UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

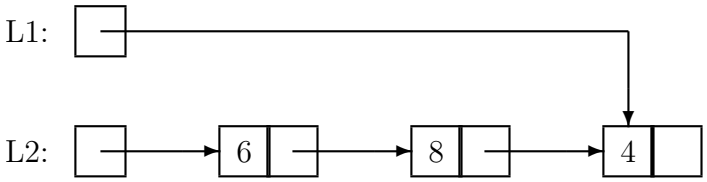
**CS61B** **P. N. Hilfinger**
**Fall 2016**

# Final Examination Solution (revised)

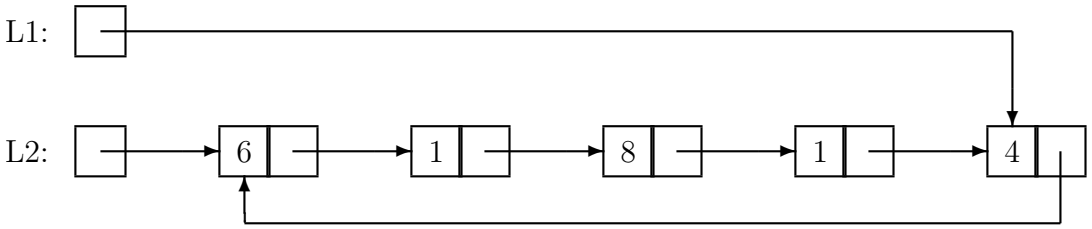
**1.** [4 points] Draw the box-and-pointer diagram corresponding to the state at each of the marked points (Q1, Q2, Q3, Q4) in the main program below. There are no runtime errors in executing the program.

```
public class Mystery {
    public static void mystery1(IntList L) {
        IntList front = L;
        for (; L.tail != null; L = L.tail.tail) {
            IntList one = new IntList(1, L.tail);
            L.tail = one;
        }
        L.tail = front;
    }

    public static void mystery2(IntList L) {
        L.tail.tail = new IntList(0, L.tail.tail);
    }

    public static void mystery3(IntList L1, IntList L2) {
        for (int i = 0; i < 3; i += 1, L1 = L1.tail, L2 = L2.tail) {
            int tempInt = L1.head;
            L1.head = L2.head;
            L2.head = tempInt;
            IntList tempIL = L1;
            L1 = L2;
            L2 = tempIL;
        }
    }

    public static void main(String[] args) {
        IntList L1 = new IntList(4, null);
        IntList L2 = new IntList(6, new IntList(8, L1));
        // Q1
        mystery1(L2);
        // Q2
        mystery2(L1);
        // Q3
        L1 = L2.tail.tail.tail;
        mystery3(L1, L2);
        // Q4
    }
}
```

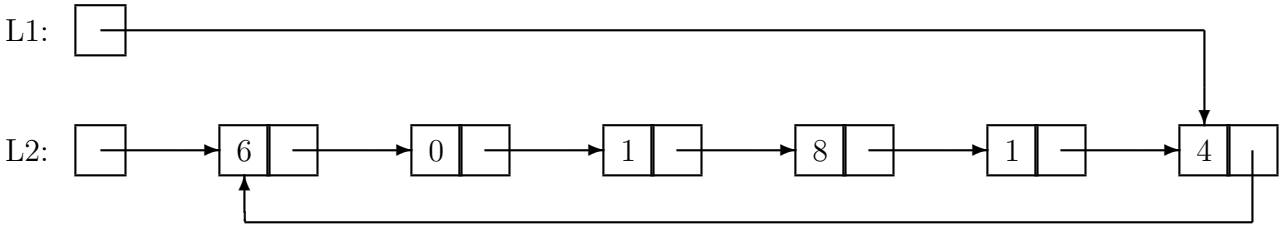
Q1.



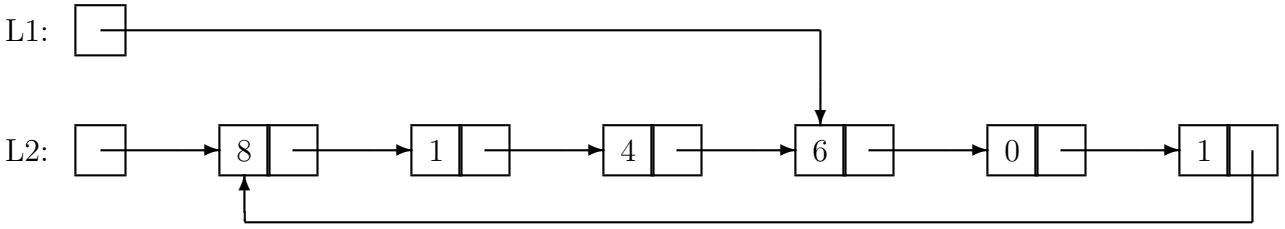

Q2.




