Exercise 1: Hashing - Collision Resolution with Open Addressing

(a) Let \( h(s, j) := h_1(s) - 2j \mod m \) and let \( h_1(x) = x + 2 \mod m \). Insert the keys 51, 13, 21, 30, 23, 72 into the hash table of size \( m = 7 \) using linear probing for collision resolution (the table should show the final state).

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
</table>

(b) Let \( h(s, j) := h_1(s) + j \cdot h_2(s) \mod m \) and let \( h_1(x) = x \mod m \) and \( h_2(x) = 1 + (x \mod (m - 1)) \). Insert the keys 28, 59, 47, 13, 39, 69, 12 into the hash table of size \( m = 11 \) using the double hashing probing technique for collision resolution. The hash table below should show the final state.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
</table>

(c) Repeat part (a) using the “ordered hashing” optimization from the lecture.

(d) Repeat part (b) using the “Robin-Hood hashing” optimization from the lecture.

Exercise 2: Amortized Analysis - Stack with Multipop

Consider the data structure “stack” in which elements can be stored in a last in first out manner. For a stack \( S \) we have the following operations:

- \( S.push(x) \) puts element \( x \) onto \( S \).
- \( S.pop() \) deletes the topmost element of \( S \). Assume \( pop() \) is only called if \( S \) is nonempty.
- \( S.multipop(k) \) removes the \( k \) top objects of \( S \), popping the entire stack if \( S \) contains fewer than \( k \) objects.

Assume the costs of \( S.push(x) \) and \( S.pop() \) are 1 and the cost of \( S.multipop(k) \) is \( \min(k, |S|) \) where \( |S| \) is the current number of elements in \( S \).

Use the bank account paradigm to show that all three operations have constant amortized cost. Assume that \( S \) is initially empty.
Consider the following data structure. We define arrays \( A_i \) (for \( i = 0, 1, 2, \ldots \)), where \( A_i \) has size \( 2^i \) and stores integer keys in a sorted manner (ascending). During the runtime we ensure that each Array is either completely full, or completely empty.

We informally describe an operation \( \text{insert}(k) \). It first tries to insert the key \( k \) into \( A_0 \). If \( A_0 \) is empty we insert \( k \) into \( A_0 \) and are done. If \( A_0 \) happens to be already full (i.e. it contains one element), \( A_0 \) is merged with \( k \) to form a new sorted array \( B_1 \) of size 2. If \( A_1 \) is empty, \( B_1 \) becomes the new Array \( A_1 \) and we are done. Else \( B_1 \) is merged with \( A_1 \) into a sorted Array \( B_2 \) of size 4 and the same procedure is repeated with \( A_2, A_3, \ldots \) until we find an Array \( A_i \) that is empty.

(a) Describe a subprocedure \( \text{merge}(A, B) \) (as pseudo code or as informal algorithm description) that merges the contents of two sorted Arrays \( A, B \) of size \( m \) into a new, sorted array of size \( 2m \) in \( \mathcal{O}(m) \) runtime. Explain why your algorithm has the runtime \( \mathcal{O}(m) \).

(b) Show that any series of \( n \) \( \text{insert} \)-operations has an amortized runtime of at most \( \mathcal{O}(\log n) \).