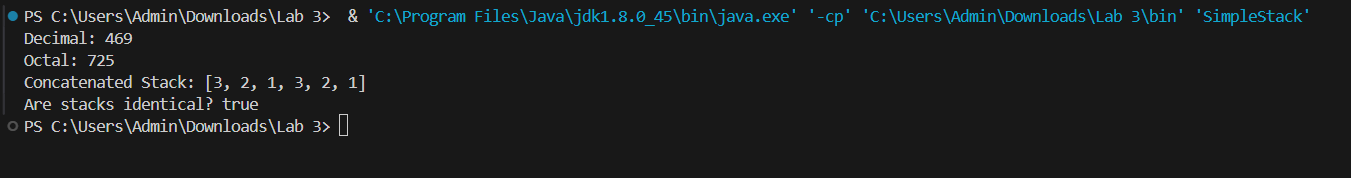
LAB 3 REPORT

Problem 1:

A screen shot of a computer screen

Description automatically generated



In this sample we use 469 as an input decimal number and have the output is 725 as an octal number,

Concatenated Stack:

* Two separate stacks were combined into one.
* The elements from both original stacks were merged to form a single stack.
* In the concatenated stack, the elements from the first stack appeared first, followed by the elements from the second stack

Stack Identical Check:

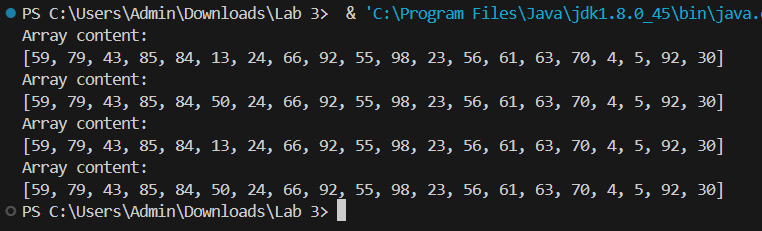
* The contents of two stacks were compared to ascertain whether they are identical.
* The result indicates that the contents of the two stacks are indeed identical, as the output is "true".

In summary, the operations conducted on the provided data encompass decimal to octal conversion, stack concatenation, and stack content comparison. The detailed explanation of the concatenated stack highlights the process and the resulting sequence of elements after the concatenation.

Problem 2:

A screen shot of a computer screen

Description automatically generated



The provided output demonstrates the functionality of the `SpecialArray` class, which encapsulates an array with 20 random integer values. The class offers methods to update the values of the array, undo the updates, redo the updates, and display the array content.

* + Initial Array Content:
    - Upon instantiation of the `SpecialArray` object, the array is initialized with 20 random integer values ranging from 0 to 99.
    - The initial content of the array is displayed, showing the 20 random integer values.
  + Update Value:
    - An update is performed on the array at position 5, changing the value to 50.
    - After the update, the array content is displayed, revealing the modification made to the array.
  + Undo Update:
    - The last update is undone, restoring the array to its previous state.
    - The array content is displayed again to confirm the restoration.
  + Redo Update:
    - The previously undone update is redone, reintroducing the modification to the array.
    - Once again, the array content is displayed, showing the array with the updated value restored.

Problem 3: QueueApp.java

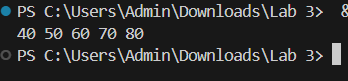
A screen shot of a computer

Description automatically generated

In this Problem, for shortly describe, we use the Algorithm to:

* Creates a Queue object with a maximum size of 5.
* Inserts four elements (10, 20, 30, 40) into the queue.
* Removes three elements (10, 20, 30).
* Inserts four more elements (50, 60, 70, 80). Due to limited size, insertion wraps around, pushing out the oldest element (40).
* Uses a loop to remove and display all remaining elements (40, 50, 60, 70, 80) until the queue becomes empty.

And the output is:



Problem 4: ReverseApp.java

A screen shot of a computer

Description automatically generated

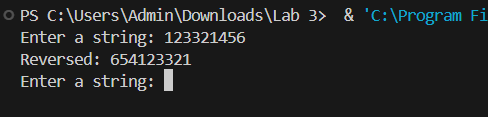
In this problem, we using Stack data structure algorithm to reverse a string:

* Pushing characters onto the stack (push).
* Popping characters from the stack (pop).
* Peeking at the top character without removing it (peek).
* Checking if the stack is empty (isEmpty).

After that, we create **ReverseApp Class** to:

* Prompting the user to enter a string.
* Creating a Reverser object with the input string.
* Calling the doRev method to reverse the string.
* Displaying the reversed string.

The output:



Problem 4: PriorityQApp.java

A screen shot of a computer screen

Description automatically generated

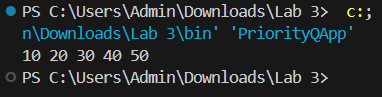
In this problem, there are some changes in the method:

* Modified the **insert** method to insert the new item at the rear of the queue.
* Removed the shifting logic from the **insert** method.
* Modified the **remove** method to find and remove the minimum item from the queue.

A screen shot of a computer program

Description automatically generated

Output:



The efficiency of **PriorityQ** with **QueueApp.java**:

* Both implementations offer different functionalities. **PriorityQ** is designed to find and remove the minimum item efficiently, while **QueueApp.java** is a simple implementation of a queue.
* The efficiency depends on the specific use case. If the requirement is to maintain elements in sorted order or process items based on priority, **PriorityQ** would be more efficient. However, if a basic FIFO (First-In-First-Out) queue suffices, **QueueApp.java** would be more appropriate.

Regarding the use of a priority queue instead of an ordinary one in simulation experiments, it depends on the requirements of the simulation. If the simulation involves prioritizing tasks/events based on certain criteria (e.g., urgency, importance), a priority queue would be suitable. However, if tasks/events are processed in the order they arrive, an ordinary queue would suffice.

Problem 5: Customer Service Center (Queue)

* In this problem, we use two separate queues to efficiently manage the priority between VIP and regular customers

A computer screen shot of code

Description automatically generated

* If the customer is VIP, add them to the VIP queue.
* If the customer is regular, add them to the regular queue.

A screenshot of a computer program

Description automatically generated

* Check if the VIP queue has any customers. If it does, remove and serve the first VIP customer.
* If the VIP queue is empty, serve the first regular customer from the regular queue.

A screen shot of a computer program

Description automatically generated

Test:

A screen shot of a computer code

Description automatically generated

Output:

A black screen with white text

Description automatically generated