Problem 1

Modify the definition of a 2-3 tree so that it supports the following operations with the specified running times. You may assume that the reader understands the standard definition of a 2-3 (the one given in class, with all the values in the leaves); all you have to describe are the modifications that need to be made.

Note that we want a single (compound) data structure that supports all these operations, not different data structures for each operation.

- **add(x)**: Add element x to the set. Time: $O(\log(n))$
- **delete(x)**: Delete element x from the set. Time: $O(\log(n))$.
- **element?(x)**: Is x in the set? Time: $O(\log(n))$.
- **index(i)**: Find the ith smallest element in the set. Time: $O(\log(n))$.
- **indexOf(x)**: Find the index of x in the set. Time: $O(\log(n))$.
- **subrange(i,k)**: Return k elements in the set in sequence starting with the ith. For instance, `subrange(100,5)` should return a list of the 100th, 101st, 102nd, 103rd, and 104th smallest elements. Time: $O(k + \log(n))$.
- **min()**: Find the smallest element in the set. Time: $O(1)$.
- **max()**: Find the largest element in the set. Time: $O(1)$.
- **median()**: Find the median element in the set. For instance if there are 99 or 100 elements in the set, return the 50th. Time: $O(1)$.

Problem 2

Suppose that we have a hash table of size 23 and we insert the keys 10, 39, 4, 27, 17, 1, 62, 33, 48. Show the final state of the hash table, assuming we use:

A. Chaining, with the hash function $h(k) = k \mod 23$.
B. Linear probing, with hash function $h(k, i) = (k + i) \mod 23$
C. Double hashing, with hash function $(k + i \ast (1 + k \mod 17)) \mod 23$

(Assume 0-based indexing in the hash table, and assume that $i$ is initially 0 on the first probe and increases by 1 afterward.)
Problem 3

Suppose that you have a set of \( n \) large, orderable, objects, each of size \( q \), so that it requires time \( \Theta(q) \) to time to compute a hash function \( h(x) \) for any object. Describe a compound data structure, built out of a heap and a hash table, that supports the following operations with the specified run times.

- **elt(x)** — Is \( x \) an element of the set? Expected run time \( O(q) \).
- **add(x)** — Add \( x \) to the set. Expected run time \( O(q + \log n) \).
- **delete(x)** — Delete \( x \) from the set. Expected run time \( O(q + \log n) \). Here, note that, in a heap, when you replace \( x \) by the last element \( y \), \( x \) may be either greater than \( y \) or less than \( y \) and your algorithm has to consider both cases.
- **min(x)** — Return the minimum element in the set. Worst case run time \( O(1) \).

Problem 4

(Modified version of problem set 2, problem 3). Consider the following problem. The input is a sequence of \( n \) records with an integer key \( x.key \). You wish to sort the records in increasing order by key. The value of the key has many duplicates, so that the number \( w \) of distinct integers among the keys is much smaller than \( n \). In fact, assume that \( w \log w \ll n \).

A. Describe an algorithm to sort the sequence that runs in worst case time \( O(n \log w) \).

B. Describe an algorithm to sort the sequence that runs in expected time \( O(n) \).