Q3.




Q4.



**2.** [2 points] Fill in the return statement in the following to fulfill the comment.

```
/** Returns the result of rotating the right half (i.e., the least
 *  significant 16 bits) of X left by 1, leaving the left half  unchanged.
 *  Rotating a binary integer left means moving all but the most
 *  significant bit left by one bit position and moving the most
 *  significant bit to the least significant (units) position.  For example,
 *  (0b indicates binary notation; 0x indicates hexadecimal):
 *      rotateHalf(3) == rotateHalf(0b0011) == 0b0110 == 6
 *      rotateHalf(32769) == rotateHalf(0x8001) == rotateHalf(0b1000000000000001)
 *          == 0b0011 == 3
 *      rotateHalf(3309569) == rotateHalf(0x328001)
 *          == rotateHalf(0b11001010000000000000001)
 *          == 0b11001000000000000000011 == 0x320003 == 3276803
 */
 int rotateHalf(int x) {
     int mask = 0xffff;

     return (x & ~mask) | ((x >> 15) & 1) | ((x << 1) & mask)
;
}
```

**3.** [2 points] Fill in the blank in the following to fulfill the comment.

```
import java.util.regex.Pattern;

/** A Java pattern that matches a single numeral consisting of an optional
 *  sign followed by one or more digits, with groups of three digits
 *  separated by commas. For example, the pattern matches any of
 *      1      -1      23     +25     117    1,243    -20,176
 *  but not
 *      1,2    32,   1024    121,32   --75
 *
 */
static final Pattern NUMBER =

    Pattern.compile("[-+]?\\d\\d?\\d?(,\\d\\d\\d)*");
```

**4.** [4 points]

Below you will find intermediate steps in performing various sorting algorithms on the same input list. The steps do not necessarily represent consecutive steps in the algorithm (that is, many steps are missing), but they are in the correct sequence. For each of them, select the algorithm it illustrates from among the following choices: insertion sort, selection sort, mergesort, quicksort (first element of sequence as pivot), heapsort, LSD radix and MSD radix sort.

**Input List:**

```
1429, 3291, 7683, 1337,  192,  594, 4242, 9001, 4392,  129, 1000
```

a. LSD radix sort

```
1000, 3291, 9001,  192, 4242, 4392, 7683,  594, 1337, 1429,  129
1000, 9001, 1429,  129, 1337, 4242, 7683, 3291,  192, 4392,  594
1000, 9001,  129,  192, 4242, 3291, 1337, 4392, 1429,  594, 7683
```

b. Quicksort

```
1337,  192,  594,  129, 1000, 1429, 3291, 7683, 4242, 9001, 4392
 192,  594,  129, 1000, 1337, 1429, 3291, 7683, 4242, 9001, 4392
 129,  192,  594, 1000, 1337, 1429, 3291, 7683, 4242, 9001, 4392
```

c. Insertion sort

```
1337, 1429, 3291, 7683,  192,  594, 4242, 9001, 4392,  129, 1000
 192, 1337, 1429, 3291, 7683,  594, 4242, 9001, 4392,  129, 1000
 192, 594,  1337, 1429, 3291, 7683, 4242, 9001, 4392,  129, 1000
```

d. Heapsort

```
1429, 3291, 7683, 9001, 1000,  594, 4242, 1337, 4392,  129,  192
7683, 4392, 4242, 3291, 1000,  594,  192, 1337, 1429,  129, 9001
 129, 4392, 4242, 3291, 1000,  594,  192, 1337, 1429, 7683, 9001
```

In all these cases, the final step of the algorithm will be this:

```
129,  192,  594, 1000, 1337, 1429, 3291, 4242, 4392, 7683, 9001
```

**5.** [6 points] For each question below, respond in the best way that you can, based on the material you have learned in this class. Read each question carefully; the exact questions being asked will differ from scenario to scenario.

