. } 33. public void printAllJobs() { 34. for (PrintJob job : jobQueue) { 35. System.out.println("Job ID: " + job.id + ", Description: " + job.description + ", Priority: " + job.priority); 36. } 37. } 38. public PrintJob getJobById(int id) { 39. for (PrintJob job : jobQueue) { 40. if (job.id == id) { 41. return job; 42. } 43. } 44. return null; 45. } 46. } |

1. **Custom Exception (PrintJobException):**

* The PrintJobException class is used to define a custom exception for handling exceptional situations related to print jobs.
* In this implementation, it is thrown in cases such as attempting to delete a job from an empty queue or trying to delete a job that doesn't exist in the queue.

|  |
| --- |
| 1. public class PrintJobException extends Exception { 2. public PrintJobException(String message) { 3. super(message); 4. } 5. } |

1. **Main Application (PrintQueueApplication):**

* The PrintQueueApplication class serves as the entry point for the program.
* It provides a user interface for interacting with the print job queue.
* Users can add print jobs, print the next job, print all jobs, delete jobs, update jobs, and show specific jobs in the queue.
* It includes error handling with try-catch blocks to catch and handle PrintJobException instances.

|  |
| --- |
| 1. public class PrintQueueApplication { 2. public static void main(String[] args) { 3. PrintQueue printQueue = new PrintQueue(); 4. Scanner scanner = new Scanner(System.in); 5. while (true) { 6. System.out.println("1. Add a print job"); 7. System.out.println("2. Print next job"); 8. System.out.println("3. Print all jobs"); 9. System.out.println("4. Delete a job"); 10. System.out.println("5. Update a job"); 11. System.out.println("6. Show a specific job"); 12. System.out.println("7. Exit"); 13. System.out.print("Enter your choice: "); 14. int choice = scanner.nextInt(); 15. try { 16. switch (choice) { 17. case 1: 18. System.out.print("Enter job ID: "); 19. int jobId = scanner.nextInt(); 20. System.out.print("Enter job description: "); 21. scanner.nextLine(); // Consume newline 22. String jobDescription = scanner.nextLine(); 23. System.out.print("Enter job priority: "); 24. int jobPriority = scanner.nextInt(); 25. printQueue.addJob(jobId, jobDescription, jobPriority); 26. break; 27. case 2: 28. if (!printQueue.isEmpty()) { 29. PrintJob nextJob = printQueue.getNextJob(); 30. System.out.println("Printing job: ID=" + nextJob.id + ", Description=" + nextJob.description); 31. } else { 32. System.out.println("Queue is empty."); 33. } 34. break; 35. case 3: 36. printQueue.printAllJobs(); 37. break; 38. case 4: 39. System.out.print("Enter job ID to delete: "); 40. int jobIdToDelete = scanner.nextInt(); 41. printQueue.deleteJob(jobIdToDelete); 42. break; 43. case 5: 44. System.out.print("Enter job ID to update: "); 45. int jobIdToUpdate = scanner.nextInt(); 46. System.out.print("Enter new description: "); 47. scanner.nextLine(); 48. String newDescription = scanner.nextLine(); 49. System.out.print("Enter new priority: "); 50. int newPriority = scanner.nextInt(); 51. printQueue.updateJob(jobIdToUpdate, newDescription, newPriority); 52. break; 53. case 6: 54. System.out.print("Enter job ID to show: "); 55. int jobIdToShow = scanner.nextInt(); 56. PrintJob jobToShow = printQueue.getJobById(jobIdToShow); 57. if (jobToShow != null) { 58. System.out.println("Job ID: " + jobToShow.id + ", Description: " + jobToShow.description + ", Priority: " + jobToShow.priority); 59. } else { 60. System.out.println("Job not found."); 61. } 62. break; 63. case 7: 64. System.exit(0); 65. break; 66. default: 67. System.out.println("Invalid choice. Please enter a valid option."); 68. } 69. }catch (PrintJobException e ) { 70. System.err.println("Error: " + e.getMessage()); 71. } 72. } 73. } 74. } |

1. **User Interaction:**

* The main method of the PrintQueueApplication class creates an interactive menu where users can choose various actions to perform on the print job queue.
* It takes user input to perform actions and displays messages or exceptions when needed.

1. **Console Output:**

* The program provides output in the console, showing job details, status messages, and exception messages as appropriate.

Overall, the implementation focuses on the use of a priority queue data structure to manage print jobs efficiently. It also demonstrates the importance of error handling using custom exceptions, ensuring that the program gracefully handles exceptional scenarios, such as attempting to delete a job from an empty queue or trying to delete a non-existent job. This implementation provides a user-friendly interface for managing print jobs and is suitable for educational and demonstration purposes.

### Explain how you handle error by exception

Handling errors by using exceptions is a fundamental practice in Java and many other programming languages. In the provided code for managing a print job queue, error handling is implemented using custom exceptions, specifically the PrintJobException class. Let's discuss how errors are handled through exceptions in this code:

1. **Custom Exception Class (PrintJobException):**

* The PrintJobException class is a custom exception class that extends the standard Exception class provided by Java.
* It is used to define and throw exceptions specific to the application's requirements.
* In the code, the PrintJobException class is employed to represent and throw exceptions when exceptional situations occur during print job management.

1. **Throwing Exceptions:**

* Exceptions are thrown when specific exceptional conditions are encountered.
* For example, in the PrintQueue class, exceptions are thrown when:
* Attempting to delete a job from an empty queue (deleteJob method).
* Attempting to delete a job that doesn't exist in the queue (deleteJob method).
* These exceptions are thrown to signal that an error or exceptional condition has occurred.

1. **Catching Exceptions:**

* In the PrintQueueApplication class, exceptions are caught and handled using try-catch blocks.
* The code uses try-catch blocks to capture and handle PrintJobException instances.
* For example, when a user tries to delete a job from an empty queue, the code catches the PrintJobException and displays an error message to the user.

