HITOMISS from SPOJ

This problem is much more tricky than I initially thought. Not only I got (my first) “wrong answer” at SPOJ, but I got it 3 times. The last problem was discovered by Adrian Kuegel who took time to look at my code. Thanks!

The two difficulties raised by this problem are (1) finding a good criterion for deciding whether a case is *unwinnable* and (2) careful implementation of the simulation.

We can stop when the first player does not discard anything no matter how long it will continue. We can be sure this happens when we see the same card for the second time with the same count and in the meantime no card was discarded.

So we get to the second problem.

Implementing the above efficiently (and considering that the first player changes when it discards the last card) is hard. So I will cheat. I will just remember the last time step when something was discarded. If nothing is discarded for 663 steps then we are stuck. The magic number I just mentioned is the maximum LCM between 13 and a number from 1 to 52. This also gives a simple upper bound on the number of steps executed by the algorithm: $663 \cdot 52 \cdot 10$; at each step all 10 players are analyzed.

The overall structure.

```
⟨*⟩≡
  #include <stdlib.h>
  #include <stdio.h>
  #include <string.h>

  ⟨Defines⟩
  ⟨Constants⟩
  ⟨Global data⟩
  ⟨Global helper functions⟩

  int main() {
    int tests; scanf("%d ", &tests);
    for (int t = 0; t < tests; ++t) {
      ⟨Read input and construct data structures⟩
      ⟨Solve the problem⟩
      ⟨Write result⟩
      ⟨Cleanup⟩
      ⟨Report work⟩
    }
  }
```

```
⟨Constants⟩≡
  const int MAXLCM = 663;
```

As usual for SPOJ problems I shall evaluate efficiency using *mems*.

⟨*Defines*⟩≡

```
  #define o    (verbose ? ++mems : 0)
  #define oo   (verbose ? mems += 2 : 0)
  #define ooo  (verbose ? mems += 3 : 0)
  #define oooo (verbose ? mems += 4 : 0)
  #define MEMSET(c, v) \
    (verbose ? (mems += sizeof(c) / sizeof(int) + 1) : 0), \
    memset(c, v, sizeof(c))
```

⟨*Global data*⟩≡

```
  int mems;   // Memory references.
```

For representing the cards that a player holds the most natural data structure is a circular doubly-linked list. For each player we need to keep a pointer to the *top* card, the counter, and the last discarded card.

⟨*Constants*⟩+≡

```
  const int CARDS = 52;
  const int COUNT = 13;
  const int PLAYERS = 10;
```

⟨*Global data*⟩+≡

```
  struct Node {
    Node(int v) : v(v) {}
    int v;
    Node *p, *n; // Previous and next.
  };
  Node* top[PLAYERS];
  int last[PLAYERS];
  int counter[PLAYERS];
```

These data structures are built while reading the input.

⟨*Read input and construct data structures*⟩≡

```
  scanf("%d ", &players);
  for (c = 0; c < CARDS; ++c) {
    int v; scanf("%d ", &v);
    ooo, top[0] = insert_after(new Node(v-1), top[0]);
  }
  ooo,  top[0] = top[0]->n;
```

⟨*Global data*⟩+≡
```
  int players; // The total number of players.
  int p; // The current player of interest.
  int c; // The current card of interest.
```

I have already used an operation on the circular list: insertion of an element after a given one. I will also extract elements from lists.

⟨*Global helper functions*⟩≡
```
  Node* insert_after(Node* nn, Node* n) {
    if (!n) {
      oo, nn->p = nn->n = nn;
    } else {
      o,    nn->p = n;
      oo,   nn->n = n->n;
      oooo, nn->p->n = nn->n->p = nn;
    }
    return nn;
  }

  Node* insert_before(Node* nn, Node* n) {
    if (n) o, n = n->p;
    return insert_after(nn, n);
  }

  Node* extract(Node* n) {
    if (!n) return NULL;
    Node* np;
    o, np = n->p;
    if (np == n) return NULL;
    ooo, (np->n = n->n)->p = np;
    return np;
  }

  Node* delete_list(Node* n) {
    if (n) {
      delete_list(extract(n));
      delete n;
    }
    return NULL;
  }
```

We can now start the simulation. The pointer `handed_out[p]` keeps track of the card that was discarded by player `p`, if any.

⟨*Solve the problem*⟩≡
```
  MEMSET(counter, 0);
  int discard_time, time;
  discard_time = time = 0;
  int to_discard = players * CARDS;
  Node* handed_out[PLAYERS];
  while (to_discard && time <= discard_time + MAXLCM) {
    ++time;
    MEMSET(handed_out, 0);
    for (p = 0; p < players; ++p) if (o, top[p]) {
      if (ooo, top[p]->v == counter[p]) {
        if (verbose)
          fprintf(stderr, "Player %d discards %d.\n", p+1, counter[p]+1);
        oooo, last[p] = (handed_out[p] = top[p])->v;
        oo, top[p] = extract(top[p]);
        --to_discard;
        discard_time = time;
      }
      if (o, top[p]) ooo, top[p] = top[p]->n;
      o, ++counter[p] %= COUNT;
    }
    for (p = 1; p < players; ++p) if (o, handed_out[p-1]) {
      if (verbose)
        fprintf(stderr, "Take card handed by player %d.\n", p);
      ooo, top[p] = insert_before(handed_out[p-1], top[p]);
      ooo, top[p] = top[p]->n;
    }
    oo, delete handed_out[players-1];
  }
```

In case the game is “unwinnable” players still hold some cards and we need to delete them from memory and prepare for the next step.

⟨*Cleanup*⟩≡
```
  if (to_discard) {
    for (p = 0; p < players; ++p)
      oo, top[p] = delete_list(top[p]);
  }
```

The last part is writing the result.

⟨*Write result*⟩≡
```
if (to_discard)
  printf("Case %d: unwinnable\n", t+1);
else {
  printf("Case %d:", t+1);
  for (p = 0; p < players; ++p)
    o, printf(" %d", last[p] + 1);
  printf("\n");
}
```

For debuging purposes.

⟨*Constants*⟩+≡
```
const bool verbose = false;
```

⟨*Report work*⟩≡
```
if (verbose) {
  fprintf(stderr, "I used %d mems to solve case %d.\n", mems, t+1);
  mems = 0;
}
```