Circular linked-list, A cycle detection problem

Larry LIU Xinyu

September 11, 2014

1 The problem

In imperative settings, a linked-list may be corrupted, that it is circular. In such a list, some node points back to previous one. Figure 1 shows such situation. The normal iteration ends up infinite looping.

1. Write a program to detect if a linked-list is circular;
2. Write a program to find the node where the loop starts (the node being pointed by two precedents).

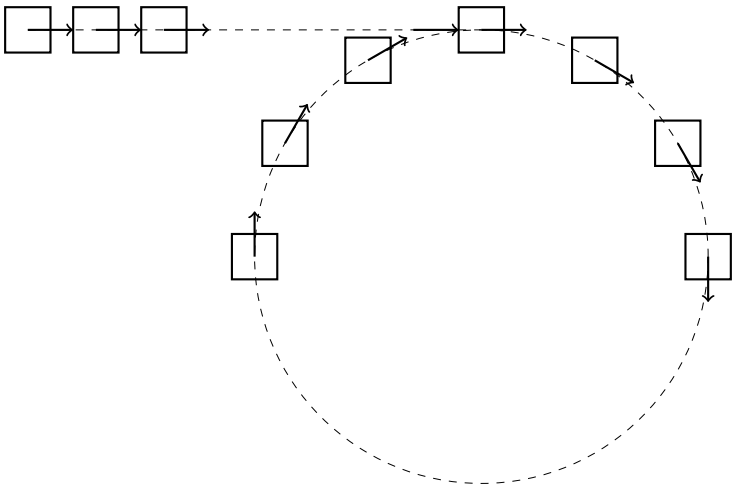


Figure 1: A circular linked-list

The brute-force solution utilizes extra spaces to record the nodes being visited so-far. When visits a node, if it has been visited, then the linked-list is circular, and this node is the staring point of the loop. When arrives at the tail, the linked-list isn't circular. This method isn't good enough. The space required is $O(n)$, where n is the number of nodes in the list. Depends on what data-structure is used to store the visited nodes, the performance varies from $O(n \lg n)$ (using tree-set) to quadratic $O(n^2)$ (using list for example).

2 Floyd's algorithm

Consider a clock. The hand of hours rotates slower than the hand of minutes. Starts from 12:00, they meets 12 times before the next 12:00. For this problem, we can mimic the clock hands by two pointers. They iterate in different speed. If there is a loop in a linked-list, the slower pointer must be caught by the faster one at some time.

But we must be careful when extend from the continuous case (the clock example) to the discrete case. Because the faster one may skip past the slower one [1], [2]. For some circle contains n nodes, the slower pointer iterates k nodes a step, the faster pointer iterates m nodes a step. Where n, m, k are all natural numbers. What if $am \not\equiv bk \pmod{n}$ for any integer a, b? Can you deduce the solution constraint for this linear congruence equation?

We can set the slower speed as $v_s = 1$ node per step, and the faster speed as $v_f = 2$ nodes per step. Starts from the head of the linked-list, if they meet at some time, the linked-list is circular. Otherwise, if the faster pointer arrives at or goes beyond the tail, the linked-list isn't circular.

```
1: function CIRCULAR?(L)
2:     p ← q ← L
3:     while p ≠ NIL ∧q ≠ NIL do
4:         p ← NEXT(p)
5:         q ← NEXT(q)
6:         if q = NIL then
7:             break
8:         q ← NEXT(q)
9:         if p = q then
10:            return True
11:    return False
```

The following ANSI C example program implements this method.

```
int is_circular(struct Node* h) {
    struct Node *a, *b;
    a = b = h;
    while (a && b) {
        a = a->next;
        b = b->next;
        if (!b)
            break;
        b = b->next;
        if (a == b) return 1;
    }
    return 0;
}
```

Suppose there are two runners. One runs two times faster than the other, If they start running at some point in a circular stadium, when they meet again?

The answer is the same starting point. The faster runner runs 2 circles and the the slower runner runs one circle when they meet again.

But for this problem, the situation is a bit different. The two 'runners' don't start racing both at point A as shown in figure 2. When the slower pointer arrives at A, the faster one is at point B. Because the speed $v_f = 2v_s$, we have $|OA| = |AB|$ if the circumference is longer than $|OA|$. Denote the node where the circle start as the k-th one, the faster pointer starts at the $2k$-th node when this 'racing' begins. Because it's a circle, the faster pointer runs after the slower one with distance of $n - k$ nodes.

Here we only consider the case that $k < n$, we'll explain the $k \geq n$ case next.

