CS 251 Project 4 Handout

Department of Computer Science
Purdue University

July 2025

1 ScapeGoat Tree with Range Queries

1.1 Problem Statement

In this task, you will implement a ScapeGoat Tree from scratch. Your code should define a class named `ScapeGoatTree<K, V>` that implements `ScapeGoatTreeInterface<K, V>` and correctly manages a dynamic binary search tree structure with self-balancing operations. Additionally, you must implement a subclass `ScapeGoatIntKey<V>` that extends `ScapeGoatTree<Integer, V>` and supports efficient range queries over integer keys via `getRange(int start, int end)`.

`ScapeGoatTree` should:

- Support insertion, deletion, and retrieval of key-value pairs.
- Automatically rebalance unbalanced subtrees using the scapegoat tree strategy.
- Find and rebuild the subtree at the first unbalanced node (the "scapegoat") after insertions.
- Rebuild the full tree after deletions if a size invariant is violated.

The `ScapeGoatIntKey` class must support:

- Construction via a no-arg constructor or by passing in an initial root key-value pair.
- A method `getRange(int start, int end)` that returns a sorted list of all values whose keys lie in the range $[start, end]$.

1.2 Constraints and Requirements

- `ScapeGoatTree` should maintain:
  - A root node (`Node<K,V> root`),
  - A count of current nodes (`nodeCount`),
  - A count of the maximum size ever reached (`maxNodeCount`).
- Use `compareTo()` and `equals()` for key comparisons. Do not rely on object reference equality.
- Rebuilds must produce perfectly balanced BSTs using an in-order traversal followed by recursive reconstruction.
- Insertion must be rejected for duplicate keys (no modification to the tree).
- Deletion must follow standard BST deletion using successor nodes.
- After deletion, if `nodeCount` $\leq \alpha\cdot$ `maxNodeCount`, rebuild the entire tree.
- The `getRange` method must perform in $O(\log n + r)$ where r is the number of results returned.

1.3 Interface and Required Methods

Your `ScapeGoatTree` and `ScapeGoatIntKey` classes must implement:

- `void add(K key, V value)` inserts key-value pair; may rebalance subtree
- `void remove(K key)` removes the given key if present
- `V get(K key)` retrieves the value associated with the given key
- `int size()` returns the number of nodes in the tree
- `Node<K, V> root()` returns the current root of the tree
- `void clear()` resets the tree to empty
- `List<V> getRange(int start, int end)` returns all values with keys in $[start, end]$ (only in `ScapeGoatIntKey`)

1.4 Implementation Notes

- Balance must be maintained using the scapegoat tree algorithm (see Wikipedia).
- Use the `ALPHA_THRESHOLD` constant (from the interface) to determine whether a node is balanced.
- Use the `rebuild()` helper to restructure unbalanced subtrees and reassign parent pointers.
- When rebuilding, pick the middle of a sorted node list and recursively attach subtrees.
- The `getRange` method must use an in-order traversal, adding values whose keys lie in the specified range.
- Null keys must be ignored and not inserted. Retrieval of nonexistent keys should return null.

1.5 Testing and Grading

Your code will be tested by the JUnit suite in `ScapeGoatTreeTest`, which covers:

- correctness of add, remove, get, root, and clear operations
- handling of null keys and invalid removals
- rebalance operations after insertions and deletions
- deep equality and use of compareTo (e.g., complex object keys)
- correctness and efficiency of the `getRange` method
- robustness against extreme values and large insertions
- performance of `getRange` under random query stress tests

You may use `java.util.List`, `ArrayList`, and other collection classes as needed. All internal tree operations must be implemented from scratch. Do not use `TreeMap`, `TreeSet`, or similar classes.

Performance Constraints The test suite may insert up to 2×10^6 elements and run thousands of queries.

2 Story 1: The Monstrous Road Minimizer

To balance the monster density across the new game map, content creators have decided to connect every city with as few roads as possible, while keeping the total length of all roads to a minimum. Your tool, `WorldBuildingManager`, reads a list of city coordinates and selects the set of roads (undirected connections) that spans all cities with minimal total length, using the Euclidean distance between city pairs as the edge weight.
Implement the method

```
List<CityEdge> getMinimumConnectingRoads(String filename)
```

which reads its input from `filename` and returns a list of `CityEdge` objects representing the roads in the minimal network.