Brevity is the soul of wit. We reserve the right to deduct partial credit for overly lengthy answers.

a. Your friend, a budding politician, meets several hundred people a day and places their names onto the front of an `ArrayList`. Once there, he never removes a name, but sometimes looks through the list to see the order in which he met people.

At least one aspect of his procedures is slower than it could be. Describe a small change in your friend's use of the data structure that would improve runtime without changing the data structure involved. Briefly justify why your change speeds things up.

Putting items at the end of the list instead of the front would speed up addition to the list.

Now describe a change in the data structure that would improve runtime without requiring a change in actions taken. *Briefly* justify the speedup.

Using a linked list would allow adding to the front of the list in $O(1)$ time.

b. A `TopBalancedTree` class is a BST that (somehow) ensures the number of stored elements to the left of the root of the tree differs by no more than one from the number of elements to the right of the root. Your friend claims that this data structure has $O(\lg N)$ worst-case retrieval time for any element, because you only have to look at half of the elements in the data structure.

How long does the `.contains(·)` method *actually* take in the worst case for this data structure?

$\Theta(N)$ time in the worst case. Since we only balance at the root, the two subtrees can still be stringy.

Suggest a modification of the class that uses the same idea, but does achieve $O(\lg N)$ look-up time. You do not have to justify your answer.

Balance all the nodes, not just the root.

c. You are building a map server application that uses A* search to find shortest routes between points. Your friend has a suggestion for a suitable heuristic estimate from any point $A$ to destination $B$: collect crowd-sourced data from drivers in which they report the longest time it has ever taken them to travel between $A$ and $B$ (i.e., on the busiest of days) and use the average of these worst-case times as the heuristic. (Most people would find it tedious to keep track of that information, but fortunately there's an app for that.) What is wrong with your friend's suggestion? Be specific.

The heuristic can easily overestimate distance, making it inadmissible.

d. Why doesn't the fact that Radix Sort is a linear time sorting algorithm violate the $\Omega(N \lg N)$ lower bound which we established for the other sorts?

Apples and oranges. The $N$ in $\Omega(N \lg N)$ bound refers to the number of keys, whereas in radix sort, the bound is $\Omega(B)$, where $B$ is the number of bytes of data. Furthermore $\Omega(N \lg N)$ is a count of comparison operations, which radix sort is not restricted to comparisons.

e. You are told that in a certain application, insertion sort is guaranteed to be the best algorithm to sort a list of numbers used in that application. What might you infer about the characteristics of that list?

That it is nearly sorted already, or that it is always very short.

f. Consider the following class:

```
class Hasher {
    private Random gen = new Random();
    int hash(int y, int M) {
        int r = rotate(y, gen.nextInt(32)) >>> 1;
        return r % M;
    }
}
```

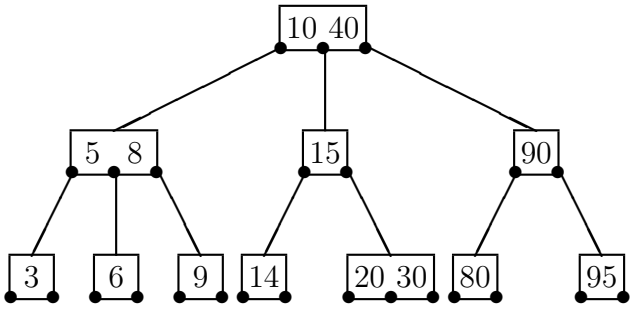
That is, `.hash` rotates the bits of `y` left by a random amount $< 32$, and then logically shifts the result right by one to get a non-negative number before taking modulo `M`. If `H` is an object of type `Hasher` is `H.hash(x, N)` good to use for hashed values of `x` (where `N` is the size of the table)? Briefly, why or why not?

`H.hash` won't work at all, since it produces different values for identical values of `x`.

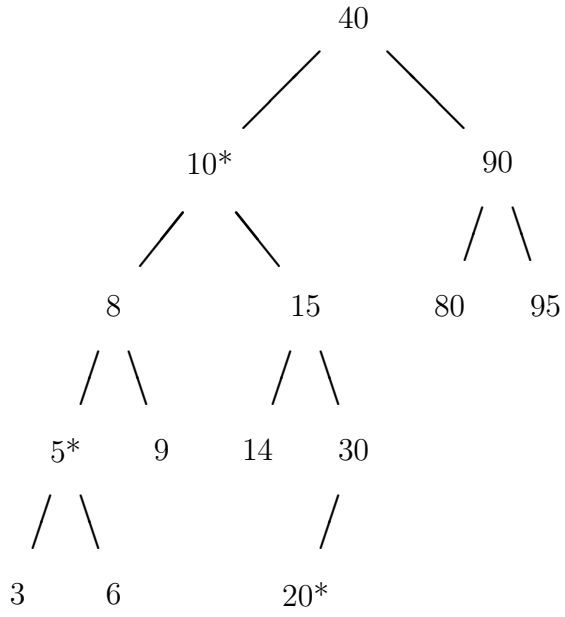
**6.** [6 points]

