Fun with Linked Lists

Due October 16, 2023

The following is the user-defined type and function signatures that your programs must use:

```
template<class T>
struct Link {
    explicit Link(const T& info, Link *next = 0) : info(info), next(next) { }

    // This avoids stack overflow
    ~Link() {
        Link *p = next;
        while (p) {
            Link *q = p->next;
            p->next = nullptr;
            delete p;
            p = q;
        }
    }

    T info;
    Link *next;
};

vector<int> loopTail(Link* head);
vector<int> josephus(int n, int k);
```

This assignment has two parts, which are only related in that they both involve singly-linked lists in their implementation.

(a) Josephus

A story tells of a group of doomed soldiers, with the only honorable solution, suicide. In order to randomize the order of their deaths, they stand in a circle, count off, killing every third man, closing the circle as each death occurs. In your implementation, the number of soldiers, `n`, and the number counted off, `k`, are parameters, and the men are numbered from 1 to `n`. For example, if n is 10 and k is 3, the order of execution is

```
3 6 9 2 7 1 8 5 10 4
```

and the signature of the function you are to write is

```
vector<int> josephus(int n, int k);
```

Use a singly-linked list that is circular (the last link points to the first) to model the situation.

(b) Floyd

It sometimes happens that a program bug will cause a linked list that is intended to be linear (with a last element that points to null) to become looped, so that there is no last element. When this happens, in general the resulting list has an initial “tail” that is attached to a “loop”. Floyd’s algorithm is a way to tell whether this has occurred, and returns the number of elements in the loop (precisely, the number of pointers in the cyclical part of the structure) and the number of elements in the tail, in a two-element vector.

The algorithm works in three phases: the first determines if the list does, in fact, have a loop, the second determines the length of the loop, and the third determines the length of the tail.

The first phase is the cleverest, using a slow pointer and a fast pointer that traverses the list at twice the rate of the first. These two pointers meet if and only if the list has a loop.

The second phase, the most straightforward, commences once the two pointers have met (if they never meet, we are done). One pointer is held fixed, and the other counts around the loop until it reencounters the first.

The third phase is a bit trickier than the second. Move one pointer back to the beginning of the list, and place the second a distance (number of links) equal to the length of the loop found in the second phase. Then move both pointers one link at a time until they meet. The number of links that the two pointers move in parallel is the length of the tail.

The signature of the function you are to write is

```
vector<int> loopTail(Link* head);
```

The two-element vector returned contains the length of the loop followed by the length of the tail. Note that either element may be zero. (In fact, both may be zero, if the list is empty.)