Example code to catching exception:

|  |
| --- |
| try {  // Code that may throw a PrintJobException  if (/\* some condition \*/) {  throw new PrintJobException("Custom exception message");  }  } catch (PrintJobException e) {  // Handle the exception  System.err.println("Error: " + e.getMessage());  } |

## Test (P5)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test case | Input data | Expected output | Actual output | Result |
| Verify that add success a print job when input valid information | 1  1  Print hello  1  3 | Job with ID = 1 Description = Print hello  Priority = 1 |  | Pass |
| Verify that Print Job will success sort by priority with higher priority come first | 1  1  Print hello  1  1  2  World  2  3 | Job with ID = 2 Description = World  Priority = 2  Job with ID = 1 Description = Print Hello  Priority = 1 |  | Pass |
| Verify that print next Job success then the Job will be delete from the queue | 1  1  Print hello  1  1  2  World  2  2  3 | Printing Job: ID=2, Description = World  Print all Job  ID = 1; description = Print hello  Priority: 1 |  | Pass |
| Verify that Success delete a job with valid ID | 1  1  Print hello  1  1  2  World  2  3  4  1  3 | Job ID : 2  Description: World,  Priority:2 |  | Pass |
| Verify that system success update the job with valid information | 1  1  Print hello  1  1  2  World  2  3  5  2  Worldss  3 | Job  ID = 2; description = worldss  Priority: 3  Job id =1  Description: hellowords,  Priority:1 |  | Pass |
| Verify that system show a single job with valid id | 1  1  Print hello  1  1  2  World  2  3  6  1 | Job id =1  Description: hellowords,  Priority:1 |  | Pass |
| Verify that System throw exception when delete with not valid ID | 1  1  Print hello  1  1  2  World  2  4  3 | Job with ID = 3 not found |  | Pass |