2.1 Specifications

- You may use any of your own data-structure implementations, but only `List`, `ArrayList`, and `java.util.stream.*` from the Java standard library.
- The selected network must connect all C cities with exactly $C - 1$ roads.
- Road length between city i at (x_i, y_i) and city j at (x_j, y_j) is

$$\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}.$$

2.2 Input

```
C
a_0 b_0
a_1 b_1
...
a_{C-1} b_{C-1}
```

where

- C is the number of cities.
- Each of the next C lines contains two integers a_i, b_i, the coordinates of city i.

2.3 Output

```
totalWeight
```

Print a single floating-point number: the sum of the lengths of the selected roads, formatted with six decimal places.

2.4 Constraints

- $1 \leq C \leq 5000$
- $0 \leq a_i, b_i \leq 10000$
- No two cities share the same coordinates.

2.5 Sample Input 1

```
8
550 550
641 508
573 453
489 471
450 549
487 627
571 648
640 595
```

2.6 Sample Output 1

```
619.529741
```

2.7 Explanation 1

We compute the lengths of the chosen roads and sum them directly (no naming of algorithm):

$$\begin{aligned}
d_1 &= \sqrt{(573-489)^2+(453-471)^2} = 85.896\\
d_2 &= \sqrt{(450-487)^2+(549-627)^2} = 86.328\\
d_3 &= \sqrt{(487-571)^2+(627-648)^2} = 86.588\\
d_4 &= \sqrt{(571-640)^2+(648-595)^2} = 87.021\\
d_5 &= \sqrt{(640-641)^2+(595-508)^2} = 87.021\\
d_6 &= \sqrt{(641-573)^2+(508-453)^2} = 87.491\\
d_7 &= \sqrt{(550-487)^2+(550-627)^2} = 99.494
\end{aligned}$$

Summing:

$$85.896+86.328+86.588+87.021+87.021+87.491+99.494 = 619.529741.$$

2.8 Sample Input 3

```
7
2000 0
6000 0
0 3464
4000 3464
8000 3464
2000 6928
6000 6928
```

2.9 Sample Output 3

```
23999.471994
```

2.10 Explanation 3

For this arrangement, all six chosen roads have the same length:

$$d = \sqrt{2000^2 + 3464^2} \approx 3999.912$$

and there are six such segments, so the total is

$$6 \times 3999.912 = 23999.471994.$$

3 Story 2: The Electric Voyage Planner

In the sprawling grid of intercity highways on the terraformed moons of Europa, electric vehicles (EVs) must chart a course that respects their battery limits. The `EVRoutePlanner` is your mission control: given a map of cities, waypoints, and charging stations, it computes the shortest path from a starting location to a destination, ensuring that no single hop exceeds the vehicle's maximum range. If the road ahead is too long, the planner will guide the EV through intermediate charging stations.
Your task is to implement:

```
List<Location> findEVRoute(String filename, int sourceId, int destId, double
                                        maxRange)
```

which reads the map data from `filename`, then returns the sequence of `Location` objects from `sourceId` to `destId`. If no valid route exists, return `null`.

3.1 Specifications

- The method signature:

  ```
  List<Location> findEVRoute(String filename, int sourceId, int destId, double maxRange)
  ```

  must compute a route such that each consecutive pair of stops is at most `maxRange` apart (Euclidean distance).

- The input file `filename` has the following format:
  - First line: an integer N, the number of locations.
  - Next N lines: five space-separated tokens:

    `id x y type name`

    where

    `id` unique integer identifier (0-based).

    `x, y` integer Cartesian coordinates.

    `type` one of `CITY`, `CHARGING_STATION`, `RESTAURANT`, `TOURIST_ATTRACTION`, or `REGULAR_WAYPOINT`.

    `name` a single-word string identifier.

- The planner may only stop at charging stations (plus the source and destination), so intermediate hops are restricted to nodes of type `CHARGING_STATION`.

- Return the list of `Location` objects along the shortest valid route, or `null` if no route exists.

3.2 Input

```
N
id x y type name
...
```

3.3 Output

If a route exists:

```
sourceId destId maxRange totalDistance
id0 id1 id2 ... idk
```

If no route exists:

```
sourceId destId maxRange -1.0
NO_PATH
```

Here `totalDistance` is printed with three decimal places.

