4 Introduction

Complete the task(s) below described in Section 6. Certain classes have been provided for you alongside this specification in the Student files folder. A very basic main has been provided. **Please note this main is not extensive and you will need to expand on it**. Remember to test boundary cases. Submission instructions are given at the end of this document.

5 Background

5.1 Self-Organizing data structures

A self-organizing data structure is a data structure that has some rule(s) or algorithm attached to the access method for each node in the data structure. These rule(s) or algorithms result in changing the structure or order of the data structure in an attempt to speed up the problem of searching for data in the data structure [AW05]. The access method is a function or function that searches for the data in the data structure and returns the data. The main goal of a self-organizing data structure is to move away from a lookup time of O(n) and towards an instant lookup time of O(1). Examples of self-organizing data structures are self-organizing lists, binary trees and m-way trees.

5.2 Treaps

A treap is a variation of a randomized search tree algorithm. Treaps differ from standard binary search trees in the following ways:

1. Node:
   Nodes in a standard binary search tree only has a data member which stores the data of the node. This differs from treaps in the sense that nodes has an additional member which is the priority of the node.

2. Properties:
   Standard Binary Search trees only maintain the property that data that is smaller than the current node's data is part of the left subtree of the node and data that is greater than the current node's data are part of the right subtree of the node. With treaps, an additional property is introduced. The max heap property. This property states that the data of node n is greater or equal to either of its children [ASSS86]. Treaps combine the idea of Heaps (where the aforementioned property originated from) and Binary Search Trees. Thus Treaps maintain the max heap property of heaps as well as the efficient search property of Binary Search Trees. The max heap property is applied to the priority of each treap node and the binary search is applied to the data of each treap node.

Treaps assign priorities randomly to each node which originates from the randomized search tree algorithm. This allows for the average case of the treap to be perfectly balanced [SA96].

The standard implementation of a treap is to use an array but for this assignment you will be implementing the treap as a tree.

5.3 Self-organizing treaps

This assignment will also put a twist on the standard treap in the sense that we add the idea of self-organizing data structures on top of the treap. Thus with each access of a node, the priority of the node will increase and a rotation may be required to maintain the max heap property.

5.4 Database

A database is usually a collection of rows that are somehow related to each other. It is possible to have unstructured databases (e.g. NoSQL) databases but for this assignment, you will be implementing a structured database.

5.4.1 Database

A structured database refers to a database that has a strict structure that cannot be changed or will result in possibly undefined behaviour. A database's structure is described by a set of properties the data has which will be called columns. Rows in the database are a grouping of data that has all the properties and that are closely related to each other. Thus each row's properties are somehow closely related to each other. Usually, databases have multiple tables to model the real world more accurately. Due to complexity, this assignment will only be implementing a single-tabled database. Databases usually have the following set of operations:

1. Insert
2. Update
3. Delete
4. Search

In this assignment you will be implementing each of these operations.

5.4.2 Indexing

Searching is one of the most important features of a database. A naive approach is to linearly search through the database for a record or possibly multiple records. As a core part of COS212 this assignment will try to optimize this naive approach. In databases especially relational databases[1] is to index columns that are used to search through often. Indexing implies building an external data structure that will increase the efficiency of the searching algorithm when searching for data. Usually, this is accomplished by using a B+ or B* tree. This assignment

[1]Relational Database - https://www.oracle.com/za/database/what-is-a-relational-database/

will attempt to use a different data structure: Self Organizing Treaps. This will hopefully increase the efficiency of the searching algorithm over the linear searching algorithm.

6 Task

Your task for this assignment would be to implement the following class diagram as described in later sections.