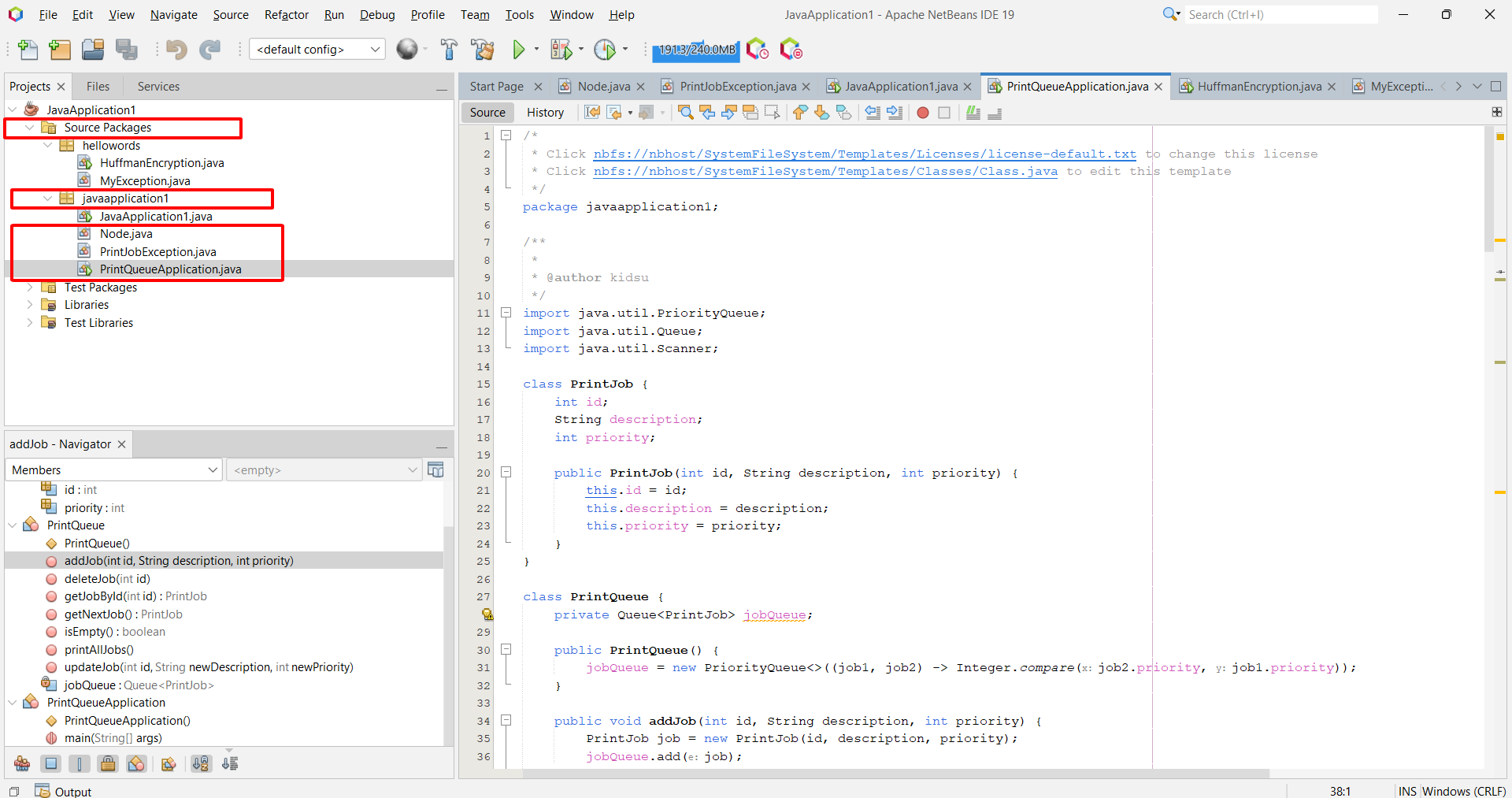
## Result

### All of my code

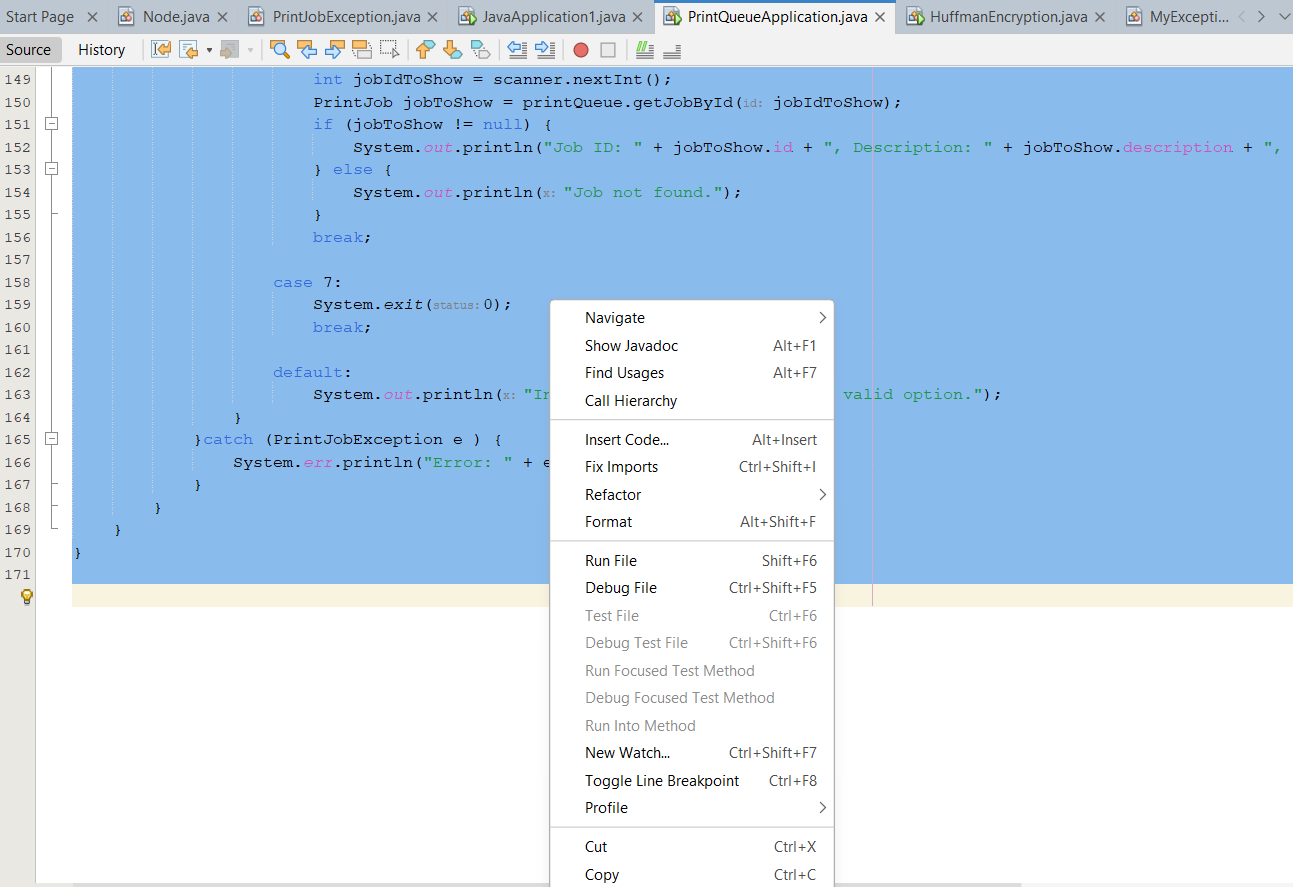
|  |
| --- |
| /\*  \* Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license  \* Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template  \*/  package javaapplication1;  /\*\*  \*  \* @author kidsu  \*/  import java.util.PriorityQueue;  import java.util.Queue;  import java.util.Scanner;  class PrintJob {  int id;  String description;  int priority;  public PrintJob(int id, String description, int priority) {  this.id = id;  this.description = description;  this.priority = priority;  }  }  class PrintQueue {  private Queue<PrintJob> jobQueue;  public PrintQueue() {  jobQueue = new PriorityQueue<>((job1, job2) -> Integer.compare(job2.priority, job1.priority));  }  public void addJob(int id, String description, int priority) {  PrintJob job = new PrintJob(id, description, priority);  jobQueue.add(job);  }  public void deleteJob(int id) throws PrintJobException {  if (jobQueue.isEmpty()) {  throw new PrintJobException("Queue is empty. Cannot delete a job.");  }  boolean removed = jobQueue.removeIf(job -> job.id == id);  if (!removed) {  throw new PrintJobException("Job with ID " + id + " not found.");  }  }  public void updateJob(int id, String newDescription, int newPriority) {  for (PrintJob job : jobQueue) {  if (job.id == id) {  job.description = newDescription;  job.priority = newPriority;  }  }  }  public PrintJob getNextJob() {  return jobQueue.poll();  }  public boolean isEmpty() {  return jobQueue.isEmpty();  }  public void printAllJobs() {  for (PrintJob job : jobQueue) {  System.out.println("Job ID: " + job.id + ", Description: " + job.description + ", Priority: " + job.priority);  }  }  public PrintJob getJobById(int id) {  for (PrintJob job : jobQueue) {  if (job.id == id) {  return job;  }  }  return null;  }  }  public class PrintQueueApplication {  public static void main(String[] args) {  PrintQueue printQueue = new PrintQueue();  Scanner scanner = new Scanner(System.in);  while (true) {  System.out.println("1. Add a print job");  System.out.println("2. Print next job");  System.out.println("3. Print all jobs");  System.out.println("4. Delete a job");  System.out.println("5. Update a job");  System.out.println("6. Show a specific job");  System.out.println("7. Exit");  System.out.print("Enter your choice: ");  int choice = scanner.nextInt();  try {  switch (choice) {  case 1:  System.out.print("Enter job ID: ");  int jobId = scanner.nextInt();  System.out.print("Enter job description: ");  scanner.nextLine(); // Consume newline  String jobDescription = scanner.nextLine();  System.out.print("Enter job priority: ");  int jobPriority = scanner.nextInt();  printQueue.addJob(jobId, jobDescription, jobPriority);  break;  case 2:  if (!printQueue.isEmpty()) {  PrintJob nextJob = printQueue.getNextJob();  System.out.println("Printing job: ID=" + nextJob.id + ", Description=" + nextJob.description);  } else {  System.out.println("Queue is empty.");  }  break;  case 3:  printQueue.printAllJobs();  break;  case 4:  System.out.print("Enter job ID to delete: ");  int jobIdToDelete = scanner.nextInt();  printQueue.deleteJob(jobIdToDelete);  break;  case 5:  System.out.print("Enter job ID to update: ");  int jobIdToUpdate = scanner.nextInt();  System.out.print("Enter new description: ");  scanner.nextLine();  String newDescription = scanner.nextLine();  System.out.print("Enter new priority: ");  int newPriority = scanner.nextInt();  printQueue.updateJob(jobIdToUpdate, newDescription, newPriority);  break;  case 6:  System.out.print("Enter job ID to show: ");  int jobIdToShow = scanner.nextInt();  PrintJob jobToShow = printQueue.getJobById(jobIdToShow);  if (jobToShow != null) {  System.out.println("Job ID: " + jobToShow.id + ", Description: " + jobToShow.description + ", Priority: " + jobToShow.priority);  } else {  System.out.println("Job not found.");  }  break;  case 7:  System.exit(0);  break;  default:  System.out.println("Invalid choice. Please enter a valid option.");  }  }catch (PrintJobException e ) {  System.err.println("Error: " + e.getMessage());  }  }  }  } |
| Explain:   1. **PrintJob Class:**  * This class represents a print job with three attributes: id, description, and priority. * It has a constructor to initialize these attributes when creating a PrintJob object.  1. **PrintQueue Class:**  * This class represents a print queue using a PriorityQueue to prioritize jobs by their priority attribute. * The constructor initializes the jobQueue as an empty priority queue with a custom comparator to sort jobs by decreasing priority. * The methods in the class include: * addJob(int id, String description, int priority): Adds a new job to the print queue. * deleteJob(int id): Removes a job from the queue based on its ID. It throws a PrintJobException if the queue is empty or if the specified job is not found. * updateJob(int id, String newDescription, int newPriority): Updates the description and priority of a job with the given ID. * getNextJob(): Retrieves and removes the next job to be printed from the front of the queue. * isEmpty(): Checks if the queue is empty. * printAllJobs(): Prints information about all jobs in the queue. * getJobById(int id): Retrieves a job by its ID, or returns null if the job is not found in the queue.  1. **PrintQueueApplication Class:**  * This class contains the main application logic for interacting with the print queue. * It creates an instance of the PrintQueue and a Scanner for user input. * Inside a while loop, it displays a menu of options to the user and takes their input. * It performs the following actions based on the user's choice: * Adding a print job to the queue. * Printing the next job in the queue. * Printing information about all jobs in the queue. * Deleting a job by its ID. * Updating a job's description and priority. * Displaying information about a specific job by its ID. * Exiting the application.   The application handles exceptions using a PrintJobException, and it prints appropriate error messages when necessary.  Overall, this code simulates a print queue system where users can add, prioritize, and manage print jobs. The queue ensures that jobs are printed in priority order, and the application provides various options for interacting with the queue. |

