For the following questions, we will consider a schema consisting of the following relations:

Store:	[SID, City, EID]
Employee:	[EID, Name, Salary, SID]
Article:	[AID, Name, Producer, Price, ProductionDate]
Inventory:	[AID, SID, Count]
Invoice:	[IID, SID, Customer]
Item:	[IID, SID, AID, Count]

`Store.EID` is a foreign key referencing `Employee.EID` and designates the store manager. `Employee.SID` is a foreign key referencing `Store.SID` and describes in which store this employee is employed. The inventory table stores (with foreign keys) how many articles (`AID`) are in stock at which store (`SID`). An invoice is generated at a store (`SID`) and consists of multiple items, where each item is composed of an article (`AID`) and a count.

Question 1: Index tuning (1 P.)

All relevant primary indices and foreign keys are already created (e.g.: *Inventory.SID* is a foreign key referencing *Store.SID*).

As the system seems to be very unresponsive lately, the owner of the brand tasked you with improving the performance of the system. An analysis of the query load showed that the following values are queried regularly:

- Q1: A list of all articles together with the stores in which they are present.
- Q2: A histogram showing how many employees work per different rates of salary, given a specific store.
- Q3: All articles for a specified price range.
- Q4: All articles for a specified price range (as previously) for a given producer.
- Q5: An overview of all of the customers of a given store.
- Q6: An overview of the articles with price within a specific range and date of production in a specified time period, with respect to the selectivity of both predicates.

Which indices will you create to improve the performance of the given queries? For each index discuss why you created it. TIP: Maybe some indices can be used for multiple queries. Which of your created indices could benefit from being a hash index, instead of a B+ tree?

Question 2: Index Costs (1 P.)

Given the following values for the `Article` table: There are 400 000 articles stored in this relation. Each is numbered sequentially between 1 and 400 000. Each page on the disk can store exactly 5 article tuples. A B^+ tree was created as primary index. It has a height of h and the leaves contain 15 entries, together with a pointer to a data page.

a) How many pages have to be accessed to answer the query `select * from Article where AID between 10000 and 30000`? (HINT: Keep the classification of indices from the lecture in mind)

b) After analyzing the query load, the database administrator notices that many queries select articles originating from the same producer. He decides to cluster the table `Article` by the `Producer` column. An additional index is created on the primary key `AID`. How would the lecture classify this index? How many page accesses would the query from question a) require now?

Question 3: Composite Index (1 P.)

The `Employee` table contains 30 000 employees, of which exactly two tuples fit into one page. Each employee has a (uniformly distributed) salary between 40 000 and 150 000. There are 8 stores, each with the same amount of employees. The `Employee` table has two secondary composite-key indices on `(Salary,EID)` and `(EID,Salary)`. Both indices are B^+ trees with a height of h and each leaf can store 10 references to pages.

- Which index requires less page accesses for the query `select * from Employee where EID = 15 and Salary > 75 000`? Please calculate the number of expected page accesses for both indices.

Question 4: Index Implementation (1 P.)

This question requires you to implement code in any programming language you want. The code has to compile and return the correct result. In OLAT, we provide a Java template with most of the boilerplate code already in place. This template checks your result for correctness and times the execution. Please don't change anything else than the specified parts of the code.

If you use a different language than Java, provide instructions on how to compile and run your code. Also, your code should then include the same checks as the template Java main method.

Given a dataset of N tuples, consisting of a unique ID and a string value with $|S| = \frac{N}{10}$ distinct strings. Initially the dataset is ordered by the ID and each tuple is assigned a random string from S like the following:

(1, ABC), (2, BCD), (3, ABC), (4, XYZ), (5, XYX), (6, BCD), (7, XYZ)

The system should be able to answer the following queries:

- Return all tuples for which the string is equal to a given query string q.
- Return all tuples for which the string is lexicographically equal or greater to a given query string q.

Your task is to:

a) Implement a default access method, which iterates all tuples and returns the correct results. (Given in the Java template code)

b) Implement a dense index as introduced in the lecture.

If it is not possible for your index implementation to perform one of these operations, then use the default execution method and state why it is not possible.

Execute your code a few times with different query strings (For the java template, just execute it multiple times). Describe how the index creation time and query times for both operations differ in your implementations. You may change the values of N and S to check the effects on your implementation.