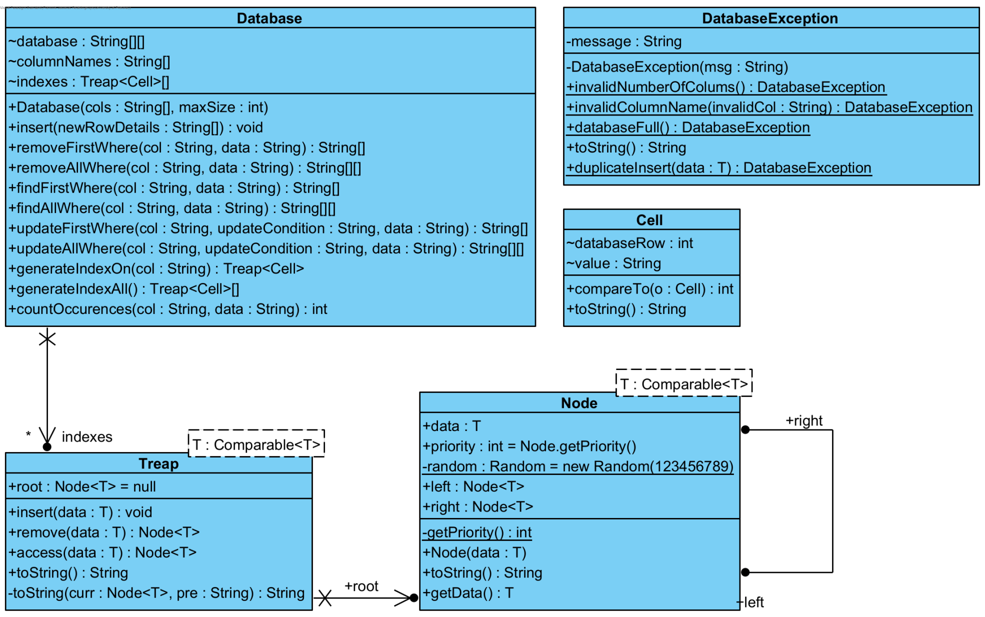
Database
~database : String[][]
~columnNames : String[]
~indexes : Treap<Cell>[]
+Database(cols : String[], maxSize : int)
+insert(newRowDetails : String[]) : void
+removeFirstWhere(col : String, data : String) : String[]
+removeAllWhere(col : String, data : String) : String[][]
+findFirstWhere(col : String, data : String) : String[]
+findAllWhere(col : String, data : String) : String[][]
+updateFirstWhere(col : String, updateCondition : String, data : String) : String[]
+updateAllWhere(col : String, updateCondition : String, data : String) : String[][]
+generateIndexOn(col : String) : Treap<Cell>
+generateIndexAll() : Treap<Cell>[]
+countOccurences(col : String, data : String) : int
DatabaseException
-message : String
-DatabaseException(msg : String)
+invalidNumberOfColums() : DatabaseException
+invalidColumnName(invalidCol : String) : DatabaseException
+databaseFull() : DatabaseException
+toString() : String
+duplicateInsert(data : T) : DatabaseException
Cell
~databaseRow : int
~value : String
+compareTo(o : Cell) : int
+toString() : String
*
indexes
T : Comparable<T>
Treap
+root : Node<T> = null
+insert(data : T) : void
+remove(data : T) : Node<T>
+access(data : T) : Node<T>
+toString() : String
-toString(curr : Node<T>, pre : String) : String
+root
T : Comparable<T>
Node
+data : T
+priority : int = Node.getPriority()
-random : Random = new Random(123456789)
+left : Node<T>
+right : Node<T>
-getPriority() : int
+Node(data : T)
+toString() : String
+getData() : T
+right
+left

Figure 1: Class diagram

For this assignment, you will be creating a single table database with a set of functionalities, described in Section 6.2. To perform the indexing this assignment will use a Treap which will be described in Section 6.1

6.1 Generic Self-Organizing Treap

This task will focus on creating the generic self-organizing treap. You will thus be implementing the following class diagram:

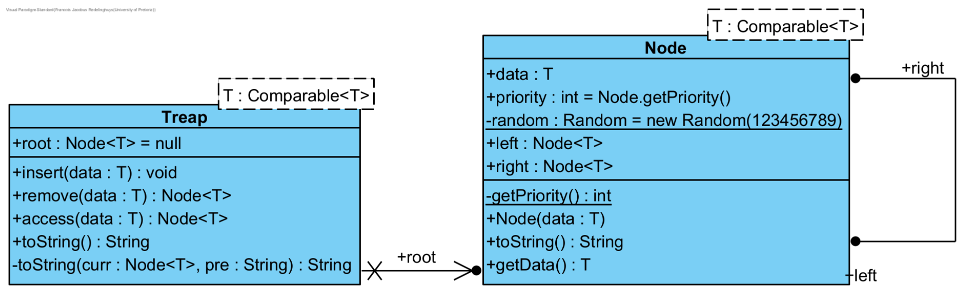

Figure 2: Treap class diagram

6.1.1 Node

The node class will be the class representing nodes in the tree.

