```java
/**
 * Name: Kevin Shen
 * EECS Account: kshen94
 * Student ID: 212298535
 */

public class Question1 {

    public partA(int[] S, int[] T, int k){
        int n = s.length;
        sCount;     //Counter for each array
        tCount;
        kCount = 0;         //keep track of element number
        int search;
        if(k < n){          //if k is less than half the list, start from the beginning
            sCount =0;
            tCount =0;
            search = k-1;
            while(sCount != n+1 && tCount !=n+1){       //iterate through both arrays at the same time
                if(S[sCount] = T[tCount]){      //if 2 numbers are equal, increment them both
                    sCount++;                   //and increment search counter by 1
                    tCount++;
                    kcount++;
                }
                else if(S[sCount] > T[tCount]){ //if S[] is greater than T[]
                    if(kCount == search)        //if found return value
                        return S[sCount];
                    kCount++;                   //increment S
                    sCount++;
                }
                else{
                    if(kCount == search)        //if T[] is greater than S[]
                        return S[sCount];       //if found return value
                    kCount++;                   //increment T
                    tCount++;
                }
            }
        }else{                  //if k is greater than n, start from the end of the list
            sCount = n-1;
            tCount = n-1;
            search = 2*n -k;
            while(sCount != -1 && tCount != -1){        //iterate through both arrays at the same time
                if(S[sCount] = T[tCount]){      //if 2 numbers are equal, decrement them both
                    sCount--;                   //and increment search counter by 1
                    tCount--;
                    kcount++;
                }
                else if(S[sCount] > T[tCount]){ //if S[] is greater than T[]
                    if(kCount == search)        //if found return value
                        return S[sCount];
                    kCount++;                   //decrement S
                    sCount--;
                }
                else{
                    if(kCount == search)        //if T[] is greater than S[]
                        return S[sCount];       //if found return value
                    kCount++;                   //decrement T
                    tCount--;
                }
            }



        }
    }

}
/**
 * This method works by checking whether k is greater or less than n.
 * If it is less than n, it will search from the beginning of the arrays.
 * If it is greater than n, it will search from the end of the arrays.
 *
 * A) k=6, Output = 18, k=10, Output = 41.
 * These two cases would be the average case time.
 * B) k=n, Output = 41.
 * This case would be the worst case time.
 **/

    public int partC(Map S, Map T, int k){
        n = s.length -1;
        if(k < n){                                              //checks whether k is less than n
            while(k>0){                                         //starts from beginning of both maps
                remove lesser of(S.first, T.first);             //removes the lesser if the 1st key until k is found
                using table.remove(0);
                k--;
            }
            return lesser of (S.first, T.first);

        }
        else{
            k = 2*n -k;                                         //if larger than n, adjust to count from the end
            while(k >0){
            remove larger of (S.last, T.last);                  //removes larger of the last key of both maps until k is found
            using table.remove(size());
            k--;
            }
            return larger of (S.last,T.last);
        }
    }

    /**
     * By removing the values of the 2 maps, the operation takes O(1).
     * Searching for k takes O(log n) because only half the map is searched
    **/
```

```java
/**
 * Name: Kevin Shen
 * EECS Account: kshen94
 * Student ID: 212298535
 */

public class Question2 {
        public countRange(k1,k2){
            count = tree.size;
            Node left = tree;
            Node right;
            while(left != null){                //traverse left side of tree
                if(left == k1){                 //if key is found, subtract size by left subtree
                    count -= left.left.size;
                }
                if(left.left >k1 )              //keep going left until a smaller key is found
                    left = left.left;
                else if{                        //when smaller key is found
                    count -= left.left.size;    //subtract the size by left subtree
                    left = left.right           //traverse right
                }
            }
        }
            while(right != null){
                //do the same as above while loop
                //but mirrored for right side

            }
        return count;
}

/**
*   This is in O(h) time because each side does a comparison once per height level
*   until the key or a null is found. So worst case is accessing a node h times.
*
*   For the insert operation, a height check can be done as the searching recurses
*   back up the tree using size= Max(left,right)+1
*   This should keep the running time about the same.
*
*   For the delete operation, a height check can be done from the deleted node.
*   By using size= Max(left,right)+1 on each ancestor in the tree from the deleted node,
*   The worst case for this additional operation is O(h).
*   So O(n) + O(h) = O(n)
**/
```

```java
/**
 * Name: Kevin Shen
 * EECS Account: kshen94
 * Student ID: 212298535
 */

public class Question3 {

    public int question3(int[] S, int k){
        int maxVotes;
        int maxIndex;
        int[] votes = new int[k];
        for(int vote: S){                   //counts votes similar to bucket sort
            votes[vote-1]++;                //instead of sorting votes in buckets, it increments a counter in an array
        }
        for(int v: votes){                  //goes through buckets to find the largest amount of votes
            compare each v;
            maxVotes and maxIndex record largest amount of votes;
        }
    }
    return maxIndex+1;
}

/**
 * Using something similar to bucket sort, it takes O(n) to obtain all the votes.
 * Then searching for the largest amount of votes take O(k).
 * O(n) + O(k) = O(n+k).
 */
```