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

Assignment 5

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:

- To gain experience working with Priority Queues and Heaps.
- To gain experience working with Hash Tables.

Part I: Implementing a Priority Queue with Array-Based Resizing for Customer Management

A priority queue is a data structure where elements are inserted and removed based on their priority. Higher priority elements are dequeued before lower priority ones. We will implement the priority queue using an array-based approach, manually handling resizing when the capacity is reached. We are using priority queue to **insert** customer details **as they come** but remove the customer with lowest priority upon receiving **dequeue** command.

Input Files:

You will need the following files to get started:

• *Order.java*	*[* ***Need to create]***
• *PriorityQueue.java*	***[Need to complete]***
• *TestPriorityQueue.java*	*[No need to change]*

Steps:

1. **Order class:** You need to implement the Order class from scratch. This class should represent an order entity with the following specifications:
    - **Attributes:**
        - quantity (int): Represents the number of items in the order
        - customerName (String): Stores the name of the customer placing the order
        - priority (int): Indicates the priority level of the order
    - **Implement the Comparable interface:**
        - The Order class should implement the Comparable<Order> interface
        - Override the compareTo method to allow comparison based on priority
    - **Constructor:**
        - Create a constructor that initializes all attributes
    - **Getter methods:**
        - Implement getter methods for all attributes
    - **toString method:**
        - Override the toString method to provide a string representation of the Order
    - **Additional methods:**
        - You may add any additional methods you find necessary or helpful

- **Implement PriorityQueue class**: Create a class named PriorityQueue to represent the priority queue. This class should contain the following methods:

  - **Constructor (PriorityQueue()):**
    - Initializes a new priority queue with a default capacity and sets the size to 0.
    - It initializes the array to store elements using generics.

  - **Ensure Capacity (ensureCapacity()):**
    - Checks if the array has enough capacity to accommodate new elements.
    - If the current size is equal to the array length, doubles the capacity by creating a new array and copying existing elements into it.

  - **Size (size()):**
    - Returns the number of elements currently stored in the priority queue.

  - **Is Empty (isEmpty()):**
    - Checks if the priority queue is empty.
    - Returns true if the size is 0, indicating no elements are present; otherwise, returns false.

  - **Enqueue (**enqueue(E item)**):**
    - Adds a new element to the priority queue.
    - Ensures capacity, then inserts the new element at the appropriate position based on its priority.
    - It shifts existing elements to make space for the new element while maintaining the priority order.

  - **Dequeue (**dequeue()**):**
    - Removes and returns the element with highest priority from priority queue.
    - It removes the root element (the one with the highest priority), replaces it with the last element, and then reorders the elements to maintain the heap property.

  - **Front Value (**frontValue()**):**
    - Returns the element with the highest priority (root of the heap) without removing it.
    - It returns the element at the root of the heap, the first element in the array.

  - **PrintInfo (**printInfo()**):** Prints Queue dump (check screenshot below)

  - **Parent Index (**parent(int i)**):**
    - Computes the index of the parent node given the index of a child node in the heap.
    - It calculates the parent index using the formula (i - 1) / 2 and returns the result.