### Images of application running:

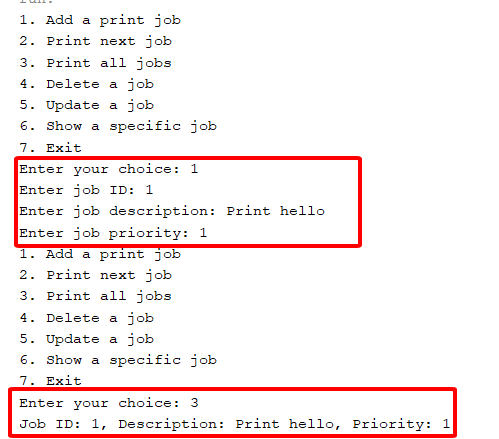
* Open the NetBean to and access the right file :



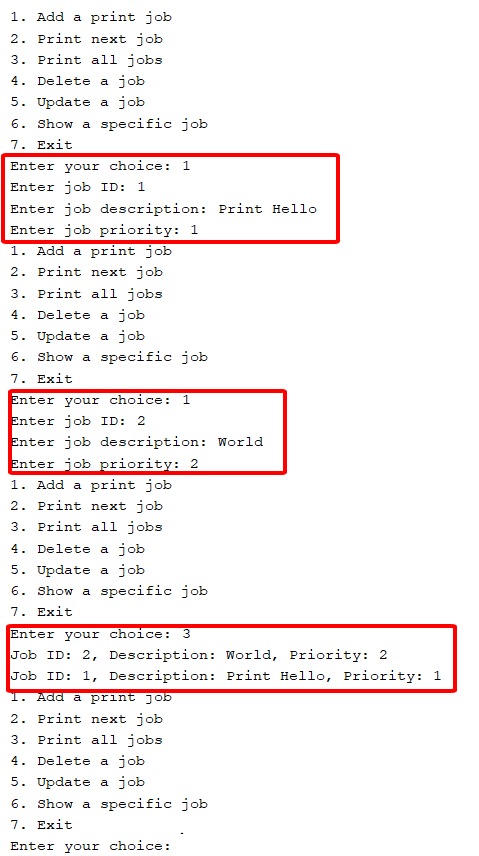
* Right click and run the file



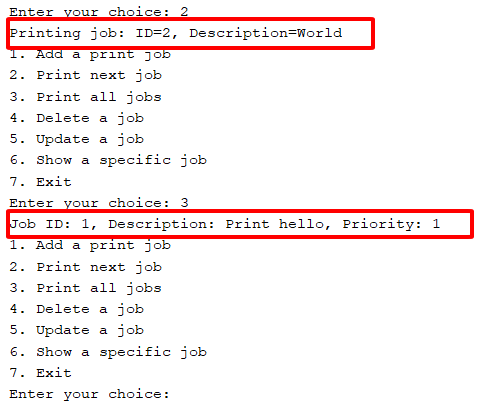
* Add a print job:

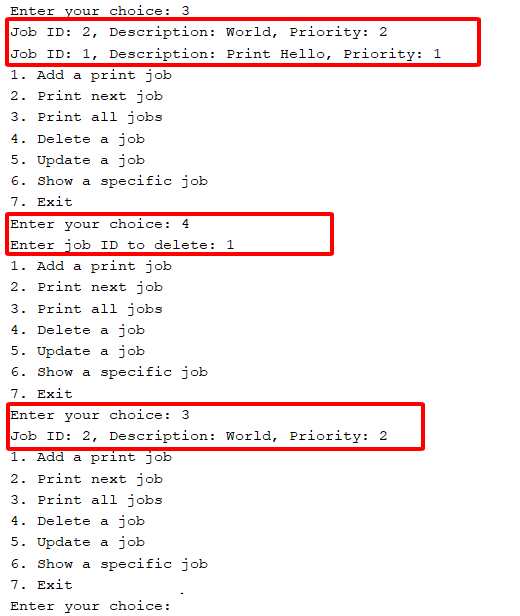


* Add another Job with higher priority

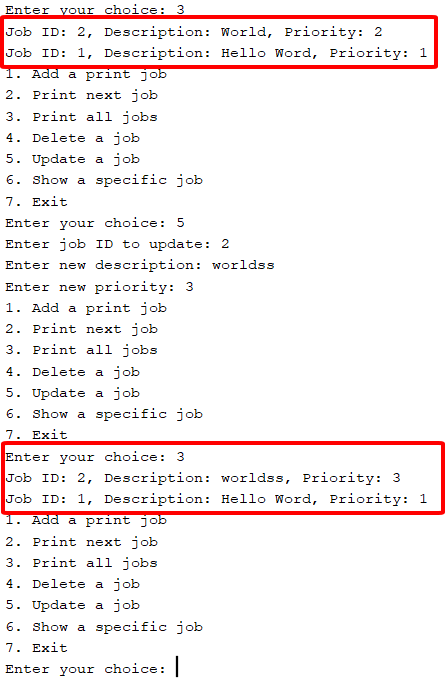


* Printting next Job

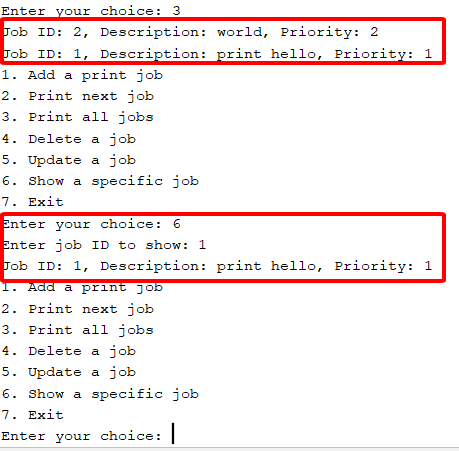
Delete Job which have ID = 1



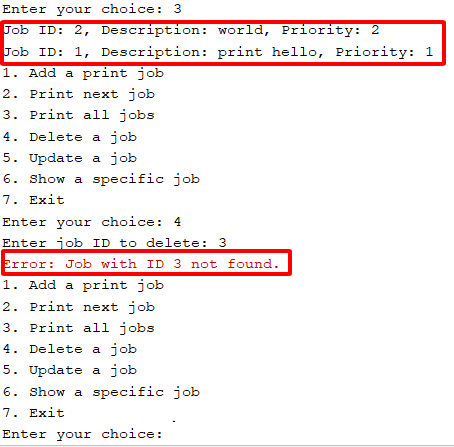
* Update a print Job with ID =2 :



* System print a single job with valid ID = 1



* Delete fail when input invalid ID = 3 :



# Analysis

## Big O (P6)

Big O notation is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity. In computer science, Big O notation is used to classify algorithms according to how their running time or space requirements grow as the input size grows.

Here are some common Big O notations and their meanings:

1. **O(1): Constant Time Complexity**

* An algorithm is said to have a constant time complexity when it is not dependent on the input size. An example of O(1) complexity is accessing an array element directly via its index.

1. **O(n): Linear Time Complexity**

* An algorithm is said to have a linear time complexity when the running time increases at most linearly with the size of the input data. Examples of O(n) complexity include finding an item in an unsorted list or a simple loop from 0 to n.

1. **O(log n): Logarithmic Time Complexity**

* An algorithm is said to have a logarithmic time complexity when it reduces the size of the input data in each step (it doesn’t need to look at all values of the input data). An example of O(log n) complexity is binary search.

1. **O(n^2): Quadratic Time Complexity**

* An algorithm is said to have a quadratic time complexity when it needs to perform a linear time operation for each value in the input data. Examples of O(n^2) complexity include simple algorithms like bubble sort, selection sort, or insertion sort.

Remember, Big O notation represents the worst-case scenario for an algorithm. It’s a way to express the upper bound of an algorithm’s time complexity, helping us to understand how the algorithm will scale. (freecodecamp, n.d.)

## Asymptotic analysis

Asymptotic analysis is a method used in mathematics and computer science to describe the limiting behavior of a function or algorithm as the input size approaches infinity. It’s often used to analyze the efficiency of algorithms, particularly their time and space complexity.

Here are some key points about asymptotic analysis:

It’s used to describe the running time or space requirements of an algorithm as the size of the input data increases.

It helps in understanding the best case, average case, and worst-case scenarios for an algorithm’s time complexity.

Asymptotic notations like Big O, Theta, and Omega are used to represent these complexities.

