University of Calgary
Faculty of Science
Department of Computer Science

UNIVERSITY OF CALGARY

CPSC 331 - Data Structures, Algorithms and their Analysis
Assignment 3
Binary Search Tree, Hash Table, Heap, and Graph
Winter 2024

Problem 1: Binary Search Tree

In this problem, we attempt to verify the time complexity of some of the basic operations on binary search tree in relation to the height of the tree. To achieve this, you need to write a JAVA program to implement the tasks listed below. Assume first that the function $g(n)$ returns a random integer r_i between 0 and n, and $g_e(n)$ returns an even random integer e_i between 0 and n.

1. Call the function $g(n)$ and let r_i be the random number generated. Starting with an empty BST T_0, insert r_i in T_0, if it doesn't already exist in the tree. Repeat this process of generating a random number and insert it into T_0 until m distinct nodes are inserted in T_0. Let $100 \leq m \leq 200$. (Example: Let's say $n = 300$, and $m = 100$. We call $g(300)$ to generate random integers between 0 and 300. We'll insert each generated integer into the BST if it doesn't already exist in the tree. After enough calls to $g(300)$, we should have a BST containing 100 unique random integers from the range 0 to 300.

2. Remove all the odd values from T_0. This will give a new tree T_1 having only the even numbers generated above. For example, After removing all the odd numbers from the BST $T_0 = [1, 3, 8, 9, 7, 2, 12, 6]$, the resulting tree T_1 is modified to contain only even numbers $[8, 2, 12, 6]$.

3. Repeat the steps below to insert k new random integers not existing in the tree T_1 where $50 \leq k \leq 150$:

    (a) Let r_i be the random integer generated by a call to the function $g(n)$

    (b) If r_i is not in the tree T_1 then do the following:

        i. Find the height h_i of the tree T_1

        ii. Insert r_i into the tree T_1

    iii. Measure the time t_i needed for the insertion procedure[1]
    iv. Store the values (h_i, t_i) in a suitable data structure.

4. Plot the values (h_i) against the (t_i) ensuring the h_i is on y axis and t_i is on the x axis. There are several libraries in Java to make simple plotting. You may use Xchart library. After making the plot, you may save the figure and added to a pdf file including other explanations (see below).

5. Use the plot above to compare the results obtained by this experiment and the corresponding the big-Oh notation for inserting an element in a BST as discussed in the class. Justify your answer. (you need to submit a pdf file not exceeding one page, and containing the plot above)

Problem 2: Designing a Hash Table for School Database Management

You are responsible for creating the school's database. For this, you will create a hash table with chaining in which the keys are IDs and the values are names. Your hash table should have the following functionality:

1. Search: Check whether a student is in the database.
2. Retrieve: Return the corresponding name.
3. Insert: Insert/update the student's information in the database.
4. Delete: Remove the student from the database.
5. ToString: Print the whole database.

To implement these, you will also need a hash function [method: hashValue]. More detail on all these things is in the implementation details.

2.1 Task

(a) Do the following steps in order:

1. `Insert` the students with the following information: "20500120:Bob", "20700200:Alice", "30100230:Cathy", and "20200156:Ali".
2. Call the `ToString` function to print the entire database.
3. `Insert` "20500120:Bobby" to update Bob's name.
4. Call the `Search` function for the ID "20500120"

[1]Details about this will be discussed in the tutorials.

5. `Retrieve` the value associated with the ID ”20700200”.
6. `Remove` the student with the ID ”20700200”.
7. Again call the `Remove` function with the ID ”20700200”. (Error Handling: Should print an appropriate message)
8. Now call the `Retrieve` function with the ID ”20700200”. (Error Handling: Should print an appropriate message)
9. Call the `ToString` function again to print the updated database.