- Members:
  - data: T
    - This will be the data that is stored in the node.
  - priority: int
    - On creation of an object the priority will immediately be assigned a value using the getPriority() function.
    - Note this member variable will change over time. Thus it **cannot** be assumed that the priority will remain constant.
    - This member variable will act as the priority value for nodes in the treap.
  - random: Random
    - This is a static variable.
    - Do not change this declaration or reinitialize the member variable.
  - left: Node<T>
    - This member variable will represent the left child of the current node.
  - right: Node<T>
    - This member variable will represent the right child of the current node.
- Functions:
  - getPriority(): int
    - This is a static function.
    - This function is provided and **should not** be changed.
    - Altering this function will result negatively in your final mark.
  - Node(data: T)
    - This is the constructor for the Node class and should initialize all appropriate member variables.
  - toString(): String
    - This function is provided and **should not** be changed.
    - Altering this function will result negatively in your final mark.
    - This function will be used to print out the Node.
  - getData(): T
    - This will return the data associated with the node.

6.1.2 Treap

This class will represent the Treap in the assignment. Note duplicates are **not** allowed in the treap.

- Members:
  - root: Node<T>
    - This is the root of the treap.
    - This member variable should be initialized to null initially.
    - *Hint: remember java has the ability to initialize member variables with either constant values or static function calls*
- Functions:
  - insert(data: T): void
    - This function should insert the data into the tree.
    - The function should insert the data in a standard binary search tree way and then rotate the nodes in the tree such that the heap properties of a treap are maintained.
    - Use the priority of the newly created node as the priority in the treap rotation algorithm.
    - If the passed-in parameter is already contained in the treap the treap should **throw** a **DatabaseException** using the **duplicateInsert** function and passing the data parameter.
    - Note: The insertion algorithm in the lecture slides will change as follows: **3ii) if the inserted/updated node has a greater or equal priority than its parent, then the node should be rotated about its parent**
  - remove(data: T): Node<T>
    - This function should remove and return the node whose data element matches the passed-in parameter.
    - In the case that there is no node in the tree whose data member variable matches the passed-in parameter the function should return null and leave the tree unaltered.
  - access(data: T): Node<T>
    - This function should access the node in the tree whose data element matches the passed-in parameter and return the node.
    - This function should increment the priority of the node by 1 and perform the rotations as specified in the insertion algorithm such that the heap properties are maintained.

    * *Hint: Remember to still rotate about the parent even when the parent has the same priority. Only stop rotating once the parent's priority is strictly greater than the child*
    * If no nodes in the tree have a data element that matches the passed-in parameter the function should return null.
- toString(): String
    * This function is provided and should not be altered.
    * It can be used to print out the tree.
    * Altering this function will negatively impact your mark.
- toString(curr: Node<T>, pre: String): String
    * This function is provided and should not be altered.
    * It can be used to print out the tree.
    * Altering this function will negatively impact your mark.

6.2 Database

This section will discuss the implementation of a simplified single-table database. Note columns in the database will only be indexed (Refer to section 5.4.2 for an explanation of indexing) once the appropriate command is issued to the database. The figure below dictated the functions and classes that will be described in the preceding sections.

