TRINITY COLLEGE
DEPARTMENT OF COMPUTER SCIENCE
CPSC 215: Data Structures and Algorithms
Instructor: Dr. Chandranil Chakraborttii
Spring 2025

Assignment 3

Academic Honesty Policy

In working on programming assignments, you may discuss broad issues of interpretation and understanding and general approaches to a solution. However, conversion to a specific solution or to program code must be your own work. Programming assignments are expected to be the work of the individual student, designed, and coded by him or her alone. Violations are easy to identify and will be dealt with promptly according to the Academic Integrity and Intellectual Dishonesty outlined in the Student Handbook.

- Copying another person's programs or encouraging or assisting another person to commit plagiarism is cheating. In particular, the following activities are strictly prohibited:
- Giving and receiving help in the actual development of code or writing of an assignment.
- Looking at another person's code or showing your code to another person.
- Sharing a copy of all or part of your code regardless of whether that copy is on paper or in a computer file.
- Turning in the work of any other person(s) (former students, friends, textbook authors, people on the Internet, etc.) and representing it as your own work.
- Fabricating compilation or execution results.
- Use of AI ChatBots for any help regarding this assignment is **strictly prohibited.**

The penalty for cheating is a failing grade (F) for the course, and the student is asked to appear at an Academic Dishonesty Hearing. In addition, the College also places a record of the incident in the student's permanent record. It is your responsibility to protect your work from unauthorized access. Do not discard copies of your programs in public places. Do not leave computers unattended and copies of output lying around.

———————————————————————————————————————

Objective:

- Practice with double linked lists.
- Understand and Practice with Stacks.
- Evaluate Algebraic expressions with Stacks.

———————————————————————————————————————

Part I: Convert the following Infix expressions to Prefix and Postfix

a. A+B/C*(D-A)^F^H
b. ((P+R)∗Q)^(X/(Y+Z))

***Note**: You need to **show steps** as shown in the lecture slides. Remember operator precedence and assume / and + has higher precedence over * and – respectively. If the expression is invalid or cannot be evaluated, please indicate it and explain your reasoning.*

Part II: Evaluate the following expressions:

a. 4 5 + 10 × 6 2 ÷ − (Postfix)
b. 3 4 ∗ (5 6 −) 2 3 + ∗ (Postfix)
c. – 29 + 5 * 4 6 (Prefix)
d. × 5 ÷ 6 + 6 × 3 2 (Prefix)

***Note**: You need to **show steps** as shown in the lecture slides. If the expression is invalid or cannot be evaluated, please indicate it and explain your reasoning.*

Part III: Task Management System using Double Linked Lists

The assignment involves designing a task management system using double linked lists. The system consists of four main classes:

- Node.java ***(No Need to change)***
- Task.java ***(Need to complete)***
- TaskManager.java ***(Need to complete)***
- Tester.java ***(No Need to change)***

Getting Started

- Open Eclipse and create a Java Project (File -> New -> Java Project)
- Give a project name (For example: assignment_3_CPSC_215)
- Go to Project -> Right Click -> New Package
- Give a package name (For example: assignment_3_CPSC215)
- Now to create the classes, go to package -> Right Click -> New Java class.
- Paste contents of the given files. Keep the first line in place (package "package_name";)
- Understand the code structure before starting to write code.

Class Description:

Node.java ***(No Need to change)***

- **Review** the Node class represents a node in a doubly linked list. Here's a summary:
  - T data: Holds the node's data.
  - Node<T> prev: Reference to the previous node.
  - Node<T> next: Reference to the next node.
  - public Node(T data): Constructor initializes data and sets prev and next to null.