2.2 Implementation Details

2.2.1 Hash Tables with chaining

You've learned about hash tables with chaining during lectures. In this kind of hash table, keys are passed through a hash function which produces indices for an array. The key along with the value then placed in a linked list at this index of the array. See the below diagram for an illustration of a hash table with chaining:

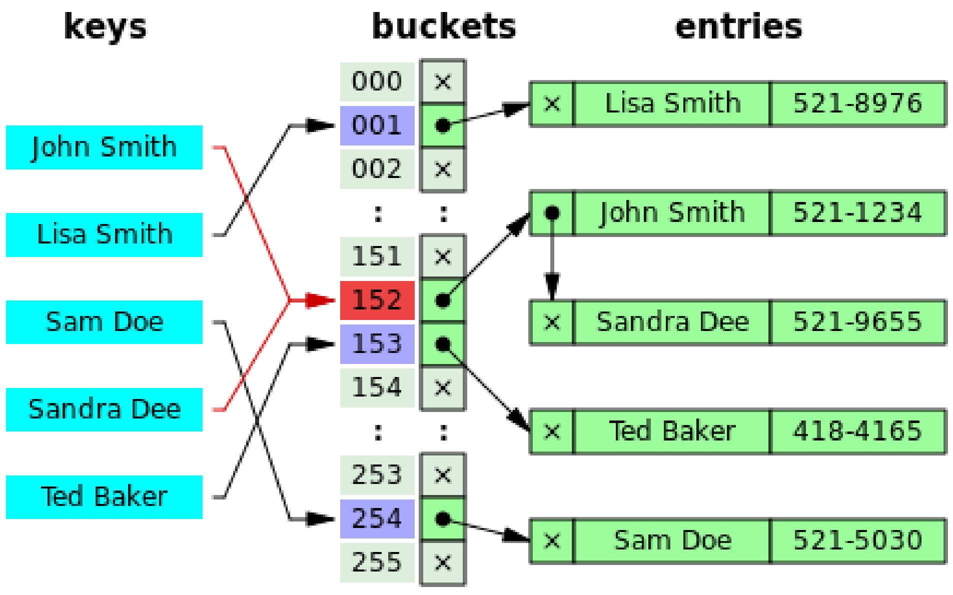

Figure 1: An example of a hash table with chaining.

The keys of the hash table will be the ID of the students and will be of type Integer. The values are the names and are of type int. The size of your array will be the constant LEN, which will be set to 8.

2.2.2 Hash Function Implementation

Your hash function will take an ID and return the index as modulo the size of the array. For example, 30200189 % 8 returns 5.

2.2.3 Linked List Implementation

In this assignment, you can use the native Java implementation **java.util.LinkedList.** The type of object stored in your linked list will be Student. To store a student's info, you should create a class named Student which contains the student ID and name as attributes. Documentation for Java LinkedList can be found at:

docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html

Geeks for Geeks has a fairly comprehensive page about Java LinkedList here:

geeksforgeeks.org/linked-list-in-java/

Below are a few LinkedList methods you are likely to use:

1. Initializing: LinkedList$\langle ObjectType \rangle$ l = new LinkedList$\langle ObjectType \rangle$
2. Append to the end of the list: *l.add(element)*
3. Get element at specified index: *l.get(index)*
4. Remove by index: *l.remove(index)*
5. Length: *l.size()*

Note: Java LinkedList nodes don't seem to have **.next** pointers as would usually be expected so if you need to iterate through the nodes of a LinkedList, you can do so in a similar fashion to an array using the **l.get(index)** method (I believe doing so actually increases the time complexity but don't worry about it)[2] .

2.2.4 Method Preconditions and Postconditions

The following are the main preconditions and the postconditions of the operations involved.

1. **HashValue(int ID)**
   Precondition: ID is an integer.
   Postcondition: ID modulo LEN is returned.

[2] *You may also choose to use the LinkedList.listIterator() method to iterate through the nodes.*

2. **Search(int ID)**
   Precondition: ID is an integer.
   Postcondition: If the hash table contains a student by this ID, then return true. Otherwise, return false.

3. **Retrieve(int ID)**
   Precondition: ID is an integer.
   Postcondition: If the hash table contains a student with this ID, return the corresponding name. Otherwise, print a message indicating that no student with this ID was found in the hash table.

4. **Insert(int ID, String name)**
   Precondition: ID is an integer and name is a string composed of alphanumeric characters
   Postcondition: If the hash table does not contain a Student of this name, then a Student with attributes ID and name is added to the appropriate linked list in the hash table. If a Student of this name is already in the hash table, then this Student has their name updated to the inputted name.

5. **Delete(int ID)**
   Precondition: ID is an integer.
   Postcondition: If the hash table contains a *Student* with this ID, this Student is removed from the hash table. Otherwise, a message is printed indicating that no Student with this name was found in the hash table.

