Diffchecker

- 94 Removals + 85 Additions

1.java

```
1  public class IntListMethods {
2      public static String getAuthorName() {
3          return "Yap, Calvin"; // This is my name
4      }
5
6      public static String getRyersonID() {
7          return "500825267"; // This is my Student id
8      }
9
10     public static int removeIfDivisible(int n, int k) {
11         int temp = 0; // to keep for switching
12         int first = n; // keep reference to first element
           int counter = 0; // counter to see when first elemen
13 t is removed or not
14         if(n == 0){
15             return  0; // list empty
16         }
17         while (IntList.getNext(n) != 0) {
18             if(IntList.getKey(n) % k != 0){
19                 counter++;
20             }
21             if (counter < 1 && IntList.getKey(n) % k == 0) {
22                 temp = IntList.setNext(n, 0);
23                 IntList.release(n);
24                 n = temp;
25                 first = n;
26             } else if (counter >= 1 && IntList.getNext(n) !=
   0) {
27                 int temp1 = IntList.getNext(n);
28                 if (IntList.getKey(temp1) % k == 0) {
29                     IntList.setNext(n, IntList.setNext(temp
   1, 0));
30                     IntList.release(temp1);
31                     temp1 = IntList.getNext(n);
32                 } else {
33                     n = temp1;
34                     temp1 = IntList.getNext(temp1);
35                 }
36             }
37         }
38         return first;
39     }
40     static int[] arrNodes;
41     public static int sort(int n) {
42         if(n == 0 || IntList.getNext(n) == 0){
43             return n;
44         }
45
   int arrKeys[]= new int [IntList.getAllocatedNodeCount()]; //
   new array to hold keys
46         arrNodes = new
```

```
1  public class IntListMethods {
2      public static String getName() {
3          return "John Smith";
4      }
5
6      public static String getBrockID() {
7          return "4862154";
8      }
9
10     public static int removeIfDivisible(int n, int k) {
11         int temp = 0;
12         int first = n;

13         int counter = 0;
14         if (n == 0) {
15             return 0;
16         }
17         while (IntList.getNext(n) != 0) {
18             if (IntList.getKey(n) % k != 0) {
19                 counter++;
20             }
21             if (counter < 1 && IntList.getKey(n) % k == 0) {
22                 temp = IntList.setNext(n, 0);
23                 IntList.release(n);
24                 n = temp;
25                 first = n;
26             } else if (counter >= 1 && IntList.getNext(n) !=
   0) {
27                 int temp1 = IntList.getNext(n);
28                 if (IntList.getKey(temp1) % k == 0) {
29                     IntList.setNext(n, IntList.setNext(temp
   1, 0));
30                     IntList.release(temp1);
31                     temp1 = IntList.getNext(n);
32                 } else {
33                     n = temp1;
34                     temp1 = IntList.getNext(temp1);
35                 }
36             }
37         }
38         return first;
39     }

40     public static int sort(int n) {
41         if (n == 0 || IntList.getNext(n) == 0) {
42             return n;
43         }
44
    int keys[] = new int[IntList.getAllocatedNodeCount()];

45         int[] nodes = new
   int[IntList.getAllocatedNodeCount()];
```

```
int[IntList.getAllocatedNodeCount()]; // new array to hold n
odes
      for(int x = 0; x < arrNodes.length;x++) { // for loop
to fill contents
       arrKeys[x] = IntList.getKey(n);
       arrNodes[x] = n;
       n = IntList.getNext(n);
       }
       buildHeap(arrKeys);// implement the sort
       int count = 0;
       for(int i = 0; i < arrNodes.length-1; i++){
           IntList.setNext(arrNodes[i], arrNodes[i+1]);
           count++;
       }

IntList.setNext(arrNodes[count], 0); // building chain
       return arrNodes[0]; // returning first element of th
e chain
    }

    public static void buildHeap(int keys[])
    {
        int GetfloorDiv =2;
        int heapSize = keys.length;
        // Build heap (rearrange array)
        for (int i = heapSize / GetfloorDiv - 1; i >= 0;
i--) // set floor down to 1
            heapify(keys, heapSize, i); // heapify

        int i=heapSize-1;
        while (i>=0) // extracting elements
        {
            int grabber = keys[0]; //
            keys[0] = keys[i]; // swapping current node
            keys[i] = grabber;  // to end

            int grabberNodes =
arrNodes[0]; // moving array for noedes
            arrNodes[0] = arrNodes[i]; // swapping for curre
nt node
            arrNodes[i] = grabberNodes; // to end

            heapify(keys, i,
0); // removing elements in heap to make it smaller
            i--;
        }
    }
    public static void heapify(int keys[], int length, int i
ndex) {   // don't have to subtract 1 because of previously
 did in for loop
        int LgRoot = index;  // largest element
        int left = 2 * index + 1; // left
        int right =left + 1 ; //right

    if (left < length && keys[left] > keys[LgRoot]) { //left
Child is larger than largest
            LgRoot = left;
```

```


        for (int x = 0; x < nodes.length; x++) {
            keys[x] = IntList.getKey(n);
            nodes[x] = n;
            n = IntList.getNext(n);
        }
        createHeap(keys, nodes);
        int count = 0;
        for (int i = 0; i < nodes.length - 1; i++) {
            IntList.setNext(nodes[i], nodes[i + 1]);
            count++;
        }
        IntList.setNext(nodes[count], 0);



        return nodes[0];
    }

    public static void createHeap(int keys[], int[] nodes) {
        for (int i = keys.length / 2 - 1; i >= 0; i--)
            heapify(keys, keys.length, i, nodes);









        int i = keys.length - 1;
        while (i >= 0) {
            int grabber = keys[0];
            keys[0] = keys[i];
            keys[i] = grabber;


            int grabberNodes = nodes[0];



            nodes[0] = nodes[i];
            nodes[i] = grabberNodes;

            heapify(keys, i, 0, nodes);


            i--;
        }
    }









public static void heapify(int keys[], int len, int idx, int
[] nodes) {
        int root = idx;
        int l = 2 * idx + 1;
        int r = l + 1;

        if (l < len && keys[l] > keys[root]) {
```

```
92          }
93          if (right < length && keys[right] >
            keys[LgRoot]){ // right Child is larger than largest
94              LgRoot = right;
95          }
96          if (LgRoot != index) // not equal
97          {
98              int swappingLg = keys[index]; // shift elements
99              keys[index] = keys[LgRoot];
100             keys[LgRoot] = swappingLg;
101
102             int swapNode = arrNodes[index]; // shifts nodes
103             arrNodes[index] = arrNodes[LgRoot];
104             arrNodes[LgRoot] = swapNode;
105
106             heapify(keys, length, LgRoot);
107         }
108     }
109 }
```

```
86              root = l;
87          }
88          if (r < len && keys[r] > keys[root]) {

89              root = r;
90          }
91          if (root != idx) {
92              int swappingLg = keys[idx];
93              keys[idx] = keys[root];
94              keys[root] = swappingLg;

95
96              int swapNode = nodes[idx];
97              nodes[idx] = nodes[root];
98              nodes[root] = swapNode;
99
100             heapify(keys, len, root, nodes);
101         }
102     }
103 }
```