Task.java ***(Need to complete)***

- **Review** the Task class represents individual tasks in the task management system.
  - Each task has the following attributes:
  - **taskId**: An integer representing the unique identifier of the task.
  - **taskName**: A string representing the name or title of the task.
  - **taskDescription**: A string describing the details or specifics of the task.
  - **dueYear**, **dueMonth**, **dueDay**: Integers representing the year, month, and day when the task is due, respectively.
  - **priority**: An integer representing the priority level of the task.
- **Implement** getters and setters for accessing and modifying these attributes.
- Additionally, **implement** the toString() method to provide a string representation of the task.

TaskManager.java: ***(Need to complete)***

- The **TaskManager** class manages a collection of tasks using a linked list data structure.
- It maintains a doubly linked list of tasks, allowing for efficient addition, removal, updating, and retrieval of tasks.
- **Implement** the class methods to perform the following operations:
  - **addTask(Task task):** Adds a new task to the task list.
  - **removeTask(int taskId):** Removes a task from the list based on its ID.
  - **updateTask(int taskId, Task updatedTask):** Updates an existing task with new information.
  - **getTask(int taskId):** Retrieves a task from the list based on its ID.
  - **displayAllTasks():** Displays all tasks currently stored in the task list.

Test.Java: ***(No Need to change)***

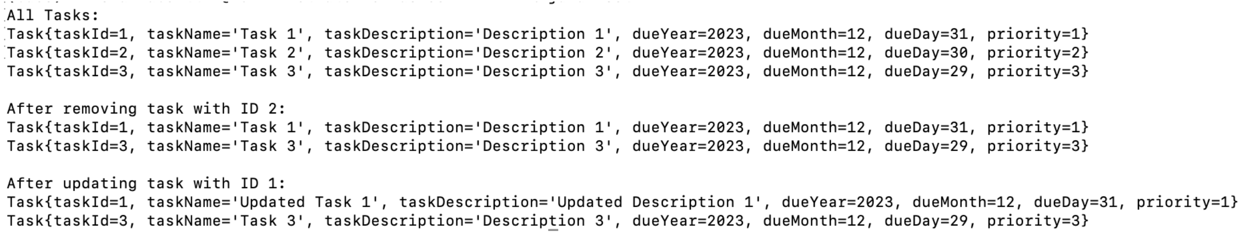
- The **Tester** class serves as the entry point for the task management system.
- It contains the **main()** method, where instances of the **TaskManager** class are created and various operations on tasks are demonstrated.
- In the **main()** method, sample tasks are created, added to the task manager, and then operations such as removal and updating are performed.
- The results of these operations are displayed to the console.

Overall, the task management system provides a flexible and efficient way to manage tasks, allowing users to easily add, remove, update, and retrieve tasks as needed. The system's design ensures scalability and ease of use for task management purposes.

To do:

- Review the node class and the code structure.
- Complete the Task class.
- Implement the methods in TaskManager class.
- Use the Tester class to test the implementation.

Sample Output

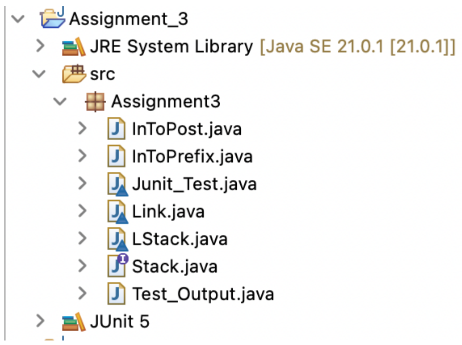


```
All Tasks:
Task{taskId=1, taskName='Task 1', taskDescription='Description 1', dueYear=2023, dueMonth=12, dueDay=31, priority=1}
Task{taskId=2, taskName='Task 2', taskDescription='Description 2', dueYear=2023, dueMonth=12, dueDay=30, priority=2}
Task{taskId=3, taskName='Task 3', taskDescription='Description 3', dueYear=2023, dueMonth=12, dueDay=29, priority=3}

After removing task with ID 2:
Task{taskId=1, taskName='Task 1', taskDescription='Description 1', dueYear=2023, dueMonth=12, dueDay=31, priority=1}
Task{taskId=3, taskName='Task 3', taskDescription='Description 3', dueYear=2023, dueMonth=12, dueDay=29, priority=3}

After updating task with ID 1:
Task{taskId=1, taskName='Updated Task 1', taskDescription='Updated Description 1', dueYear=2023, dueMonth=12, dueDay=31, priority=1}
Task{taskId=3, taskName='Task 3', taskDescription='Description 3', dueYear=2023, dueMonth=12, dueDay=29, priority=3}
```

