PSH: Algorithmic Trading 1

by JeremyBlum

Watch Kevin Salvin's TED talk on algorithmic trading.

The algorithm compares the median of the last *m* sales prices with the last price. If this median is greater than the last price, the algorithm recommends that you sell the stock; if the median is equal to the last price, the algorithm recommends that you hold; and if the median is less than the last prices, the algorithm recommends that you buy.

Input Format

The first line of input consists of two integer *n* and *m*, $3 \le n \le 5 * 10^5$ and $3 \le m \le n$.

n indicates the number of prices in the input, and *m* indicates how many of the most recent values should be considered when calculating the median.

The remaining *n* lines give a sequence of prices.

Constraints

m will be odd.

Output Format

Starting with the m^{th} price, your program should output, on a line by itself, `buy`, `hold`, or `sell`, based on the comparison of the median of the last *m* prices and the most recent price.

Note: You will output *n* - *m* + 1 values

Sample Input

```
5 3
10
12
8
9
11
```

Sample Output

```
sell
hold
buy
```

Explanation

For the first decision, the program calculates the median of <10, 12, 8> which is 10, and compares this with the last price 9. Since the median is greater than the last price, the program outputs "sell".

For the second decision, the program calculates the median of <12, 8, 9> which is 9, and compares this with the last price 9. Since the median is equal to the last price, the program outputs "hold".

For the third decision, the program calculates the median of <8, 9, 11> which is 9, and compares this with the last price 11. Since the median is less than the last price, the program outputs "buy".