a. Show the left-leaning red-black tree that corresponds to the (2,3) tree on the left. Indicate red nodes with an asterisk (as in part (b) below).

**(2, 3) tree**

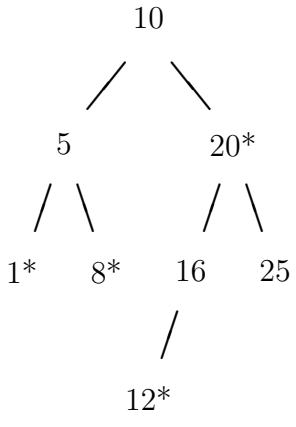



**Left-Leaning Red-Black Tree**

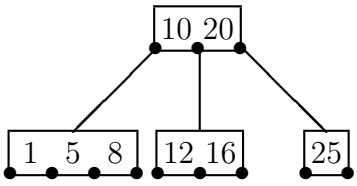



b. Show the (2,4) tree that corresponds to the red-black tree on the left. Red nodes are marked with an asterisk.

**Red-Black Tree**




**(2, 4) tree**



**7.** [6 points]

In the following, assume that `cnst()` is a constant-time function call. When giving bounds, give bounds that are as tight and simple as possible. For example if $\Theta(E + G)$ is a correct answer for some expressions $E$ and $G$, and $\Theta(E + G) = \Theta(E)$, then your answer should be $\Theta(E)$, since it is simpler. If $O(E)$ and $O(G)$ are both correct answers, $O(E) \subset O(G)$, but $O(E) \neq O(G)$, then your answer should be $O(E)$, since it is tighter.

a. Give best- and worst-case bounds as a function of $N$.

```
public static void foo1(int N) {
    if (N == 0) return;
    for (int i = N; i >= 1; i = i/2) {
        cnst();
    }
    foo1(N - 1);
}
```

Best case: $\Theta($*N lg N*$)$ Worst case: $\Theta($*N lg N*$)$

b. Give best- and worst-case runtime bounds for the call `foo2(N, N)` as a function of $N$.

```
public static void foo2(int i, int N) {
    if (N == 0) return;
    for (int j = 0; j < i; j = j + 1) {
        cnst();
    }
    if (i > N/2) {
        foo2(i - 1, N);
    } else {
        foo2(i/2, N) + foo2(i/2, N);
    }
}
```

Best case: $\Theta($*N*$^2$$)$ Worst case: $\Theta($*N*$^2$$)$

c. True or false: if $f(N) \in O(N)$ and $g(N) \in O(N^2)$, and both functions are non-negative, then $|g(N) - f(N)| \in \Omega(N)$. If true, explain why; otherwise give a counterexample.

False: Consider $f(N) = g(N) = N$.

d. True or false: if $f(N) \in \Theta(N)$ and $g(N) \in \Theta(N^2)$, and both functions are non-negative, then $|g(N) - f(N)| \in \Omega(N)$. If true, explain why; otherwise give a counterexample.

True: Since $g(N) \in \Theta(N^2)$, it is also in $\Omega(N)$, while $f(N) \in O(N)$. Since $g$ grows at least as $N^2$, an $O(N)$ quantity is eventually negligible in comparison.

e. What is the running time of the following algorithm, as a function of the parameter $r$?

```
/** Assumes that VALS is a square array, and that 0 <= R, C < vals.length.
 */
double best(double vals[][], int r, int c) {
    if (r == 0)
        return vals[r][c];
    double v;
    v = best(vals, r - 1, c);
    if (c > 0)
        v = Math.max(v, best(vals, r - 1, c - 1);
    if (c < vals[r].length - 1)
        v = Math.max(v, best(vals, r - 1, c + 1);
    return v + vals[r][c];
}
```

Answer: $\Theta(3^r)$

f. Suppose that the function in part (e) is memoized. Give an expression $E$ involving the parameter $r$ and $N =$`vals.length` such that the running time of the memoized function is $\Theta(E)$.