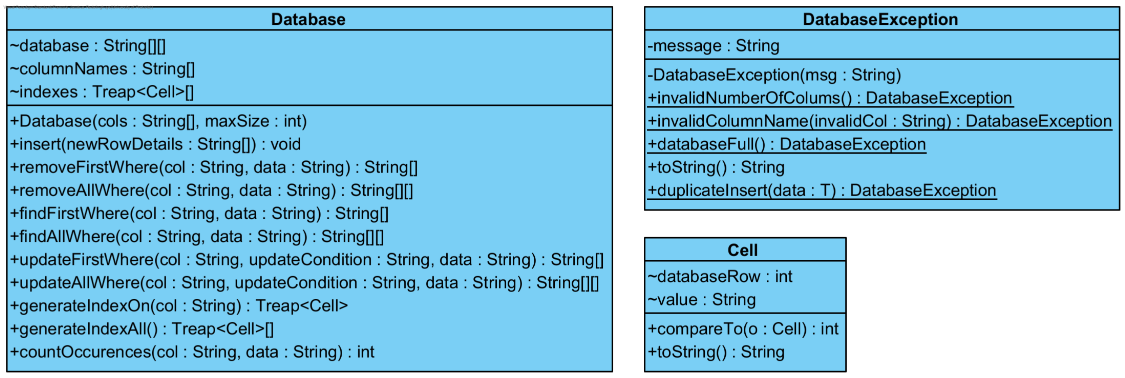
Database
~database : String[][]
~columnNames : String[]
~indexes : Treap<Cell>[]
+Database(cols : String[], maxSize : int)
+insert(newRowDetails : String[]) : void
+removeFirstWhere(col : String, data : String) : String[]
+removeAllWhere(col : String, data : String) : String[][]
+findFirstWhere(col : String, data : String) : String[]
+findAllWhere(col : String, data : String) : String[][]
+updateFirstWhere(col : String, updateCondition : String, data : String) : String[]
+updateAllWhere(col : String, updateCondition : String, data : String) : String[][]
+generateIndexOn(col : String) : Treap<Cell>
+generateIndexAll() : Treap<Cell>[]
+countOccurences(col : String, data : String) : int
DatabaseException
-message : String
-DatabaseException(msg : String)
+invalidNumberOfColums() : DatabaseException
+invalidColumnName(invalidCol : String) : DatabaseException
+databaseFull() : DatabaseException
+toString() : String
+duplicateInsert(data : T) : DatabaseException
Cell
~databaseRow : int
~value : String
+compareTo(o : Cell) : int
+toString() : String

Figure 3: Database class diagram

6.2.1 DatabaseException

This class has been provided for you to use in the Database class and its use will be discussed in further detail in Section 6.2.3. Note this class is **given** and **will** be overwritten.

6.2.2 Cell

This class will be used as the datatype for nodes in the Treap when a treap is created for a column which will be discussed in Section 6.2.3. This class has also been provided **but** can be **altered**. To ease the construction process of a Cell object a Cell constructor can be added.

6.2.3 Database

This class will act as the Database or Database Management System of the simplified database.

- Members:
  - database: String[][]
    - This is a 2D array that will represent the database table.
    - Empty rows should be saved as an array of null.
  - columnNames: String[]
    - This is a 1D array that will contain the names of each column in the database.
    - The order of the names in the column is the same order in that columns will be stored in the database.
  - indexes: Treap<Cell>[]
    - This will be an array of treaps that will be used to index each column **only on request**.
    - The order of the treaps in the array is the same order in that columns will be stored in the database.
- Functions:
  - Database(cols: String[], maxSize: int)
    - This is the constructor for the database class.
    - Initialize the database member variable to have a size of maxSize and initialize all the rows to have an array of null values.
    - Initialize the columnNames array to the contents of the cols array.
    - Initialize the indexes array to have a size of $len(cols)$.
    - Initialize each index in the indexes array to null.
  - insert(newRowDetails: String[]): void
    - This function should attempt to add a new record into the database at the first available row.
    - If the database is full the function should **throw** an **DatabaseException** using the **databaseFull()** function.
    - For every indexed column add the corresponding data of the row to the column's treap.
    - If the treap throws an exception during the insertion of data the function should re-throw the exception and **not** add the row to the database.
    - Use each non-null treap's access function to determine if the corresponding element in the row is already contained in the treap before adding the row to the database. Your code should throw the exception on the **first** duplicate that was encountered.