Part IV: Convert Infix to Postfix and Prefix using a Stack

In this exercise, we will use a stack to convert an infix expression to its Postfix and Prefix form. You will need the following files to get started:

- Link.java *(No Need to change)*
- Stack. java *(No Need to change)*
- LStack.java *(No Need to change)*
- Test_output. java *(No Need to change)*
- InToPost. Java ***(Need to change)***
- IntoPrefix.java ***(Need to change)***
- Junit_Test. java *(**Need to change**)*



```
Assignment_3
  JRE System Library [Java SE 21.0.1 [21.0.1]]
  src
    Assignment3
      InToPost.java
      InToPrefix.java
      Junit_Test.java
      Link.java
      LStack.java
      Stack.java
      Test_Output.java
  JUnit 5
```

Class Description:

Link.java:

- **Description**: This class represents a node in a singly linked list. Each node contains an element of type char and a reference to the next node in the list.
- **Purpose**: It serves as a building block for creating linked list data structures.

LStack.java:

- **Description**: This class implements a stack using a linked list. It provides methods for stack operations such as push, pop, and topValue.
- **Purpose**: It provides a data structure for managing elements in a Last-In-First-Out (LIFO) manner.

Stack.java:

- **Description**: This interface defines the operations supported by a stack data structure, including methods for pushing, popping, getting the top element, and checking if the stack is empty.
- **Purpose**: It serves as a contract for implementing different types of stacks, allowing for interchangeability and abstraction.

InToPost.java:

- **Description**: This class converts infix expressions to postfix notation using a stack data structure.
- **Purpose**: It provides a method to convert infix expressions to postfix notation, which can be useful in parsing and evaluating arithmetic expressions.

InToPrefix.java:

- **Description**: This class converts infix expressions to prefix notation using a stack data structure.
- **Purpose**: It provides a method to convert infix expressions to prefix notation, which can be useful in parsing and evaluating arithmetic expressions in certain scenarios.

Test_Output.java:

- **Description**: This class contains the main method and is used to test the functionality of the infix to postfix and infix to prefix conversion classes.
- **Purpose**: It allows for the execution and testing of the conversion algorithms to ensure they produce the expected results.

Junit_Test.java

- The primary responsibility of this class is to ensure that the **InToPrefix** and **InToPost** 5lasses correctly converts infix expressions to prefix notation using **inToPrefixConvert** and **inToPost_convert** methods.

Algorithms for Conversion:

Infix to Postfix Conversion:

- Initialize an empty stack for operators and an empty list for the output.
- Scan the infix expression from left to right.
- If an operand is encountered, add it to the output list.
- If an operator is encountered:
- While the stack is not empty and the precedence of the current operator is less than or equal to the precedence of the operator at the top of the stack, pop the operator from the stack and add it to the output list.
- Push the current operator onto the stack.
- If a left parenthesis is encountered, push it onto the stack.
- If a right parenthesis is encountered, pop operators from the stack and add them to the output list until a left parenthesis is encountered. Pop and discard the left parenthesis.
- Repeat steps 3-6 until the end of the infix expression.
- Pop any remaining operators from the stack and add them to the output list.

Infix to Prefix Conversion:

- Reverse the infix expression.
- Replace each left parenthesis with a right parenthesis and vice versa.
- Obtain the postfix expression using the algorithm for infix to postfix conversion.
- Reverse the postfix expression to get the prefix expression.