6. **toString()**
   Precondition: *arr* is an array of LinkedList of Student.
   Postcondition: A string is returned consisting of

   - One line per entry of the array, including empty linked lists (it is acceptable to have an extra newline at the end)
   - Each line takes the form

     ArrayIndex: [FirstStudentID:FirstStudentName, SecondStudentID:SecondStudentName, . . . , LastStudentID:LastStudentName]

     Example: If 20500120:Bob, 20700200:Alice, 30100230:Cathy, 20200156:Ali are inserted (in that order), *toString()* produces the following:

```
0:  [20500120:Bob, 20700200:Alice]
1:  [ ]
2:  [ ]
3:  [ ]
4:  [20200156:Ali]
5:  [ ]
6:  [30100230:Cathy]
7:  [ ]
```

Problem 3: Heap Sort

In this question, your task is to implement Heap Sort to sort an array, counting the swaps required for both heap creation and sorting using two methods: heapify and adding elements one by one. With the heapify method, you'll build a max heap from the array. Similarly, with the adding elements one by one method, you'll construct a max heap by inserting elements individually and maintain the heap property. After creating each heap, heap sort will be performed. Note that you should only count the number of swaps, excluding removals in the Heap Sort.

3.1 Tasks

(a) Implement the Heap Sort algorithm and the functions required for creating a max-heap for both methods. For the implementation of a heap, you are allowed to use arrays or linked lists.

(b) For the following inputs:

- A random array with a size of 1000.
- A sorted array $[0, ..., 999]$ with a size of 1000.

1. Count the number of swaps required for creating only a max-heap for both methods.
2. Get the sum of the number of swaps in both heap creation and sorting steps for both methods.
3. Upload a PDF file explaining and justifying the average-case and worst-case time complexity of creating a max-heap and Heap Sort algorithm using each of the methods based on the results in (a) and (b).

3.2 Input and Output Format

Input

The input consists of the following lines:

```
{1, 2, 3, 4}
```

An array with type int

Output

Format each output line as follows:
For part 1 in Section (b):

```
3
4
```

For part 2 in Section (b):

```
5
5
```

The first and second lines show the number of swaps for methods 1 and 2, respectively. For the given array, the process has been shown for each method step by step. Here, 'S' denotes 'Swap', 'A' denotes 'Add', and 'R' refers to 'Removing the max element and replacing it with the last element.'

Method 1: Heapify

Build max heap:

S: [1, 2, 3, 4] → [1, 4, 2, 3]
S: [1, 4, 2, 3] → [4, 1, 3, 2]
S: [4, 1, 3, 2] → [4, 2, 3, 1]

HeapSort:

R: [4, 2, 3, 1] → [1, 2, 3, 4] → [1, 2, 3]
S: [1, 2, 3] → [3, 2, 1]
R: [3, 2, 1] → [1, 2, 3] → [1, 2]
S: [1, 2] → [2, 1]
R: [2, 1] → [1, 2] → [1]
R: [1] → [ ]

Total number of swaps: $3 + 2 = 5$

Method 2: OneByOne:

Build max heap:

A: [ ] → [1]
A: [1] → [1, 2]
S: [1, 2] → [2, 1]
A: [2, 1] → [2, 1, 3]

S: [2, 1, 3] $\rightarrow$ [3, 1, 2]
A: [3, 1, 2] $\rightarrow$ [3, 1, 2, 4]
S: [3, 1, 2, 4] $\rightarrow$ [3, 4, 2, 1]
S: [3, 4, 2, 1] $\rightarrow$ [4, 3, 2, 1]

HeapSort:

R: [4, 3, 2, 1] $\rightarrow$ [1, 3, 2, 4] $\rightarrow$ [1, 3, 2]
S: [1, 3, 2] $\rightarrow$ [3, 1, 2]
R: [3, 1, 2] $\rightarrow$ [2, 1, 3] $\rightarrow$ [2, 1]
R: [2, 1] $\rightarrow$ [1, 2] $\rightarrow$ [1]
R: [1] $\rightarrow$ [ ]

Total number of swaps: $4 + 1 = 5$

Problem 4: Graph - Delivery Routes Optimization with Dijkstra's Algorithm

You are a software engineer tasked with designing a delivery route optimization system for a logistics company. The company operates a central warehouse and delivers packages to various locations within a city. Due to the city's complex road network and varying distances between locations, it becomes challenging to determine the most efficient routes for deliveries.