Current Buffer (saved locally, editable) Java 8

```
import java.io.*;
import java.util.*;
import java.text.*;
import java.math.*;
import java.util.regex.*;

public class Solution {

    public static void main(String[] args) throws IOException {
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

        String[] tokens = reader.readLine().split(" ");
        int n = Integer.valueOf(tokens[0]);
        int m = Integer.valueOf(tokens[1]);
        Queue<Integer> oldPriceQueue = new LinkedList<>();
        int lastPrice1 = Integer.valueOf(reader.readLine());
        BSTNode tree = new BSTNode(lastPrice1);
        oldPriceQueue.add(lastPrice1);
        BSTNode record;
        for (int i = 0; i < n - 1; i++) {
            int lastPrice = Integer.valueOf(reader.readLine());
            tree = tree.insert(tree, lastPrice);
            oldPriceQueue.add(lastPrice);
            if (m == tree.size()) {
                int medianPrice = tree.getMedian(tree);
                if (medianPrice < lastPrice) {
                    System.out.println("buy");
                }
                if (medianPrice > lastPrice) {
                    System.out.println("sell");
                }
                if (medianPrice == lastPrice) {
                    System.out.println("hold");
                }
                if (!oldPriceQueue.isEmpty()) {
                    int toDelete = oldPriceQueue.remove();
                    //System.out.println("We want to delete " + toDelete);
                    if (tree != null) {
                        tree = tree.delete(tree, toDelete);
                        //System.out.println("New root is: " + tree.getKey());
                    }
                }
            }


            // Do your magic here
        }
```

```
48      }
49  }
50  interface iMedianQueue<T extends Comparable<T>> {
51      BSTNode enqueue(BSTNode root, Integer value);
52      int size();
53      Integer first(BSTNode root);
54      BSTNode dequeue(BSTNode root, Integer value);
55      T median();
56  }
57
58  class medianQueue implements iMedianQueue<Integer> {
59      BSTNode tree = new BSTNode();
60
61      public BSTNode enqueue(BSTNode root, Integer value) {
62          return tree.insert(root, value);
63      }
64
65      public int size() {
66          return tree.size();
67      }
68
69      public Integer first(BSTNode root) {
70          return root.getKey();
71      }
72
73      public BSTNode dequeue(BSTNode root, Integer value) {
74          return tree.delete(root, value);
75      }
76
77      public Integer median() {
78          if (size() % 2 == 0) return null;
79          Integer medianValue = tree.getMedian(tree);
80          return medianValue;
81      }
82  }
83
84  class BSTNode {
85
86      private int key;
87      private int rightSubtreeSize;
88      private int leftSubtreeSize;
89      private BSTNode LC;
90      private BSTNode RC;
91
92      /**
93       * Create a new leaf node
94       *
95       * @param key the value at the leaf
96       */
97      public BSTNode(int key) {
98          this.key = key;
99      }
100
101     public BSTNode() {
102
103     }
104
105     public int getKey() {
106         return key;
107     }
108
109     public BSTNode getLC() {
110         return LC;
111     }
112
113     public BSTNode getRC() {
114         return RC;
115     }
```

```
116
117     public int size() {
118         return leftSubtreeSize + rightSubtreeSize + 1;
119     }
120
121     public int getMedian(BSTNode root) {
122         if (root.LC == null && root.RC == null) {
123             return root.key;
124         } else {
125             return helper(root, 0, 0);
126         }
127     }
128
129     public int helper(BSTNode node, int lSubSize, int rSubSize) {
130         if (node == null) {
131             return 0;
132         }
133         if (node.leftSubtreeSize + lSubSize == node.rightSubtreeSize + rSubSize) {
134             return node.key;
135         }
136         if ((rSubSize - lSubSize + node.rightSubtreeSize - node.leftSubtreeSize) > 0) {
137             return helper(node.RC, node.leftSubtreeSize + lSubSize + 1, rSubSize);
138         } else {
139             return helper(node.LC, lSubSize, node.rightSubtreeSize + rSubSize + 1);
140         }
141     }
142
143
144     /**
145      * Search for a value in a BST
146      *
147      * @param node the root of the BST
148      * @param x    the value
149      * @return true iff the BST contains x
150      */
151     public static boolean search(BSTNode node, int x) {
152         if (node == null) {
153             return false;
154         }
155         if (x == node.key) {
156             return true;
157         } else if (x < node.key) {
158             return search(node.LC, x);
159         } else {
160             return search(node.RC, x);
161         }
162     }
163
164     public static BSTNode search2(BSTNode node, int x) {
165         if (node == null) {
166             return null;
167         }
168         if (x == node.key) {
169             return node;
170         } else if (x <= node.key) {
171             return search2(node.LC, x);
172         } else {
173             return search2(node.RC, x);
174         }
175     }
176
177     /**
178      * Insert a new value into a BST, if it is not already in the tree
179      *
180      * @param root the root of the BST
181      * @param x    the new value
182      * @return the root of the tree
183      */
184     public static BSTNode insert(BSTNode root, int x) {
```

```
185         // Handle duplicate nodes
186         if (root == null) {
187             root = new BSTNode(x);
188         }
189         else {
190             BSTNode node = root;
191             while (node != null) {
192                 if (x <= node.key) {
193                     node.leftSubtreeSize++;
194                     if (node.LC == null) {
195                         node.LC = new BSTNode(x);
196                         break;
197                     }
198                     else {
199                         node = node.LC;
200                     }
201                 } else {
202                     node.rightSubtreeSize++;
203                     if (node.RC == null) {