3.4 Constraints

- $1 \leq N \leq 10^5$
- $-10^4 \leq x, y \leq 10^4$
- $0 \leq \texttt{sourceId}, \texttt{destId} < N$
- $0.0 < \texttt{maxRange} \leq 10^4$

3.5 Sample Input 1

```
10
0 0 0 CITY StartCity
1 50 0 CHARGING_STATION ChargingStation1
2 100 0 CHARGING_STATION ChargingStation2
3 25 25 RESTAURANT Restaurant1
4 75 25 TOURIST_ATTRACTION Museum1
5 150 0 CHARGING_STATION ChargingStation3
6 200 0 CHARGING_STATION ChargingStation4
7 125 50 RESTAURANT Restaurant2
8 175 25 TOURIST_ATTRACTION Park1
9 250 0 CITY DestinationCity
```

3.6 Sample Output 1

```
0 9 75.0 250.000
0 1 2 5 6 9
```

3.7 Explanation 1

- $\texttt{sourceId} = 0$, $\texttt{destId} = 9$, $\texttt{maxRange} = 75.0$.
- Valid stops are only at charging stations $\{1, 2, 5, 6\}$ plus source and destination.
- The planner hops: $0 \to 1$ (distance 50), $1 \to 2$ (distance 50), $2 \to 5$ (distance 50), $5 \to 6$ (distance 50), $6 \to 9$ (distance 50).
- Total distance $= 5 \times 50 = 250.000$.

3.8 Sample Input 2

```
6
0 0 0 CITY StartCity
1 50 0 RESTAURANT Restaurant1
2 100 0 CHARGING_STATION ChargingStation1
3 200 0 TOURIST_ATTRACTION Museum1
4 300 0 CHARGING_STATION ChargingStation2
5 400 0 CITY DestinationCity
```

3.9 Sample Output 2

```
0 5 80.0 -1.0
NO_PATH
```

3.10 Explanation 2

- `sourceId` $= 0$, `destId` $= 5$, `maxRange` $= 80.0$.
- The only charging station reachable from 0 is at 100 units (node 2), which exceeds the range.
- No valid sequence of hops exists, so output -1.0 and `NO_PATH`.

4 Testing

The following test files are provided to you on Vocareum:

`test/CoreUtilsTest/ScapeGoatTreeTest.java` tests `src/CoreUtils/ScapeGoatTree.java` and src/CoreUtils/ScapeGoatIntKey.java
`test/StoriesTest/EVRoutePlannerTest.java` tests `src/Stories/EVRoutePlanner.java`
`test/StoriesTest/WorldBuildingManagerTest.java` tests `src/Stories/WorldBuildingManager.java`

These tests act as an indicator of your progress but do not guarantee the displayed score. The final grade will be determined by the tests run on Vocareum. You are encouraged to create your own `main` methods and additional test files to debug any errors you face.

Submission

Your submission on Vocareum should include only the following Java files:

- `src/CoreUtils/ScapeGoatTree.java`
- `src/CoreUtils/ScapeGoatIntKey.java`
- `src/CoreUtils/MST.java`
- `src/Stories/EVRoutePlanner.java`
- `src/Stories/WorldBuildingManager.java`

Make sure that you upload the files to the correct locations. Do not edit any other files since those will be replaced before grading. Do not add any disallowed imports. Before submitting, remove all lines that contain `System.out`, even in comments.

The grading on Vocareum may take a few minutes, so please be patient.

All submission instructions are the same for all projects and can be found in the syllabus under "Programming Projects." You are allowed exactly **10 submissions per project** on Vocareum. No additional submissions will be granted.

You are allowed to use any of the previous structures we created in previous projects to solve the stories.

You are allowed to create private helper classes in any of the files.

The project download may take about 2 minutes—please be patient.

5 Java Version and Compatibility Requirements

- **Java 17 only**: Your code must be compatible with Java 17. Do not use features or methods from newer Java versions (Java 18+).
- **Supported APIs**: Only use standard Java 17 library methods and classes.
- **Forbidden imports**: As stated throughout this handout, you cannot use `java.util.*` classes except where explicitly permitted.
- **Testing environment**: All code will be compiled and tested using Java 17. Code that uses newer Java features will fail to compile and receive zero points.