    - If the size of the passed-in array is not equal to the number of columns in the database **throw** a **DatabaseException** using the **invalidNumberOfColums()** function.
    - Your checks should be as follows:
      1. First check if the row has the correct structure
      2. Check if there is any duplicate(s) in any of the treaps.
      3. Lastly check if the database is full.
  - removeFirstWhere(col: String, data: String): String[]
    - This function should remove a row from the database and return the row.
    - The row that should be removed is the row that has the first occurrence of the passed-in data parameter in the column whose name matches the col passed-in parameter.
    - First occurrence is defined as follows:
      - If the column has been indexed (i.e. has a treap associated with it) then your code should use the remove function of the treap to determine which row number should be removed.
      - If the column has not been indexed (i.e. does not have a treap associated with it) then your code should linearly search through the database to find the first occurrence
    - **Remember to remove the data that belongs to the row out of all the non-null treaps.**
    - Set rows to an array of null values when removed from the database.
    - If no column has a name that matches the passed-in parameter the function should **throw** a **DatabaseException** using the **invalidColumnName(col)** function.
    - If no row was removed the function should return a String array of length 0.
  - removeAllWhere(col: String, data: String): String[][]
    - This function should remove all the rows whose data, of the column that has a name that matches the passed-in col, matches the passed-in data parameter.
    - The function will return a String array containing all the rows that have been removed.
    - Do not worry about the order of the rows in the resulting array.
    - Use the **removeFirstWhere** function to remove each row from the database.
    - This function should not perform exception swallowing [2].
    - If no rows are deleted the function should return an array of size $|0 \times 0|$

[2] Exception Swallowing - `https://en.wikipedia.org/wiki/Error_hiding`

    - If no column has a name that matches the passed-in parameter the function should **throw** a **DatabaseException** using the **invalidColumnName(col)** function.
  - findFirstWhere(col: String, data: String): String[]
    - This function should return the first occurrence of a row whose appropriate column's data matches the passed-in data parameter.
    - First occurrence is defined as follows:
      - If the column has been indexed (i.e. has a treap associated with it) then your code should use the access function of the treap to determine the row number of the row that should be returned.
      - If the column has not been indexed (i.e. does not have a treap associated with it) then your code should linearly search through the database to find the first occurrence
    - If no row is found the function should return an array with a size of 0.
    - If there is a treap associated with the column whose name matches with col then the node in the tree should be accessed as described in Section 6.1.2's access() function.
    - If no column has a name that matches the passed-in parameter the function should **throw** a **DatabaseException** using the **invalidColumnName(col)** function.
  - findAllWhere(col: String, data: String): String[][]
    - This function should return all the rows that has a data entry for the appropriate column that matches the passed-in data parameter.
    - If there is a treap associated with the column whose name matches with col then the node in the tree should be accessed as described in Section 6.1.2's access() function.
    - If no row contains the data entry for the appropriate column that matches the passed-in parameter the function should return an array with a size of 0.
    - If no column has a name that matches the passed-in parameter the function should **throw** a **DatabaseException** using the **invalidColumnName(col)** function.
  - updateFirstWhere(col: String, updateCondition: String, data: String): String[]
    - This function should update the first row whose data member, for the column that matches the passed-in col's parameter, matches the passed-in updateCondition parameter.
    - The element in the row that should be updated is the data element that belongs to the column that matches the col passed-in parameter.
    - The data element should be changed to the data passed-in parameter.

    * If the column that matches col is indexed then the following procedure should be followed to update the node in the treap:
        1. Remove the cell that is associated with the data element from the treap using the treap's remove function.
        2. Insert the new node with the updated data into the treap.
    * Else if no treap is associated with the column then a simple linear search will suffice.
    * The function should return the row with the updated element.
    * If no row was updated the function should return an array with a size of 0.
    * If no column has a name that matches the passed-in parameter the function should **throw** a **DatabaseException** using the **invalidColumnName(col)** function.

- updateAllWhere(col: String, updateCondition: String, data: String): String[][]
    * This function should update all the rows whose data member, for the column that matches the passed-in col's parameter, matches the passed-in updateCondition parameter.
    * The element in the row that should be updated is the data element that belongs to the column that matches the col passed-in parameter.
    * The data element should be changed to the data passed-in parameter.
    * Use the updateFirstWhere function to update each of the appropriate nodes.
    * The function should return an array containing all the updated rows
    * If no rows were updated the function should return an array with a size of 0.
    * If no column has a name that matches the passed-in parameter the function should **throw** a **DatabaseException** using the **invalidColumnName(col)** function.