Your task is to develop a system that optimizes delivery routes by finding the shortest paths from the central warehouse to all delivery locations. The system must consider the distances between locations and identify the shortest routes to minimize delivery times. To achieve this, you will implement Dijkstra's algorithm, a renowned graph traversal algorithm known for finding the shortest paths in weighted graphs.

By implementing this system, the logistics company aims to enhance its delivery operations by reducing travel times, optimizing resource utilization, and improving customer satisfaction. The system's efficient route optimization capabilities will enable the company to streamline its delivery processes, handle increasing delivery demands, and maintain a competitive edge in the logistics industry.

4.1 Tasks

(a) Implement a data structure to represent the city as a directed weighted graph. Each node represents a warehouse or delivery location, and edges represent roads connecting them. The weight of an edge should be the distance between the connected locations.

(b) Implement Dijkstra's algorithm to find the shortest delivery routes from the central warehouse to all delivery locations based on distance. Ensure proper initialization of data structures and handle edge cases appropriately.

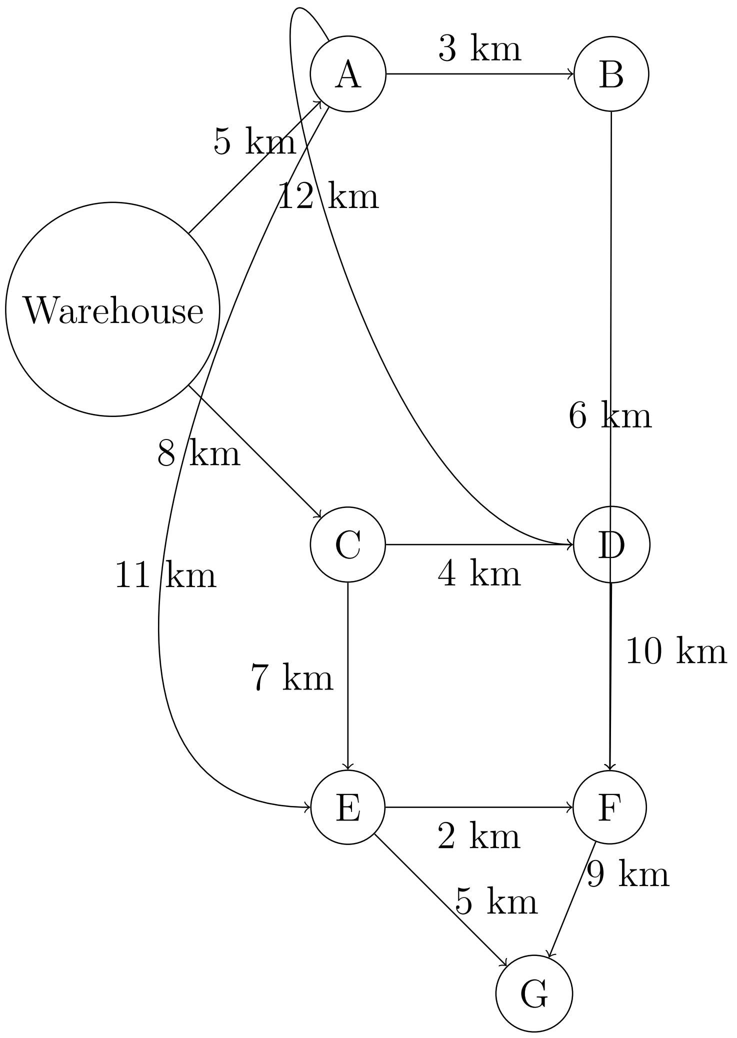

Figure 2: A sample delivery route optimization problem. The warehouse is represented by the node labelled as Warehouse, and the other nodes represent delivery locations. The edges between nodes represent possible delivery routes, with the distance labeled in kilometers.

(c) Define specified input format for specifying the number of warehouses, delivery locations, roads, and their distances. Design specified structured output format to display the shortest delivery routes and their corresponding distances, indicating ties between routes if any.

