1. Project Description: Binary Search Symbol Table

A **Binary Search Symbol Table** (often called a Binary Search ST) maintains keys in a **sorted array** (or list) and uses **binary search** to achieve logarithmic-time lookups. Unlike a Sequential Search ST (which scans through all elements in (O(n)) time), a Binary Search ST leverages the sorted nature of the keys to perform operations in (O(\log n)) time for search and rank-related queries. However, insertion and deletion in an array-based structure may take (O(n)) time in the worst case because elements might need to be shifted.

1.1 Overview of Data Structure

1. **Sorted Array**
    - Keep two parallel arrays or a single array of (key, value) pairs.
    - The keys are always in **sorted order**.
2. **Binary Search**
    - Implement a helper method (often called `rank(Key key)`) that returns the number of keys less than the given key by performing a binary search.
    - The rank can then be used for searching, insertion, or other ordered operations.
3. **Insertion (put)**
    1. Use `rank(key)` to find the index where the key would appear in sorted order.
    2. If the key already exists at that index, update its value.
    3. Otherwise, shift all elements from that index to the right by one to make room for the new entry, insert the (key, value) pair, and increment the size.
4. **Search (get)**
    1. Use `rank(key)` to find the position.
    2. Check if the key at that position matches `key`. If yes, return the corresponding value. Otherwise, return `null` (or the equivalent).
5. **Deletion (delete)**
    1. Use `rank(key)` to find the position.
    2. If the key is present, shift all elements to the left to fill the gap, decrement the size.

6. **Additional Methods**

   - **contains(Key key):** Check if `get(key)` is not `null`.
   - **isEmpty():** Return `true` if size is zero.
   - **size():** Return the number of key–value pairs.
   - **min(), max():** Return the smallest or largest key by accessing the first or last element in the array.
   - **floor(Key key), ceiling(Key key):** Use binary search logic to find the largest key (\le) `key` (floor) or the smallest key (\ge) `key` (ceiling).
   - **rank(Key key):** Returns the number of keys strictly less than `key` via binary search.
   - **select(int k):** Return the key with rank `k` (the ((k+1))-th smallest key) by accessing the sorted array at index `k`.
   - **deleteMin(), deleteMax():** Remove the first or last element in the array.
   - **size(Key lo, Key hi):** Count how many keys lie within `[lo, hi]` (this can be found by using `rank(hi+1) - rank(lo)` or similar logic).
   - **keys(), keys(Key lo, Key hi):** Return all keys (or keys in a range) in sorted order by iterating over the array.

1.2 Complexity

- **get, contains, rank, floor, ceiling, select**:
  (O(\log n)) time, thanks to binary search.
- **put, delete**:
  (O(n)) in the worst case (due to shifting elements in the array).
- **min, max**:
  (O(1)) if you maintain the array properly.
- **keys, keys(lo, hi)**:
  (O(n)) or (O(k)) for the number of keys in the range.

1.3 Steps to Implement

1. **Maintain a Dynamic Array** (in Java, you can use an array that resizes; in Python, a list can grow automatically).
2. **Implement rank(key)** to perform a binary search.
3. **Implement put(key, val)**:
   - Use `rank(key)` to find insertion point.
   - If the key is already at that index, update. Otherwise, shift and insert.
4. **Implement get(key)** using `rank(key)`.

5. **Implement delete(key)** by shifting elements left if found.
6. **Implement ordered methods** (min, max, floor, ceiling, etc.) by leveraging sorted array indexing and partial binary searches.

1.4 Testing

- **Boundary Conditions**: empty table, single-element table, etc.
- **Correctness of rank-based methods**: check that `rank(key)` is correct for keys both in and out of the table.
- **Insertion**: check that new keys are placed in the correct sorted order.
- **Deletion**: ensure the array updates properly.
- **Range Queries**: verify that `keys(lo, hi)` returns exactly the keys in `[lo, hi]`.