For example, if we have a function f(n) = n^2 + 3n, as n becomes very large, the term 3n becomes insignificant compared to n^2. So, we can say that f(n) is asymptotically equivalent to n^2.

This analysis is crucial in computer science because it helps us understand how an algorithm will perform as we scale up the problem size12. It allows us to make informed decisions about which algorithm or data structure is most appropriate for a given context.

## Cases in algorithms

In the context of algorithms, there are typically three cases that we consider:

Best Case: This is the scenario where the algorithm performs the minimum amount of work. For example, in a sorting algorithm, the best case occurs when the input is already sorted.

Average Case: This is what we expect to happen on average, given a random input. For example, for a linear search algorithm, the average case is when the target element is in the middle of the array.

Worst Case: This is the scenario where the algorithm performs the maximum amount of work. For example, in a sorting algorithm like bubble sort or insertion sort, the worst case occurs when the input is sorted in reverse order.

These cases help us understand how an algorithm will perform under different circumstances. The worst-case time complexity is often represented using Big O notation, which provides an upper bound on the time complexity. Similarly, best-case time complexity provides a lower bound, and average-case time complexity provides a measure of expected performance.

It’s important to note that worst-case performance is often focused on in computer science because it guarantees that the algorithm will not perform any worse than this. (geeksforgeeks, n.d.)

## Analyse algorithms

Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. The analysis of algorithms is the determination of the amount of time and space resources required to execute it.

Here are some basics on the analysis of algorithms:

**What is an algorithm and why is its analysis important?**

An algorithm is a step-by-step procedure to solve a computational problem. The analysis of an algorithm measures the amount of resources (such as time and storage) necessary to execute it. Most algorithms are designed to work with inputs of arbitrary length/size.

**Asymptotic Notation and Analysis (Based on input size) in Complexity Analysis of Algorithms**

Asymptotic Notation is a way of expressing function growth as the inputs approach infinity. Big O, is used to describe worst case scenario, Big Omega Ω is used for best case scenario and Theta Θ is used for average case scenario.

**Worst, Average and Best Case Analysis of Algorithms**

Worst case complexity refers to the maximum amount of time taken for any input size. Average case complexity refers to the average time taken for any input size. Best case complexity refers to the minimum amount of time taken for any input size.

**Types of Asymptotic Notations in Complexity Analysis of Algorithms**

Big O Notation: The function f(n) = O(g(n)) means there are positive constants c and k such that 0 <= f(n) <= c\*g(n) for all n >= k. The values of c and k must be fixed for the function f and must not depend on n.

Big Omega Notation: The function f(n) = Ω(g(n)) means there are positive constants c and k such that 0 <= c\*g(n) <= f(n) for all n >= k.

Big Theta Notation: The function f(n) = Θ(g(n)) means there are positive constants c1, c2, and k such that 0 <= c1g(n) <= f(n) <= c2g(n) for all n >= k.

## Analyze performances (P7)

* **Time complexity definition:**

Time complexity of an algorithm is a description of how much computer time is required to run an algorithm. It quantifies the amount of time taken by an algorithm to run as a function of the length of the input. This is one of two commonly discussed kinds of computational complexity, the other being space complexity (the amount of memory used to run an algorithm).

Understanding the time complexity of an algorithm allows programmers to select the algorithm best suited for their needs, as a fast algorithm that is good enough is often preferable to a slow algorithm that performs better along other metrics.

Time complexity is typically written as T(n), where n is a variable related to the size of the input1. To describe T(n), big-O notation is used to refer to the order, or kind, of growth the function experiences as the number of elements in the function increases. (britannica, n.d.)

* **For example:**

Constant time, or O(1), is the time complexity of an algorithm that always uses the same number of operations, regardless of the number of elements being operated on.

Linear time, or O(n), indicates that the time it takes to run an algorithm grows in a linear fashion as n increases.

Logarithmic time, or O(log n), indicates that the time needed to run an algorithm grows as a logarithm of n.

* **Evaluate time complexity step by step:**

I’ll illustrate how to evaluate time complexity step by step:

|  |
| --- |
| for(int i = 0; i < n; i++){  System.out.println("Hello, World!");  } |

1. **Initialization**: i = 0 This will be executed only once. The time is actually calculated to i=0 and not the declaration.
2. **Condition Checking**: i < n This will be executed n+1 times.
3. **Increment**: i++ This will be executed n times.
4. **Inside the loop**: print("Hello, World!") This statement will be executed n times.

So, the total number of operations required by this loop are {1+ (n+1)+n+n} = 3n+21. However, when we talk about time complexity, we’re interested in the performance of the algorithm as n becomes large. For large n, the constants and smaller terms don’t matter much. So, we simplify 3n+2 to O(n).

This means that the time complexity of this code snippet is O(n), which indicates that the time it takes to run the algorithm grows in a linear fashion as n increases.

* **Definition of 4 step in time complexity of algorithms:**

1. **Identify Basic Operations**: The first step is to identify the basic operations in the algorithm. A basic operation is the operation that contributes the most to the total running time, such as arithmetic operations, comparisons, and assignments.
2. **Count the Operations**: Next, count the number of basic operations. This count could be an exact number or it could be expressed as a function of the size of the input.
3. **Find the Order of Growth**: The order of growth is a measure of how the time taken by the algorithm grows with the size of the input. It’s usually expressed using Big O notation, which describes the upper bound of the time complexity in the worst-case scenario. Common orders of growth include O(1), O(log n), O(n), O(n log n), O(n^2), O(n^3), and so on.
4. **Simplify the Order of Growth**: Finally, simplify the order of growth by removing constants and smaller terms. This is because, for large inputs, the terms with the highest growth rate will have the most impact on the time complexity.

