TRINITY COLLEGE
DEPARTMENT OF COMPUTER SCIENCE

CPSC 215: Data Structures and Algorithms
Instructor: Dr. Chandranil Chakraborttii
Spring 2025

Assignment 6

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.
- To learn and implement Huffman Encoding for data compression.

Resources:

Getting Started

- Open Eclipse and create a Java Project (File -> New -> Java Project)
- Give a project name (For example: assignment_6_CPSC_215)
- Go to Project -> Right Click -> New Package
- Give a package name (For example: assignment_6_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.

Part I: Huffman Encoding

You need to solve this problem using a pen and paper.

e	r	s	t	n	l	z	x
34	22	24	28	15	10	9	8

Frequency in an average sample of size 150 letters

Given the distribution of frequency in a sample size of 150 characters, build the binary tree along with the prefix codes for each of the characters. How many bits are saved compared to ASCII representation for each character for the entire string of 150 characters?
[Assume 8 bits to store each character in ASCII]

Encode the following using the built trees:

- “next”
- “stern”
- “nertzrents”

Part II: Implementing Double Hashing in a Hash Table with Generics

In this exercise, you are required implement a hash table using double hashing as the collision resolution technique.

Input Files:

- **Entry.java**: Stores key-value pairs in the hash table. ***[No need to change]***
- **HashTable.java:** Implements a hash table using double hashing for collision resolution. Handles insertion, retrieval, and removal of key-value pairs. ***[Need to change]***
- **Test.java:** Tests the functionality of the HashTable class by inserting, retrieving, and removing key-value pairs. ***[No need to change]***

Tasks:

- **Review** completed class **Entry. Java.**
- **Implement** the methods in **HashTable class. The class** represents a hash table data structure. It manages the insertion, retrieval, and removal of key-value pairs using double hashing to handle collisions. It has fields for size, capacity, and an array of **Entry** objects to store key-value pairs. Methods include:
    - **hashFunction1:** Calculates the first hash value using modulo division. h1(key) = key % capacity **(Completed)**
    - **hashFunction2:** Calculates the second hash value using the formula **h2(key)= 7 – (abs(key) % 7).**
    - **doubleHash:** Combines the two hash values to resolve collisions. **(h1 + i*h2)** where i is a parameter representing the number of attempts made to resolve collisions using double hashing.
    - **put:** Inserts a key-value pair into the hash table and reports if table if full.
    - **get:** Retrieves value associated with a key from hash table, returns **null** if not found.
    - **remove:** Removes a key-value pair from the hash table.
    - **print**: Prints the contents of the hash table.
- **Verify and test** your program for correctness using **Test**.java.

Note: You need to use hashCode() for **hashfuntion2** on the key as shown in **hashfunction1**. The method is implemented in the Object class, and it returns an integer hash code value for the object. This hash code is typically based on the contents of the object and is designed to ensure that equal objects produce the same hash code.

Expected Output

```
(base) nilchakraborttii@Adiministrator's-MacBook-Air P3 % java Test
Value for key 'John': 25
Value for key 'Alice': 30
Value for key 'Bob': 35
Value for key 'Alice' after removal: null
Key: Bob, Value: 35
Key: John, Value: 25
```

Part III: Implementing Run-Length Encoding with Linked List

The objective of this exercise is to implement the Run-Length Encoding algorithm and Linked List data structure in Java.

Input Files:

- **Node.java** *[No need to change]*
    - Represents a node in the linked list. Each node contains an element of data and a reference to the next node in the sequence.
- **Pair.java [***No need to change]*
    - Represents a pair of elements, typically used to store key-value pairs. Used to represent an element and its count in the encoded data.

- **LinkedList.java** *[No Need to change]*
    - Implements a basic linked list data structure to store a sequence of elements.
- **RunLengthEncoder.java** ***[Need to change]***
    - Implements the Run-Length Encoding algorithm.
- **Tester.java** *[No need to change]*
    - Class for testing our implemented methods.

Tasks:

- **Review** completed classes **Node. Java, LinkedList.java** and **Pair.java**
- **Review** the methods in **LinkedList class** to create and manage a linked list of elements.
    - **add(E data):** Adds a new node with the provided data to the end of the linked list. Updates the tail reference if needed and increments the size.
    - **getHead():** Retrieves the head node of the linked list. size(): Returns the current size of the linked list.
    - **getNode(int index):** Retrieves the node at the specified index in the linked list. Throws an exception if the index is out of bounds.
    - **insertAfter(Node<E> prevNode, E data):** Inserts a new node with the provided data after the specified previous node. Throws an exception if the previous node is null. Updates the tail reference if necessary and increments the size.
- **Implement** the methods in **RunLengthEncoder** class to perform Run-Length Encoding on a given sequence of data.
    - **encode(LinkedList<E> list):** Performs run-length encoding on the input linked list, returning a new linked list containing pairs of elements and their counts.
    - **decode(LinkedList<Pair<E, Integer>> encodedList):** Decodes the input encoded list, reconstructing the original list by repeating each element according to its count specified in the pairs. Returns the decoded list.
- **Test** implementation by encoding and decoding a sample sequence using **Tester.java.**
    - Ensure that the encoding and decoding processes work correctly according to the Run-Length Encoding algorithm.
    - If needed, you may implement any additional methods.