(d) Ensure that your code can handle unexpected input formats without crashing or producing incorrect results. Implement error-checking mechanisms to validate input data and provide clear error messages for any inconsistencies or invalid data encountered

(e) After implementing Dijkstra's algorithm and optimizing delivery routes, provide a brief analysis of the algorithm's time and space complexities in the context of the given delivery routes optimization problem. Consider factors such as the number of vertices, edges, and the efficiency of the chosen data structures and algorithmic techniques. Discuss the scalability of the solution and potential strategies for further optimization, if applicable.

4.2 Input and Output Format

Input

The input consists of the following lines:

```
Number of warehouses (always 1)
Number of delivery locations
Number of roads connecting locations (R)
R lines, each containing three integers: starting location, destination
    location, and distance
```

Denote the central warehouse location as 0 and delivery locations start from 1

Output

Format each output line as follows:

```
Delivery Location <location number> - Shortest Route: <route>, Distance: <
    distance>
```

Replace **location number** with the delivery location's identifier. Replace **route** with the sequence of nodes representing the shortest route, separated by arrows (−>). Replace **distance** with the distance value of the shortest route. If multiple shortest routes exist for a delivery location, list all possible routes, separated by **or**. If a delivery location is unreachable from the central warehouse, indicate that no route exists and set the distance to **Infinity**.

4.3 Sample Test Cases

Test Case: 1

Input

```
1
3
5
0 1 10
0 2 15
1 2 5
1 3 20
2 3 12
```

Output

```
Delivery Location 1 - Shortest Route: 0 -> 1, Distance: 10
Delivery Location 2 - Shortest Route: 0 -> 2, Distance: 15
Delivery Location 3 - Shortest Route: 0 -> 1 -> 2 -> 3 (or 0 -> 2 -> 3)
   Distance: 27
```

Test Case: 2

Input

```
1
4
5
0 1 15
0 2 10
1 2 5
1 3 20
2 3 12
```

Output

```
Delivery Location 1 - Shortest Route: 0 -> 1, Distance: 15
Delivery Location 2 - Shortest Route: 0 -> 2, Distance: 10
Delivery Location 3 - Shortest Route: 0 -> 2 -> 3, Distance: 22
Delivery Location 4 - Shortest Route: No route exists, Distance: Infinity
   (Location 4 is unreachable from the central warehouse)
```

Test Case: 3

Input

```
1
3
3
0 1 8
1 2 4
0 2 6
```

Output

```
Delivery Location 1 - Shortest Route: 0 -> 1, Distance: 8
Delivery Location 2 - Shortest Route: 0 -> 2, Distance: 6
Delivery Location 3 - Shortest Route: No route exists, Distance: Infinity
    (Location 3 is unreachable from the central warehouse
```

Grading

The assignment is graded out of a total of 100 points, with each of the four problems accounting for 25 points. Problem 1 focuses on Binary Search Tree (BST), Problem 2 on Hash, Problem 3 on Heap, and Problem 4 on Graph. Grading will assess the correctness and efficiency of the implemented algorithms, adherence to the specified problem requirements, clarity and organization of the code, as well as any additional criteria outlined in the specific problem.

Deadline

The assignment is due by April 1^{st} at 11:59 PM, after that the deduction policy of 20% daily (part of the day is considered as a full day) will be applied. The first 15 minutes of the penalty day is a grace period and will not result in deduction.

Submission

To submit your assignment, please follow these steps:

1. Create a new folder named student name+ID (example: Name = John Doe, Student ID = 12345678, FILENAME = `John_Doe_12345678`) on your computer.

2. Place all the files and directories listed below into this folder. Ensure that you follow the same naming convention and have one file named Main.java for each problem, from which we can obtain all the required outputs.

- `PDF/ (directory)`
  - BST.pdf
  - Heap.pdf
  - Graph.pdf
- `source_code/`
  - `BST/` (directory)
    - `Main.java` (file)
    - ... (additional files)
  - `HashTable/` (directory)
    - `Main.java` (file)
    - ... (additional files)
  - `Heap/` (directory)
    - `Main.java` (file)
    - ... (additional files)
  - `Graph/` (directory)
    - `Main.java` (file)
    - ... (additional files)

3. Once all files and directories are in place, compress the folder into a ZIP file like `John_Doe_12345678.zip`.
4. Ensure that all necessary files and directories are included in the ZIP file.
5. Submit the ZIP file on D2L.

Please ensure that you follow these instructions accurately to complete the submission of your assignment.