Project Description: Sequential Search Symbol Table

A *Sequential Search Symbol Table* is a straightforward implementation of a key–value data structure (often called a “symbol table” or “map”) where all keys and values are stored in an unsorted collection, typically a linked list. This project focuses on creating a symbol table that supports various operations (as shown in the API image) by scanning through the collection linearly whenever an operation is requested.

1. Overview

1. **Data Structure Choice**
    - The simplest way to implement a sequential search symbol table is by using a singly linked list (though one could also use a dynamic array).
    - Each node in the list holds a key–value pair along with a reference to the next node.
2. **Core Operations**
    - **Insertion (put):**
      Search the list to see if the key already exists.
        - If found, update the existing value.
        - If not found, insert a new node (often at the front of the list) with the given key–value pair.
    - **Search (get):**
      Traverse the list until the key is found or the end is reached. Return the associated value if found; otherwise return `null` (or equivalent).
    - **Deletion (delete):**
      Traverse the list until the key is found, then remove the corresponding node from the linked structure.
3. **Range Queries**
   The API includes several methods beyond the basic `put` , `get` , and `delete` . Even though sequential search is not particularly efficient for these operations, the table can still be made to support them:
    - **contains(Key key)**
      Returns `true` if `get(key)` is not `null` ; otherwise `false` .
    - **isEmpty()**
      Checks if the symbol table has zero elements.
    - **size()**
      Returns the number of key–value pairs stored.

- **min(), max()**
  Find the smallest and largest keys by traversing the entire list and comparing each key.
- **floor(Key key), ceiling(Key key)**
  Traverse the list and identify the largest key less than or equal to `key` (floor) or the smallest key greater than or equal to `key` (ceiling).
- **rank(Key key)**
  Return the number of keys strictly less than `key`. This can be done by traversing and counting.
- **select(int k)**
  Return the key with rank `k` (i.e., the *k*-th smallest key). One approach is to collect keys in a structure (like an array), sort them, and return the *k*-th element. Alternatively, traverse and compare in order.
- **deleteMin(), deleteMax()**
  Identify and remove the minimum or maximum key. This requires a full scan to find the min or max, then remove it from the list.
- **size(Key lo, Key hi)**
  Return the count of keys within the range `[lo, hi]`. This is another full scan, checking each key's ordering relative to `lo` and `hi`.
- **keys(), keys(Key lo, Key hi)**
  Return an iterable collection of all keys, optionally within a specified range. Again, you would likely gather keys in a separate data structure (like a list/array), filter or sort them, then return an iterable.

2. Implementation Strategy (Conceptual)

1. **Node Class (Conceptual Structure)**

   - Each node stores:

     ```
     Key key;
     Value value;
     Node next;
     ```

   - This can be adapted to any language, but the essence is the same: each node references a single key, its associated value, and the next node.

2. **Maintaining the List**

   - Typically, maintain a single reference (`head`) to the first node in the linked list.
   - The `isEmpty()` method simply checks if `head` is `null` (or if size is 0 in some

implementations).

   - The `size()` can be tracked in a variable that is updated upon insertion or deletion, or computed by traversing the list if you prefer simplicity over performance.

3. **Method Details**

   - **put(Key key, Value value):**
     a. Traverse the list.
     b. If `key` is found, update its `value`.
     c. If you reach the end without finding `key`, create a new node at the beginning (or end) of the list.
     d. Increment size if a new node is created.
   - **get(Key key):**
     a. Traverse the list from `head`.
     b. If `key` is found, return its `value`.
     c. If you reach the end, return `null`.
   - **delete(Key key):**
     a. Special case: If `head` is the node to delete, adjust `head`.
     b. Otherwise, keep track of the previous node while traversing.
     c. When the node with the matching `key` is found, adjust links to remove it from the chain.
     d. Decrement size.

4. **Ordered Operations**
   Even though the list is not kept in sorted order, you can still implement the methods requiring comparisons (`min`, `max`, `floor`, `ceiling`, etc.) by scanning all nodes and keeping track of the best candidate key seen so far.

5. **Complexities**

   - All operations that involve searching, inserting, or deleting can take up to (O(n)) time in the worst case, where (n) is the number of key–value pairs.
   - For ordered methods (e.g., `min`, `max`, `floor`, `ceiling`), the worst-case time complexity is also (O(n)).
   - This implementation is straightforward but not efficient for large datasets, as it relies on linear scans.

3. Suggested Steps for the Project

1. **Define the Node Structure**

- Create a conceptual node class/struct to hold `key` , `value` , and a pointer/reference to the next node.

2. **Initialize the Symbol Table**
   - Keep a reference (`head`) to the first node, initially `null` .
   - Optionally track the size in a separate variable for quick access.
3. **Implement Basic Methods**
   - **isEmpty** and **size**: Straightforward checks.
   - **get**: Linear search.
   - **put**: Linear search to update or create a new node.
   - **delete**: Linear search to find and remove the node.
4. **Implement Ordered Methods**
   - **min, max**: Single pass to find the smallest/largest.
   - **floor, ceiling**: Single pass to track the key that meets the condition.
   - **rank, select**: Possibly collect keys, sort them, and then find the rank or the key. Alternatively, do a naive linear approach.
   - **deleteMin, deleteMax**: Identify min/max in one pass, then delete in another pass (or do it in a single pass carefully).
   - **keys(lo, hi)**: Gather all keys, filter by range, sort if necessary, and return them as an iterable.
   - **keys()**: Gather all keys in a list/array, sort them, and return them as an iterable.
5. **Test the Implementation**
   - Create test cases that check boundary conditions (e.g., empty table, single element table) and typical usage scenarios.
   - Ensure correctness of ordered operations by verifying the returned values or lists match expected results.
6. **Optional Enhancements**
   - **Performance Tracking**: Measure how time grows with the number of key–value pairs.
   - **Exception Handling**: Decide how to handle invalid calls (e.g., calling `min` on an empty table).
   - **Refactoring**: Consider if a sorted data structure (like a BST) might be more efficient for large input sizes.

Implementation (Language-Agnostic)

- While not providing the code here, your final project should have a complete implementation in the programming language of your choice (e.g., Java, Python, C++), strictly following the described interface and method semantics.

Why Sequential Search?

Despite its (O(n)) search and insert times, a sequential search symbol table is an excellent starting point for understanding symbol table APIs and how to manage key–value data. It clearly illustrates fundamental operations, how to handle edge cases, and paves the way for more advanced data structures (e.g., Binary Search Trees, Red-Black Trees, or Hash Tables) that improve performance at scale.