Answer: $E = rN$

**8.** [4 points]

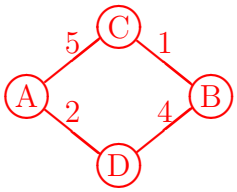
Indicate whether the following statements are true or false and briefly justify your answer (if false, a counterexample is sufficient justification).

a. If all edge weights are equal and positive, breadth-first search starting from node $A$ will return the shortest path from a node $A$ to a target node $B$.

True. Breadth-first search will find all nodes reachable in distance $w$, then $2w$, etc., where $w$ is the common weight. It will stop, therefore, at the lowest distance that reaches node $B$.

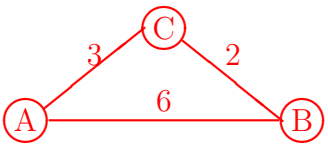
b. If all edges have distinct weights, the shortest path between any two vertices is unique.

False. Consider




c. Assume we have some graph, $G$, with a unique shortest path from vertex $A$ to $B$. We add a positive constant, $k$, to every edge to create a new graph $G'$. The shortest path in $G'$ from $A$ to $B$ is the same in $G'$ as in $G$.

False. Consider what happens for $k = 2$ where $G$ is as follows:

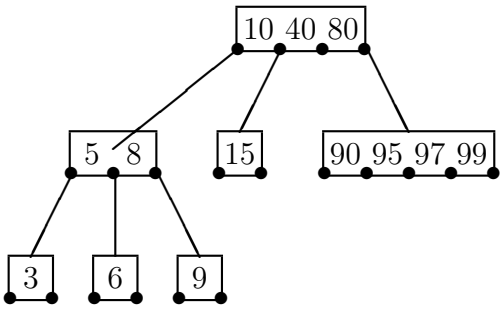



d. Topologically sorting a connected graph with $E$ edges requires $\Theta(E \lg E)$ time in the worst case.

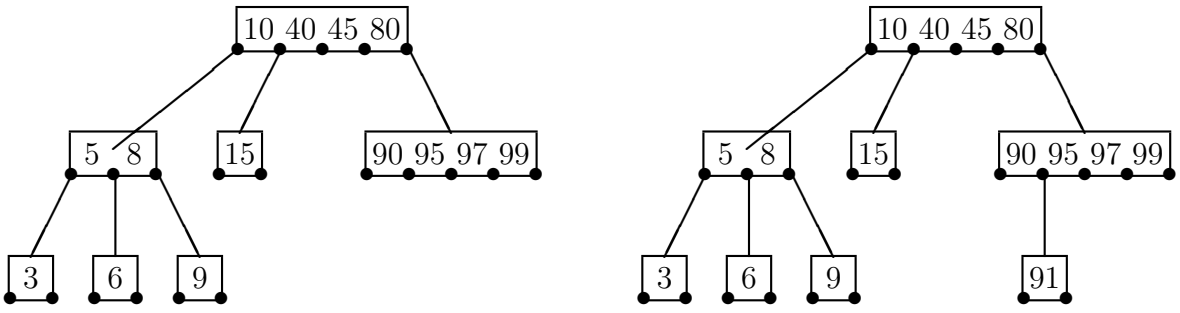
False. Topological sort can be effected by a depth-first search, which is linear in $E$ in the worst case.

**9.** [1 point] What do members of the U.S.I. at Berkeley have in common?
They have been incarcerated.

**10.** [6 points] An $M$-way search tree, of which a B-tree is a special case, has a sequence of $< M$ keys in sorted order in each node. A node with $0 < n < M$ keys (call them $K_0, \ldots K_{n-1}$) has $n + 1$ children (call them $C_0, \ldots, C_n$) with the property that for all $0 \leq i < n$, all keys in the subtree rooted at $C_i$ are $< K_i$, and all keys in the subtree rooted at $C_{i+1}$ are $> K_i$. Null values indicate empty trees. For example ($M = 5$):




We will use these trees to implement a kind of set (so adding a duplicate key should have no effect). The strategy for insertion of a new key, $A$, is to find a node, $P$, in which the appropriate child to contain $A$ is empty (null) and then either to insert $A$ in $P$ at the appropriate place if $P$ has fewer then $M$ children, and otherwise to create a new child of $P$ containing just $A$. For example, after inserting 45 into the example above, we would have the tree on the left below, and then after inserting 91, we'd have the tree on the right.