- generateIndexOn(col: String): Treap<Cell>
    * This function should generate a treap for the column whose name matches the passed-in col parameter and return it.
    * If a treap already exists for the column the function should return that treap.
    * If a treap does not exist element should be linearly inserted into the treap based on the order in which they are in the database starting at row 0 up to row $len(database) - 1$
    * If no column has a name that matches the passed-in parameter the function should **throw** a **DatabaseException** using the **invalidColumnName(col)** function.
    * If during the insertion process, an exception is encountered nullify the newly created tree and rethrow the exception.
    * If the database is empty the function should return an empty treap

    - *Note: There is a difference between an empty treap and a null treap*
- generateIndexAll(): Treap<Cell>[]
    - This function should return the indexes array.
    - The function should ensure that each column in the table has a treap associated with it before returning.
    - *Hint: use the generateIndexOn() function to populate the resulting array*
    - If a DatabaseException is received from the generateIndexOn function swallow the exception and continue to the next index.
    - If the database is empty the resulting array should be filled with empty treaps.
- countOccurences(col: String, data: String): int
    - This function should return the number of occurrences of the passed-in data parameter in the column whose name matches the passed-in col parameter.
    - If there are no matches the function should return 0.
    - If the database is empty the function should return 0.
    - *Tip: This function will come in handy for functions that have the work All in their name*
    - If no column has a name that matches the passed-in parameter the function should **throw** a **DatabaseException** using the **invalidColumnName(col)** function.

7 Equivalent SQL queries

Assume the database has the following structure of prominent Computer Scientist researchers:

ID	**computer scientist name**	**research area**	**year first publication**
0	Ada Lovelace	Algorithms	1843
1	Grace Hopper	Programming Languages	1945
2	Alan Turing	Cryptography	1936
3	Donald Knuth	Algorithms	1962
4	Barbara Liskov	Object-Oriented Programming	1968
5	Tim Berners-Lee	World Wide Web	1989
6	John Backus	High-Level Languages	1954
7	Edsger W. Dijkstra	Computer Science Theory	1959
8	Vint Cerf	Internet	1967
9	Shafi Goldwasser	Cryptography	1982
10	Frances E. Allen	Compilers	1962
11	Geoffrey Hinton	Neural Networks	1986
12	John McCarthy	Artificial Intelligence	1955
13	Douglas Engelbart	Human-Computer Interaction	1962
14	Stephen Cook	Computational Complexity	1971

Table 1: Computer Scientists and Their Research Areas

The SQL equivalent queries of each of the functions described in Section 6.2.3 is given below:

- Database(["ID","Computer Scientist Name","Research Area","Year of First Publication"], 15):

```sql
CREATE DATABASE IF NOT EXISTS computer_scientists;

USE computer_scientists;

CREATE TABLE IF NOT EXISTS computer_scientists (
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    research_area VARCHAR(255) NOT NULL,
    year_first_publication INT NOT NULL
);
```

- insert(["16","Edmund Clarke","Model Checking","1982"])

```sql
INSERT INTO computer_scientists (id, name, research_area,
   year_first_publication)
VALUES (16, 'Edmund␣Clarke', 'Model␣Checking', 1982);
```

- removeFirstWhere("research area", "Algorithms")

```
DELETE FROM computer_scientists WHERE research_area = 'Algorithms'
   LIMIT 1;
```

- removeAllWhere("research area","Algorithms")

```
DELETE FROM computer_scientists WHERE research_area = 'Algorithms';
```

- findFirstWhere("year first publication", "1962")

```
SELECT * FROM computer_scientists WHERE year_first_publication = 1962
   LIMIT 1;
```

- findAllWhere("year first publication", "1962")

```
SELECT * FROM computer_scientists WHERE year_first_publication = 1962;
```

- updateFirstWhere("research area","Artificial Intelligence", "AI")

```
UPDATE computer_scientists SET research_area = 'AI' WHERE research_area
   = 'Artificial␣Intelligence' LIMIT 1;
```

- updateAllWhere("research area","Artificial Intelligence", "AI")

```
UPDATE computer_scientists SET research_area = 'AI' WHERE research_area
   = 'Artificial␣Intelligence';
```

- generateIndexOn("research area")

```
CREATE INDEX idx_research_area ON computer_scientists (research_area);
```

- generateIndexAll()

```
CREATE INDEX idx_id ON computer_scientists (id);
CREATE INDEX idx_name ON computer_scientists (name);
CREATE INDEX idx_research_area ON computer_scientists (research_area);
CREATE INDEX idx_year_first_publication ON computer_scientists
   (year_first_publication);
```

- countOccurences("year first publication", "1962")

```
SELECT COUNT(*) FROM computer_scientists WHERE year_first_publication =
   1962;
```