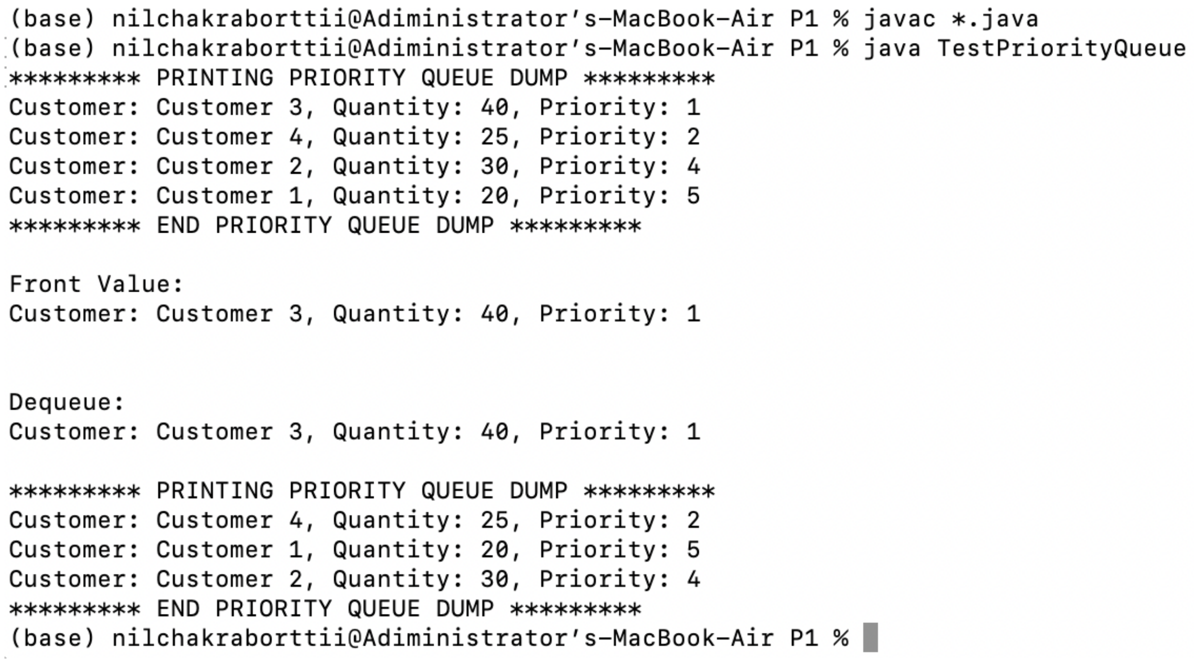
- **Heapify (**heapify(int i)**):**
  - Maintains the heap property by adjusting the position of elements in the array after removal or addition.
  - It compares the element at index i with its children and swaps it with the smallest child, if necessary, recursively applying the process to the affected subtree.

- **Swap (**swap(int i, int j)**):**
  - Swaps the elements at indices i and j in the array.
  - It is used during heapify to exchange elements to maintain the heap property.

- **Test the implementation**: Review the separate class named **TestPriorityQueue** to test the implemented priority queue functionalities by enqueueing, dequeueing, and printing elements.

Note: You should not use java.util package

Sample Output



```
(base) nilchakraborttii@Adiministrator's-MacBook-Air P1 % javac *.java
(base) nilchakraborttii@Adiministrator's-MacBook-Air P1 % java TestPriorityQueue
********* PRINTING PRIORITY QUEUE DUMP *********
Customer: Customer 3, Quantity: 40, Priority: 1
Customer: Customer 4, Quantity: 25, Priority: 2
Customer: Customer 2, Quantity: 30, Priority: 4
Customer: Customer 1, Quantity: 20, Priority: 5
********* END PRIORITY QUEUE DUMP *********

Front Value:
Customer: Customer 3, Quantity: 40, Priority: 1


Dequeue:
Customer: Customer 3, Quantity: 40, Priority: 1

********* PRINTING PRIORITY QUEUE DUMP *********
Customer: Customer 4, Quantity: 25, Priority: 2
Customer: Customer 1, Quantity: 20, Priority: 5
Customer: Customer 2, Quantity: 30, Priority: 4
********* END PRIORITY QUEUE DUMP *********
(base) nilchakraborttii@Adiministrator's-MacBook-Air P1 %
```

Part II: Hashing and Hash Tables

(a) Assume that you have a ten-slot closed hash table (slots numbered 0 through 9). Show the final hash table that would result if you used the hash function h(k) = k mod 10 use and quadratic probing on this list of numbers: 5, 12, 8, 20, 14, 7.

(b) Assume that you have a ten-slot closed hash table (slots numbered 0 through 9). Show the final hash table that would result if you used the following hash functions and double hashing on this list of numbers: 5, 12, 8, 20, 14, 7. The hash functions to be used are as follows:

- Primary hash function: h(k) = k mod 10
- Secondary hash function: h_2(k) =7−(k mod 7)

(c) Using closed hashing, with double hashing to resolve collisions, insert the following keys into a hash table of thirteen slots (the slots are numbered 0 through 12). The hash functions to be used are H1 and H2, defined below. You should show the hash table after all eight keys have been inserted. Be sure to indicate how you are using H1 and H2 to do the hashing.

Function Rev(k) reverses the decimal digits of k, for example, Rev(37) = 73; Rev(7) = 7.
H1(k) = k mod 13.

H2(k) = (Rev(k + 1) mod 11).

Keys: 2, 8, 31, 20, 19, 18, 53, 27.