204                         node.RC = new BSTNode(x);
205                         break;
206                     }
207                     else {
208                         node = node.RC;
209                     }
210                 }
211             }
212         }
213         //System.out.println("From insert, current root is: " + root.key + " with a LSS of " +
    root.leftSubtreeSize + " and a RSS of " + root.rightSubtreeSize);
214         return root;
215     }
216 
217     /**
218      * Remove a value x from a BST
219      *
220      * @param root the root of the BST
221      * @param x    the value, which must be in the tree
222      * @return the root of the tree
223      */
224     public static BSTNode delete(BSTNode root, int x) {
225         BSTNode node = root;
226         BSTNode parent = null;
227         /*while (node != null && node.key != x){// && search(node, x)) { //Traversing the tree to find node
    to delete.
228             parent = node;
229             if (x < node.key) {
230                     node.leftSubtreeSize--;
231                 if (node.LC != null) {
232                     node = node.LC;
233                 }
234                 else {
235                     break;
236                 }
237             } else {
238                     node.rightSubtreeSize--;
239                 if (node.RC != null) {
240                     node = node.RC;
241                 }
242                 else {
243                     break;
244                 }
245             }
246         }*/
247         while (node != null && node.key != x) { //Traversing the tree to find node to delete.
248             parent = node;
249             if (x < node.key) {
250                 //if (node.key != root.key) {
251                 node.leftSubtreeSize--;
```

```
                node.leftSubtreeSize--;
                //}

                node = node.LC;
            } else {
                //if (node.key != root.key) {
                node.rightSubtreeSize--;
                //}
                node = node.RC;
            }
        }
        //System.out.println("BEFORE DELETE, the node is: " + node.key + " with a size of " + node.size());
        //System.out.println("BEFORE DELETE, the root is: " + root.key + " with a size of " + root.size());
        //System.out.println("LC of " + node.key + " is: " + node.LC.key + ". RC of " + node.key + " is: " +
node.RC.key);
        if (node.LC == null && node.RC == null) {
            // Case 1: the node containing x is a leaf
            if (parent == null) {
                    root = null;
            }
            else if (node == parent.LC) {

                    parent.LC = null;

            } else {
                    parent.RC = null;
            }
        } else if (node.LC == null || node.RC == null) {
            // Case 2: the node containing x has one child
            BSTNode child = (node.LC != null) ? node.LC : node.RC;
            if (parent == null) {
                    root = child;
            } else if (node == parent.LC) {
                    parent.LC = child;
                    //System.out.println("Current left child is: " + child.key);
            }
            else {
                    parent.RC = child;
                    //System.out.println("Current right child is: " + child.key);
            }
        } else {
            // Case 3: node has two children
                BSTNode predecessor = getLargestNode(node.LC);

                // Swap values
                int tmp = node.key;
                node.key = predecessor.key;
                predecessor.key = tmp;
                node.leftSubtreeSize--;
                //System.out.println("Recursive call on: " + node.LC.key + " and key of " + predecessor.key);
                node.LC = delete(node.LC, predecessor.key);
        }
        //System.out.println("From delete, current root is: " + root.key + " with a LSS of " +
root.leftSubtreeSize + " and a RSS of " + root.rightSubtreeSize);
        //System.out.println("From delete, current NODE is: " + node.key + " with a LSS of " +
node.leftSubtreeSize + " and a RSS of " + node.rightSubtreeSize);
        //System.out.println("From delete, the current tree size at the root of " + root.key + " is: " +
root.size());
        return root;
    }


    /**
     * Get the node with the largest value in a BST
     *
     * @param node the root of the BST
     * @return the node with the largest value
     * @throws IllegalArgumentException if the BST is empty
     */
```

```
316     public static BSTNode getLargestNode(BSTNode node) {
317         if (node == null) {
318             throw new IllegalArgumentException("Empty tree");
319         }
320
321         while (node.RC != null) {
322             node = node.RC;
323         }
324
325         return node;
326     }
327
328     public void print(String prefix) {
329         System.out.println(prefix + "key: " + key + " left subtree size: " + leftSubtreeSize + " right
    subtree size: " + rightSubtreeSize);
330         if (LC != null) LC.print(prefix + " ");
331         if (RC != null) RC.print(prefix + " ");
332     }
333
334     void printInorder(BSTNode node) {
335         if (node == null) {
336             return;
337         }
338
339         printInorder(node.LC);
340
341         System.out.print("(" + node.key +  ") LSS: " + node.leftSubtreeSize + " RSS: " +
    node.rightSubtreeSize + " ");
342
343         printInorder(node.RC);
344     }
345
346     void printPreorder(BSTNode node) {
347         if (node == null) {
348             return;
349         }
350         System.out.print("(" + node.key + ") LSS: " + node.leftSubtreeSize + " RSS: " + node.rightSubtreeSize
    + " ");
351         printInorder(node.LC);
352         printInorder(node.RC);
353     }
354 }
355
```

Line: 1 Col: 1

Upload Code as File Test against custom input Run Code Submit Code

Testcase 0 ✔

Congratulations, you passed the sample test case.

Click the **Submit Code** button to run your code against all the test cases.

Input (stdin)

```
5 3
10
12
8
9
11
```

Your Output (stdout)

```
sell
hold
buy
```

Expected Output

```
sell
hold
buy
```