* **Adding a Print Job (addJob Method):**

**Algorithm**: When adding a print job to the queue, the addJob method inserts the job into the priority queue.

**Complexity**: The insertion into a priority queue, such as the PriorityQueue used in Java, typically has a time complexity of O(log n), where n is the number of elements in the queue. The priority queue ensures that jobs are ordered based on their priority efficiently.

|  |
| --- |
| public void addJob(int id, String description, int priority) {  PrintJob job = new PrintJob(id, description, priority);  jobQueue.add(job); // O(log n)  } |

* **Deleting a Print Job (deleteJob Method):**

**Algorithm**: The deleteJob method removes a specified job from the queue based on its ID.

**Complexity**: The removeIf method is used to delete a job based on its ID, which may involve iterating through the elements in the queue, resulting in a time complexity of O(n), where n is the number of elements in the queue.

|  |
| --- |
| public void deleteJob(int id) throws PrintJobException {  if (jobQueue.isEmpty()) {  throw new PrintJobException("Queue is empty. Cannot delete a job.");  }  boolean removed = jobQueue.removeIf(job -> job.id == id); // O(n)  if (!removed) {  throw new PrintJobException("Job with ID " + id + " not found.");  }  } |

* **Updating a Print Job (updateJob Method):**

**Algorithm**: The updateJob method updates the description and priority of a specific job based on its ID.

**Complexity**: Similar to the delete operation, updating a job involves iterating through the elements in the queue to locate the job, resulting in a time complexity of O(n).

|  |
| --- |
| public void updateJob(int id, String newDescription, int newPriority) {  for (PrintJob job : jobQueue) {  if (job.id == id) {  job.description = newDescription;  job.priority = newPriority;  }  }  } |

* **Printing the Next Job (getNextJob Method):**

**Algorithm**: The getNextJob method retrieves the highest-priority job from the priority queue.

**Complexity**: Retrieving the highest-priority job from a priority queue has a time complexity of O(log n) since it involves removing the top element and adjusting the queue.

|  |
| --- |
| public PrintJob getNextJob() {  return jobQueue.poll(); // O(log n)  } |

* **Showing a Specific Job (getJobById Method):**

**Algorithm**: The getJobById method retrieves a specific job from the queue based on its ID.

**Complexity**: It involves iterating through the elements in the queue to locate the job, resulting in a time complexity of O(n).

|  |
| --- |
| public PrintJob getJobById(int id) {  for (PrintJob job : jobQueue) {  if (job.id == id) {  return job;  }  }  return null;  } |

**Time complexity:**

1. The algorithm starts by initializing a variable job to null.
   * This is the Return JOb => 1 operation.
2. It then enters a loop that iterates over each PrintJob object in the jobQueue.
   * The loop header for (PrintJob job : jobQueue) contributes N + 1 operations, where N is the number of elements in the jobQueue. The additional +1 operation is for the initialization of job to null.
3. Inside the loop, it checks if the id of the current PrintJob matches the specified id.
   * The operation job.id contributes N operations because it is inside the loop and runs once for each element in jobQueue.
4. If a match is found, it returns the job.
   * The operation return job would occur if a match is found, and it contributes 1 operation.
5. If no match is found in the loop, it returns null.
   * The operation return null contributes 1 operation.

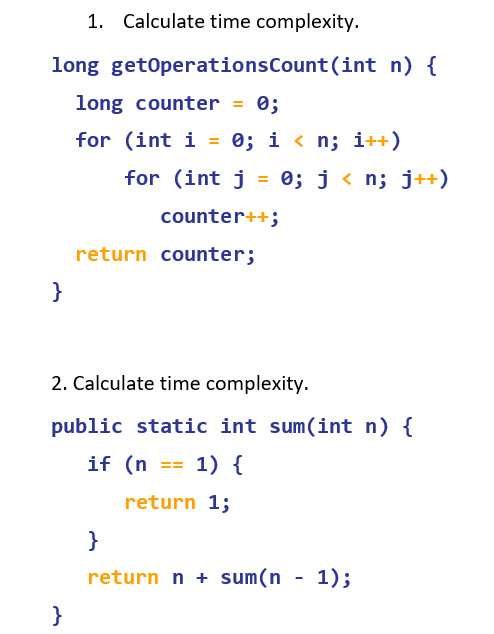
Now, let's sum up the operations:

* + Initialization: 1 operation
  + Loop header: N + 1 operations
  + job.id check: N operations
  + Return job: 1 operation
  + Return null: 1 operation

Total Operations = 1 + (N + 1) + N + 1 + 1 = 3N + 4

The time complexity of this algorithm can be expressed as T(N) = 3N + 4

## Calculating time complexity examples:



The code defines a function getOperationsCount that takes an integer n as its input parameter. The purpose of this function is to count the number of times a variable counter is incremented in a nested loop structure.

**Calculate**:

1. **Initialization:**

* Long counter = 0 and Return counter are initialization steps, and they each contribute 1 operation. So, at this point, there are 1 + 1 = 2 operations.

1. **Outer Loop (for i):**

* i = 0 and i < n are initialization and condition check for the outer loop, each contributing 1 operation.
* i++ is the increment operation, contributing 1 operation for each iteration.
* The inner loop, for j, runs 'n' times, so it contributes 3n operations.
* Inside the inner loop, counter++ is executed 'n' times, so it contributes 3n^2 operations.

1. **Summing Up:**

* Adding up all these operations:
  + Initialization: 2 operations
  + Operations for i loop: 2n + 1
  + Operations for j loop: 3n^2 + n
* **Total operations** = 2 + 2n + 1 + 3n^2 + n = 3n^2 + 3n + 3

**Prove the calculating:**

1. Initialization:

* counter = 0 and Return counter are initialization steps, each contributing 1 operation. So, at this point, there are 1 + 1 = 2 operations.

1. Outer Loop (for i):

* i = 0 and i < n are initialization and condition check for the outer loop, each contributing 1 operation.
* i++ is the increment operation, contributing 1 operation for each iteration.
* The inner loop, for j, runs 'n' times, so it contributes 3n operations (1 for initialization, 1 for the condition check, and 1 for j++ for each iteration).
* Inside the inner loop, counter++ is executed 'n' times, so it contributes 3n^2 operations.