(d) Assume that you have a ten-slot closed hash table (the slots are numbered 0 through 9). Show the final hash table that would result if you used the hash function h(k) = k mod 10 and pseudo-random probing on this list of numbers: 3, 12, 9, 2, 79, 44. The permutation of offsets to be used by the pseudo-random probing will be: 5, 9, 2, 1, 4, 8, 6, 3, 7. After inserting the record with key value 44, list for each empty slot the probability that it will be the next one filled.

Part III: Hash Table Implementation and Analysis

In this part, you will implement a custom hash table in Java and analyze its performance using different hashing techniques and collision resolution strategies. You will also write a test program to evaluate the hash table's behavior under various configurations. A sample output will be provided to guide your implementation.

Files: You will need the following files to get started:

• KeyValuePair.java	[Completed]
• HashTable.java	**[Need to complete]**
• HashTableTest.java	[**Need to create**]

KeyValuePair.java This class represents a simple key-value pair that can be used to test your hash table implementation. It provides methods to retrieve and modify the key and value, along with implementations of **hashCode()** and **equals()** for proper usage in hash-based data structures.

Tasks:

You need to implement two classes:

- **HashTable.java** A custom implementation of a hash table with support for multiple hashing techniques, collision resolution strategies, and performance metrics.
- **HashTableTest.java** A test program to evaluate the hash table's performance under different configurations.

Hash Table Implementation (HashTable.java)

You need to implement the following methods in the **HashTable** class:

- **Constructors:**
  - **HashTable():** Default constructor that initializes the hash table with default capacity and hash function.
  - **HashTable(int initialCapacity, int hashFunction, int collisionResolution):** Constructor that allows specifying initial capacity, hash function, and collision resolution strategy.

- **put(Key key, Value value)** Inserts a key-value pair into the hash table. Handles collisions using the selected collision resolution strategy. Resizes the table when the load factor exceeds a threshold.

- **get(Key key)** Retrieves the value associated with the given key. Returns **null** if the key does not exist in the hash table.

- **delete(Key key)** Deletes the key-value pair from the hash table by marking the slot as deleted (using tombstones). Ensures proper handling of probing sequences.

- **h(Key key)** Implements three different hashing techniques:
  - **Division Method:** h(key) = |key.hashCode()| % M
  - **Multiplication Method:** h(key) = (int)(M * ((key.hashCode() * A) % 1)), where A = 0.618033988749895
  - **Universal Hashing:** h(key) = ((a * |key.hashCode()| + b) % p) % M, where p is a large prime number, and a and b are random integers.

- **p(Key key, int i):** Implements three collision resolution strategies:
  Linear Probing: p(key, i) = i
  Quadratic Probing: p(key, i) = i * i
  Double Hashing: p(key, i) = i * h2(key), where **h2(key)** is a secondary hash.

- **resize(int newCapacity)**: Resizes the hash table when the load factor exceeds a threshold. Ensures that the new capacity is a prime number for better performance (especially for quadratic probing).

- **Performance Metrics Methods:**
  - **getCollisions()**: Returns the total number of collisions encountered during operations.
  - **getAverageProbeLength()**: Returns the average probe length for all operations.
  - **getLoadFactor()**: Returns the current load factor of the hash table.
  - **resetMetrics()**: Resets all performance metrics to zero.

- **Utility Methods:**
  - nextPrime(int n): Finds the next prime number greater than or equal to n.
  - isPrime(int n): Checks if a number is prime.
  - Secondary Hash Function (h2(Key key)): Used for double hashing to calculate probe step size.

Hash Table Test Program (HashTableTest.java)

You need to complete the following methods in the **HashTableTest** class:

- **testHashTable()** Tests different combinations of hashing techniques and collision resolution strategies by:
    - Creating hash tables with varying configurations.
    - Inserting a large number of random keys into each table.
    - Searching for both existing and non-existing keys multiple times.
    - Measuring and reporting metrics such as insertion time, search time, number of collisions, average probe length, and load factor.

Helper Methods:

- **generateRandomKeys(int count)**: Generates an array of random integer keys for insertion into the hash table.
- **generateNonExistingKeys(int count, Integer[] existingKeys)**: Generates keys that are guaranteed not to exist in the hash table for testing non-existing searches.