The time it takes when the two pointers meet again can be gotten by dividing the distance by the difference of their speed:

$$\begin{aligned} T &= \frac{n-k}{v_f - v_s} \\ &= \frac{n-k}{v_s} \qquad (\text{since } v_f = 2v_s) \end{aligned} \tag{1}$$

So when the two pointers meet, the distance that the slower pointer runs is $Tv_s = n - k$.

It means that the meet point is the k-th node before A around the circle. The only unknown value is k at this stage. Observe that point A is the k-th node from O as well. We can solve this problem with the following approach.

1. Use two pointers p, and q. Start from the head of the linked-list. p iterates a node per step, while q iterates two nodes per step;
2. If there is a loop, p and q will meet. Reset q to the head of the linked-list;
3. Advance p and q one node per step at the same time till they meet at A. Both pointer point to the node where the loop starts.

Let's consider the case that $k \geq n$. When the slower pointer arrives at A, The faster one points to B which is $k \bmod n$ nodes ahead. The faster one will catch the slower one by eliminating their distance of $n - (k \bmod n)$ nodes.

$$\begin{aligned} T &= \frac{n - (k \bmod n)}{v_f - v_s} \\ &= \frac{n - (k \bmod n)}{v_s} \end{aligned} \tag{2}$$

When they meet again, the slower pointer has passed $v_s T = n - (k \bmod n)$ nodes. The meet point is $k \bmod n$ nodes before A. We have the same result, that if we reset the faster pointer to the head of the linked-list, and make the two pointers advance one node per step, they will meet at A.

The following algorithm locates the node where the loop starts.

```
1: function FIND-LOOP(L)
2:     p ← q ← L
3:     while p ≠ NIL ∧ q ≠ NIL do
```

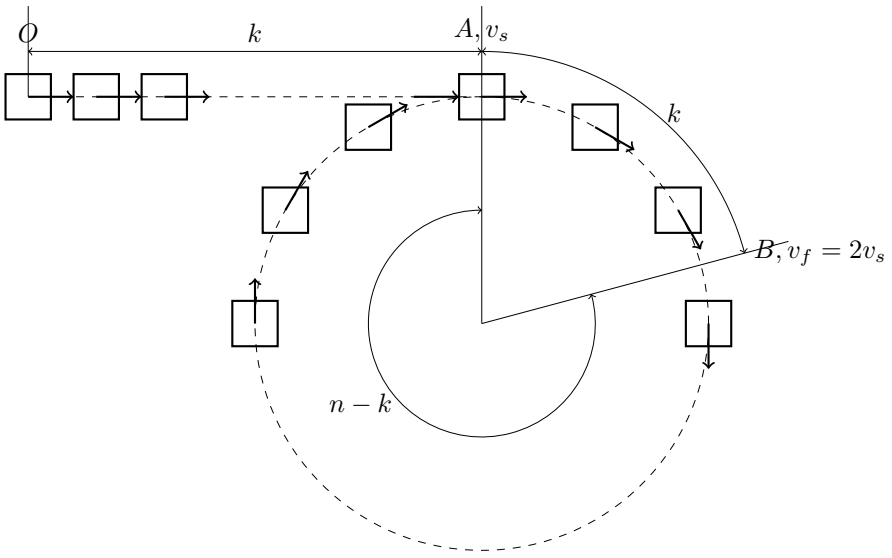


Figure 2: Two pointers solution.

```
4:        p ← NEXT(p)
5:        q ← NEXT(q)
6:        if q = NIL then
7:            break
8:        q ← NEXT(q)
9:        if p = q then
10:           q ← L
11:           while p ≠ q do
12:               p ← NEXT(p)
13:               q ← NEXT(q)
14:           return p                          ▷ Or return q
15:   return NIL                                ▷ No loop
```

The following ANSI C example program implements this solution.

```
struct Node* find_loop(struct Node* h) {
    struct Node *a, *b;
    a = b = h;
    while (a && b) {
        a = a->next;
        b = b->next;
        if (!b) break;
        b = b->next;
        if (a == b) {
            for (b = h; b != a; a = a->next, b = b->next);
            return a;
```

```
        }
    }
    return NULL; /*no loop*/
}
```

This solution is linear. The slower pointer exactly visits all nodes $(n + k)$ when there is a circle.

3 Brent's algorithm

Richard P. Brent described another linear time solution. If there is a circle with circumference of n, start from any point in the circle, a runner will eventually pass the same point after n steps.