Expected Output

```
(base) nilchakraborttii@Adiministrator's-MacBook-Air P2 % javac *.java
(base) nilchakraborttii@Adiministrator's-MacBook-Air P2 % java Tester
Input String: AAABBBCCCCXXYYDDDD
Encoded Result:
A3B3C4X2Y2D4
Decoded Result: AAABBBCCCCXXYYDDDD
```

Part IV: Sorting and Searching using Linked Lists

In this part, you will be sorting linked lists using three sorting techniques discussed in class (Bubble Sort, Selection Sort, Insertion Sort) and search for an element within the linked list

using two search techniques (linear search, binary search). As typical, we will be using generics for this assignment. You may consult class lectures for this assignment.

Input Files:

- **Node.java** ***[No need to change]***
    - Represents a single node in a linked list with fields for data and a reference to the next node.
- **LinkedList.java** ***[No Need to complete]***
    - Represents a linked list data structure with methods for adding elements, displaying elements, and sorting.
    - Implements sorting algorithms directly on the Linked list.
    - Implements searching algorithms directly on the Linked list.
- **SortTest.java** ***[No need to change]***
    - Main class for testing sorting algorithms implemented in the LinkedList class.
    - Creates a linked list, adds elements, applies sorting algorithms, and displays original and sorted lists for verification.

Note: You **MUST** use linked lists to complete this program. Use of any other data structures (such as Arrays, Stacks, Queues, etc. are not allowed)

Tasks:

- **Review** completed class **Node.java**
- **Implement** methods in **LinkedList class** to create and manage a linked list of elements for adding elements, displaying elements, and sorting (bubble, selection, and insertion).
    - **LinkedList():** Constructor initializes the linked list with head set to null and size set to 0.
    - **add(E data):** Adds a new element with the specified data to the end of the linked list. If the list is empty, it creates a new node and sets it as the head. Otherwise, it traverses the list to find the last node and appends the new node to it.
    - **display():** Displays the elements of the linked list by traversing through each node and printing its data.
    - **bubbleSort():** Sorts the elements of the linked list using the bubble sort algorithm. It repeatedly swaps adjacent elements if they are in the wrong order until the list is sorted.
    - **selectionSort():** Sorts the elements of the linked list using the selection sort algorithm. It repeatedly selects the minimum element from the unsorted portion of the list and swaps it with the first unsorted element.
    - **insertionSort():** Sorts the elements of the linked list using the insertion sort algorithm. It iteratively inserts each element into its correct position in the sorted part of the list.
    - **insert(Node<E> sorted, Node<E> newNode):** Helper method for insertion sort. Inserts the given new node into the sorted part of the list while maintaining the sorted order.
    - **linearSearch(E key):** Performs a linear search through the linked list to find the specified key. It iterates through each element in the list until it finds a match with the key. If the key is found, it returns true; otherwise, it returns false.

- **binarySearch(E key):** Performs a binary search through the sorted linked list to find the specified key. It uses the binary search algorithm by maintaining two pointers (left and right) that represent the range of elements to search within. It calculates the middle element (mid) and compares it with the key. If the key is found, it returns true; otherwise, it adjusts the pointers based on whether the key is greater or less than the middle element and continues the search until the pointers converge or the key is found. If the key is not found, it returns false.
- **getNode(int index):** Retrieves the node at the specified index in the linked list. It traverses the list starting from the head node and advances index times to reach the desired node. If the index is out of bounds, it throws an IndexOutOfBoundsException.
- **indexOf(Node<E> node):** Retrieves the index of the specified node in the linked list. It iterates through the list starting from the head node and compares each node with the specified node until a match is found. If the node is found, it returns its index; otherwise, it returns -1.

- **Verify and test** your program for correctness using **SortTest**.java.

Expected Output

```
(base) nilchakraborttii@Adiministrator's-MacBook-Air P3 % javac *.java
Note: LinkedList.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
(base) nilchakraborttii@Adiministrator's-MacBook-Air P3 % java SortTest
Original List:
Hannah Adam Zeek Mike Emily Chloe Tiffany Nikita

Bubble Sort:
Adam Chloe Emily Hannah Mike Nikita Tiffany Zeek

Selection Sort:
Adam Chloe Emily Hannah Mike Nikita Tiffany Zeek

Insertion Sort:
Adam Chloe Emily Hannah Mike Nikita Tiffany Zeek

Linear search for key Chloe: true
Linear search for key Nil: false

Binary search for key Hannah: true
Binary search for key Biden: false
```

Submission Guidelines:

- Use of java.util package or collections is NOT allowed
- Complete methods as instructed. Add JavaDoc in your code.
- Upon the completion of your lab, arrange each section into individual folders (PI, PII, PIII, PIV), then compress these folders into a single Zip file. Upload the consolidated submission to the Moodle course website.