JUnit Tests: Complete the provided Junit test class for testing the infix to postfix and infix to prefix conversion functionality. Run the test classes in your IDE to execute the test cases. Ensure that all tests pass without any errors or failures.

To do:

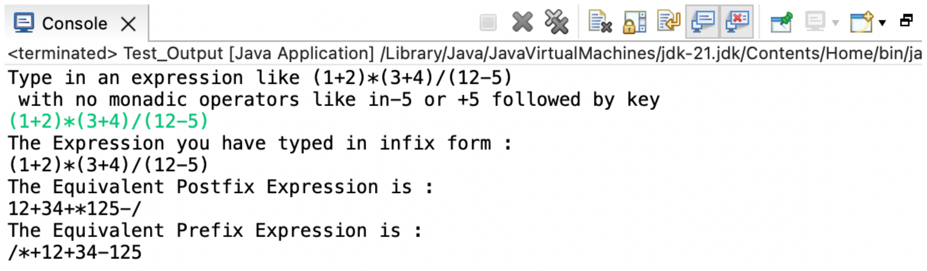
- Review code structure and paste contents of each class.
- Complete methods in IntoPost.java and IntoPrefix.java following the given algorithms.
- Verify and test your program for correctness. Use the Test file (Test_output.java)
- Complete Junit tests and ensure that each test cases pass.

Setting up JUnit Testing:

- On the project folder, Right Click -> New -> Other.
- In the wizard's section, type JUnit and select JUnit Test case -> Next.
- Provide the same package name (Eg: assignment_3_CPSC_215), Give name JUnit_Test
- Paste contents of the given files. Keep the first line in place (package "package_name";)
- Paste contents and Review: JUnit_Test.java (partially completed)
- **Check your JUnit Test:** Right Click on JUnit.java -> (Right Click) -> Run As -> JUnit Test
- You should see your tests pass.

Runs: 2/2 Errors: 0 Failures: 0

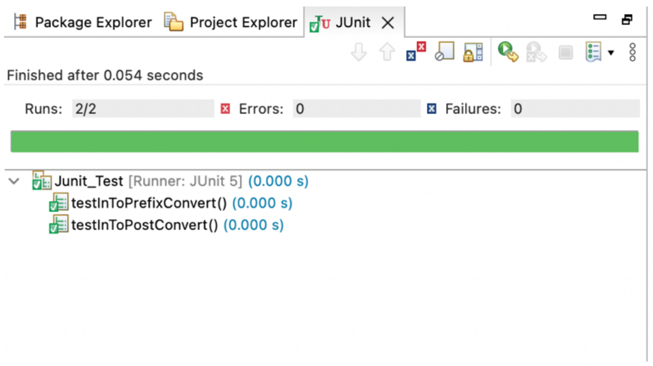
Sample Output



Console

```
<terminated> Test_Output [Java Application] /Library/Java/JavaVirtualMachines/jdk-21.jdk/Contents/Home/bin/ja
Type in an expression like (1+2)*(3+4)/(12-5)
 with no monadic operators like in-5 or +5 followed by key
(1+2)*(3+4)/(12-5)
The Expression you have typed in infix form :
(1+2)*(3+4)/(12-5)
The Equivalent Postfix Expression is :
12+34+*125-/
The Equivalent Prefix Expression is :
/*+12+34-125
```

Sample Output (JUnit)



```
Finished after 0.054 seconds
Runs: 2/2    Errors: 0    Failures: 0
Junit_Test [Runner: JUnit 5] (0.000 s)
    testInToPrefixConvert() (0.000 s)
    testInToPostConvert() (0.000 s)
```

——

Handin

After completing your assignment, keep each part of the assignment into different folders (PI, PII, PIII and PIV) and upload your submission to the Moodle course website as a single ZIP file. You may complete Parts I and II on a digital document (e.g., Word or PDF) or handwritten on paper. If you complete it on paper, you can either submit the original or take a clear picture of your work and include it in the document before submission.
——