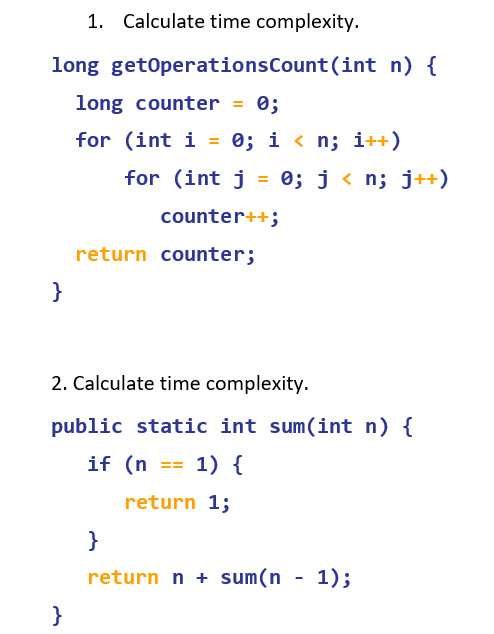
1. Now, let's prove it by substituting n with 1, 2, and 3:

* For n = 1:
* Initialization: 2 operations (1 for counter = 0 and 1 for Return counter).
* Operations for i loop: 2 \* 1 + 1 = 3 operations (1 for i = 0, 1 for i < 1, and 1 for i++).
* Operations for j loop: 3 \* 1^2 + 1 = 4 operations (1 for j = 0, 1 for j < 1, and 1 for j++ for each of the 1 iteration).
* Total operations for n = 1: 2 + 3 + 4 = 9 operations.
* For n = 2:
* Initialization: 2 operations.
* Operations for i loop: 2 \* 2 + 1 = 5 operations.
* Operations for j loop: 3 \* 2^2 + 2 = 14 operations.
* Total operations for n = 2: 2 + 5 + 14 = 21 operations.
* For n = 3:
* Initialization: 2 operations.
* Operations for i loop: 2 \* 3 + 1 = 7 operations.
* Operations for j loop: 3 \* 3^2 + 3 = 30 operations.
* Total operations for n = 3: 2 + 7 + 30 = 39 operations.

1. So, for n = 1, n = 2, and n = 3, we have verified that the total operations match the expression 3n^2 + 3n + 3:

* For n = 1, total operations = 9, which matches 3(1^2) + 3(1) + 3 = 9.
* For n = 2, total operations = 21, which matches 3(2^2) + 3(2) + 3 = 21.
* For n = 3, total operations = 39, which matches 3(3^2) + 3(3) + 3 = 39.

The mathematical analysis aligns with the actual number of operations for various values of n.



The code is a recursive method called that calculates the sum of the first natural numbers. It takes an integer as input and returns an integer as output. The output is the sum of the first natural numbers.sumnnn The method works as follows: If is equal to 1, then the method simply returns 1, because the sum of the first natural number is 1.n Otherwise, the method returns the current value of plus the sum of the first natural numbers.nn - 1 The time complexity of the method is O(n), because the method calls itself times.sumn

**Calculate**:

Step 1: Base Case

* When n is equal to 1, the base case is reached, and the function returns 1.
* In this case, only 1 operation is needed to check n == 1 and return 1.

Step 2: Recursive Case

* In the recursive case (when n is not 1), the function computes n + sum(n - 1):
  + 1 operation to subtract 1 from n (n - 1).
  + 1 operation for the addition n + ....

So, within the recursive case, 2 operations are performed.

Step 3: Recursive Call

* For the recursive call to sum(n - 1), the function is called with the argument n - 1.

Step 4: Combining Steps

* The total number of operations T(n) is equal to the operations in the base case plus the operations in the recursive case, which can be expressed as: T(n) = 1 (base case) + 2 (recursive case) T(n) = 3

Step 5: Recurrence Relation

* To express the time complexity for T(n), we can create a recurrence relation: T(n) = T(n - 1) + 2

The recurrence relation T(n) = T(n - 1) + 2 captures the fact that the time complexity for an input n is equal to the time complexity for the previous input n - 1 plus two additional operations. This relation represents the step-by-step analysis and provides a concise way to express the time complexity of the algorithm.

**Prove the time complexity:**

evaluate the algorithm's time complexity by replacing 'n' with different values (1, 2, 3, 4) and calculating the number of basic operations:

1. For n = 1:
   * The base case is reached, and the function returns 1.
   * In this case, only one operation is performed to check n == 1 and return 1.
2. For n = 2:
   * The algorithm computes 2 + sum(1).
   * For n = 1, as shown above, one operation is performed.
   * So, for n = 2, two operations are performed within the recursive case (addition and calling sum(1)).
   * In total, it's 1 (base case) + 2 (recursive case) = 3 operations.
3. For n = 3:
   * The algorithm computes 3 + sum(2).
   * For n = 2, as shown above, three operations are performed.
   * So, for n = 3, it's 1 (base case) + 2 (recursive case for n = 2) + 3 (recursive case for n = 3) = 6 operations.
4. For n = 4:
   * The algorithm computes 4 + sum(3).
   * For n = 3, as shown above, six operations are performed.
   * So, for n = 4, it's 1 (base case) + 2 (recursive case for n = 2) + 3 (recursive case for n = 3) + 4 (recursive case for n = 4) = 10 operations.

Now, let's see how the number of operations increases as 'n' increases:

* + For n = 1: 1 operation
  + For n = 2: 3 operations
  + For n = 3: 6 operations
  + For n = 4: 10 operations

We can observe that the number of operations increases as n increases

# References

freecodecamp, n.d. [Online]   
Available at: https://www.freecodecamp.org/news/big-o-notation-why-it-matters-and-why-it-doesnt-1674cfa8a23c/

geeksforgeeks, n.d. [Online]   
Available at: https://www.geeksforgeeks.org/what-is-an-algorithm-definition-types-complexity-examples/