Fill in the class on the next two pages to implement this data structure (partially, that is; some operations are not considered.)

```
public class MTreeSet<Label extends Comparable<Label>> {
    private int _M;
    private Node _root;

    /** An M-way set.  Assumes M > 1. */
    public MTreeSet(int M) { _M = M; }

    private class Node {
        ArrayList<Label> keys = new ArrayList<>();
        ArrayList<Node> kids = new ArrayList<>();
        Node(Label K) {
            keys.add(K); kids.add(null); kids.add(null);
        }
    }

    /** Return true iff K is in this set. */
    public boolean contains(Label x) {
        return contains(_root, x);
    }

    private boolean contains(Node t, Label x) {
        if (t == null) {
            return false;
        }
        int nkeys = t.keys.size();
        for (int i = 0; i < nkeys; i += 1) {
            int c = x.compareTo(t.keys.get(i));
            if (c < 0) {
                return contains(t.kids.get(i), x);
            } else if (c == 0) {
                return true;
            }
        }
        return contains(t.kids.get(nkeys), x);
    }
```

*Continued on next page.*

*Continuation of* `MTreeSet`.

```
    /** Insert K into this set.  Has no effect if K is already in the
        set. */
    public void add(Label x) {
        _root = add(_root, x);
    }

    private Node add(Node t, Label x) {
        if (t == null) {
            return new Node(x);
        }
        int nkeys = t.keys.size();
        int i;
        for (i = 0; i < nkeys; i += 1) {
            int c = x.compareTo(t.keys.get(i));
            if (c < 0) {
                break;
            } else if (c == 0) {
                return t;
            }
        }
        if (t.kids.get(i) == null && nkeys + 1 < _M) {
            t.kids.add(i, null);
            t.keys.add(i, x);
        } else {
            t.kids.set(i, add(t.kids.get(i), x));
        }
        return t;
    }
}
```

**11.** [6 points] We wish to simulate an elevator in a building with floors numbered 1 to a parameter $N$, inclusive. If $N$ is 10, for example, we create our elevator simulator with a statement of the form

```
Elevator e = new Elevator(10);
```

At any given time, we can inquire as to the current floor with the expression

```
e.floor();
```

At any given time, the elevator has a current direction (up or down) and there may be outstanding requests to stop at various floors. Initially, `e.floor()` is 1, there are no requests, and the current direction is up. The elevator accepts floor requests when it is stopped. The call

```
e.request(F);
```

adds a request to stop at floor `F`, which must be between 1 and $N$, inclusive; it does nothing if there is already such a request or it is already stopped at floor `F`. In response to the method call

```
e.move()
```

the elevator moves to the nearest requested floor in its current direction, if any. If there are no more requested floors in that direction, but there are floors in the other direction, it first reverses the current direction and then goes to the nearest requested floor. If there are no outstanding requests, a call to `.move` has no effect.

Implement the `Elevator` class on the next page. It must implement an appropriate constructor and the public methods should be `floor`, `request`, and `move`. Somewhat unrealistically, assume the elevator can have a very large number of floors and receive very large numbers of requests. The reference material at the front of the exam contains a partial synopsis of the `PriorityQueue` and `Collections` classes.

```
import java.util.PriorityQueue;
import static java.util.Collections.reverseOrder;

public class Elevator {
    private int _N;
    private boolean _up;
    private int _floor;
    private PriorityQueue<Integer> _upQueue = new PriorityQueue<>();
    private PriorityQueue<Integer> _downQueue
            = new PriorityQueue<>(<Integer>reverseOrder());

    public Elevator(int N) {
        _N = N;
        _up = true;
        _floor = 1;
    }

    public int floor() {
        return _floor;
    }

    public void request(int floor) {
        if (_upQueue.contains(floor) || _downQueue.contains(floor)) return;
        if (floor > _floor) {
            _upQueue.add(floor);
        } else if (floor < _floor) {
            _downQueue.add(floor);
        }
    }

    public void move() {
        if (_upQueue.isEmpty() && _downQueue.isEmpty()) {
            return;
        } else if (_up && _upQueue.isEmpty() || !_up && _downQueue.isEmpty()) {
            _up = !_up;
        }
        if (_up) {
            _floor = _upQueue.poll();
        } else {
            _floor = _downQueue.poll();
        }
    }
```

```
}
```