Completed Helper Methods:
private static String getHashFunctionName(int hashFunction)
private static String getCollisionResolutionName(int collisionResolution)

Sample Output:

```
(base) nilchakraborttii@ P3 % java HashTableTest
Warming up JVM...
Warm-up complete.
Testing with DIVISION_HASH and LINEAR_PROBING
Insertion time: 47 ms
Existing key search time: 32 ms
Non-existing key search time: 64 ms
Number of collisions: 201447
Average probe length: 0.63657
Load factor: 0.499985
Found 1000000 existing keys out of 1000000 searches
Confirmed 1000000 non-existing keys out of 1000000 searches

Testing with DIVISION_HASH and QUADRATIC_PROBING
Insertion time: 20 ms
Existing key search time: 49 ms
Non-existing key search time: 56 ms
Number of collisions: 66172
Average probe length: 0.44642
Load factor: 0.49959780767712697
Found 1000000 existing keys out of 1000000 searches
```

```
Confirmed 1000000 non-existing keys out of 1000000 searches

Testing with DIVISION_HASH and DOUBLE_HASHING
Insertion time: 36 ms
Existing key search time: 72 ms
Non-existing key search time: 95 ms
Number of collisions: 116045
Average probe length: 0.4547409090909091
Load factor: 0.499995
Found 1000000 existing keys out of 1000000 searches
Confirmed 1000000 non-existing keys out of 1000000 searches

Testing with MULTIPLICATION_HASH and LINEAR_PROBING
Insertion time: 30 ms
Existing key search time: 73 ms
Non-existing key search time: 96 ms
Number of collisions: 210948
Average probe length: 0.6465072727272727
Load factor: 0.499985
Found 1000000 existing keys out of 1000000 searches
Confirmed 1000000 non-existing keys out of 1000000 searches

Testing with MULTIPLICATION_HASH and QUADRATIC_PROBING
Insertion time: 12 ms
Existing key search time: 66 ms
Non-existing key search time: 94 ms
Number of collisions: 66751
Average probe length: 0.45421
Load factor: 0.49961779238882253
Found 1000000 existing keys out of 1000000 searches
Confirmed 1000000 non-existing keys out of 1000000 searches

Testing with MULTIPLICATION_HASH and DOUBLE_HASHING
Insertion time: 22 ms
Existing key search time: 31 ms
Non-existing key search time: 63 ms
Number of collisions: 185019
Average probe length: 0.45112636363636366
Load factor: 0.49999
Found 1000000 existing keys out of 1000000 searches
Confirmed 1000000 non-existing keys out of 1000000 searches

Testing with UNIVERSAL_HASH and LINEAR_PROBING
Insertion time: 24 ms
Existing key search time: 61 ms
Non-existing key search time: 91 ms
Number of collisions: 203866
Average probe length: 1.3657105308268098
Load factor: 0.5
Found 250020 existing keys out of 1000000 searches
```

```
Confirmed 1000000 non-existing keys out of 1000000 searches

Testing with UNIVERSAL_HASH and QUADRATIC_PROBING
Insertion time: 23 ms
Existing key search time: 48 ms
Non-existing key search time: 58 ms
Number of collisions: 66898
Average probe length: 0.6938514133244392
Load factor: 0.4996078000329748
Found 499650 existing keys out of 1000000 searches
Confirmed 1000000 non-existing keys out of 1000000 searches

Testing with UNIVERSAL_HASH and DOUBLE_HASHING
Insertion time: 20 ms
Existing key search time: 70 ms
Non-existing key search time: 69 ms
Number of collisions: 133572
Average probe length: 0.8773520756549813
Load factor: 0.499995
Found 250010 existing keys out of 1000000 searches
Confirmed 1000000 non-existing keys out of 1000000 searches
```

Key Points:

- Ensure that your implementation handles null keys appropriately by throwing exceptions or clearly defined behavior.
- Use absolute values where necessary to handle negative integers correctly in hashing functions.
- Carefully manage resizing logic, especially for quadratic probing (ensure new capacity is prime).
- Document your code thoroughly with comments explaining each method's purpose and functionality.

Note: You should not use any Java library classes for hash tables (e.g., HashMap)

———

Submission Guidelines:

- Complete methods as instructed. Add JavaDoc in your code.
- Upon the completion of your lab, arrange each section into individual folders (PI, PII, PIII), then compress these folders into a single Zip file. Upload the consolidated submission to the Moodle course website.

———