As illustrated in figure 2, start from any node behind the k-th one should be OK. The problem is how to reach such a point. Brent's idea is to find the smallest number $m = 2^i$, that $m > k$ and $m > n$ for $i = 0, 1, 2, ...$ We expect that after advancing some nodes, if we enter the circle, we can calculate n out directly within m steps.

```
1:  function CYCLE-LENGTH(L)
2:      p ← q ← L
3:      n ← 0
4:      m ← 1
5:      repeat
6:          if n = m then
7:              p ← q
8:              m ← 2m
9:              n ← 0
10:         if q = NIL then
11:             break
12:         q ← NEXT(q)
13:         n ← n + 1
14:     until p = NIL ∨ q = NIL ∨ p = q
15:     if p = NIL ∨ q = NIL then
16:         n ← 0
17:     return n
```

After the circumference n is known, we can use two pointers, one points to the head of the linked-list. The other moves n nodes ahead. Then we advance these two pointers in parallel. When they meet, they both point to the node, where the loop starts.

```
1:  function FIND-CYCLE(L)
2:      n ← CYCLE-LENGTH(L)
3:      if n = 0 then
4:          return NIL
5:      p ← q ← L
6:      loop n times
```

```
 7:        q ← NEXT(q)
 8:    while p ≠ q do
 9:        p ← NEXT(p)
10:        q ← NEXT(q)
11:    return p                                          ▷ or q
```

The following ANSI C example program implements Brent's algorithm.

```
struct Node* find_cycle(struct Node* h) {
    struct Node *a, *b;
    int n = 0, power = 1;
    a = b = h;
    do {
        if (n == power) {
            a = b;
            power <<= 1;
            n = 0;
        }
        if (!b) break;
        b = b->next;
        ++n;
    } while (a && b && a != b);

    if (!a || !b) return NULL;

    for (b = h; n; --n, b = b->next);
    for (a = h; a != b; a = a->next, b = b->next);
    return a; /*or b*/
}
```

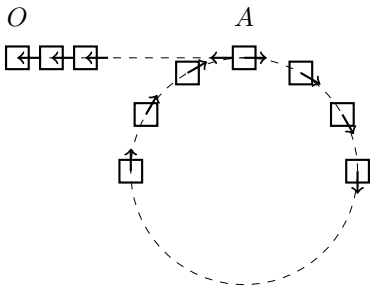
4 Other method

Another method to detect if there is a cycle without pointing the cycle starting point is to reverse the list. For a linked-list without circle, its tail becomes the new head after reversing. But for a linked-list with circle, the reversed head is same. Figure 3 shows the result.

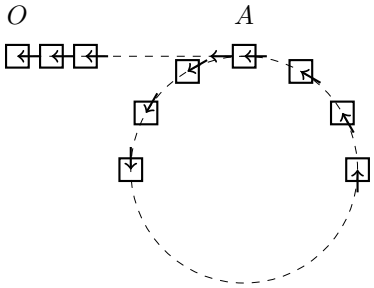
After all the pointers being reversed in stage 1 and 2, the reversing doesn't stop, but goes along with the line AO. Since all the nodes in section AO have already been reversed, these pointers are restored. We can compare if the new head is changed to tell if there is a cycle. After that, we need perform another reversing to restore the linked-list back.

The following ANSI C code implements this solution.

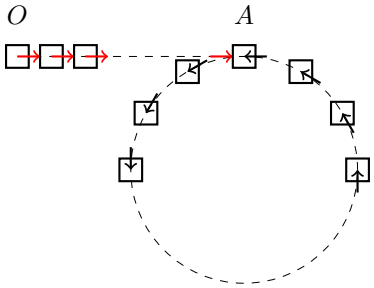
```
struct Node* reverse(struct Node* h) {
    struct Node *p = h, *h1 = NULL;
    while (h) {
```



(a) Stage 1, the part before the circle is reversed



(b) Stage 2, the circle part is reversed



(c) Stage 3, The nodes between the original head and the circle is reversed back

Figure 3: Reverse a circular list.

```
        h = p->next;
        p->next = h1;
        h1 = p;
        p = h;
    }
    return h1;
}

int detect(struct Node* h) {
    struct Node* h1 = reverse(h);
    reverse(h1); /*resume the original list*/
    return h == h1;
}
```

5 Cycle detection

Circular list is a special case of general Cycle detection problem [3]. For any function f maps finite set S to itself with any initial value x_0, the sequence of the iteration:

$$x_0, x_1 = f(x_0), x_2 = f(x_1), ...$$

must repeat. between some $i \neq j$, $x_i = x_j$. The cycle detection problem is to find i and j for given f and x_0. The first method described here is invented by Robert W. Floyd in later 1960s, as known as the 'tortoise and the hare' algorithm.

For the circular list problem, the f is defined as

$$f(x) = \begin{cases} next(x) & : & x \neq NIL \\ NIL & : & otherwise \end{cases} \tag{3}$$

The formalized Floyd's algorithm and Brent's algorithm are given as below. They accept the initial value x_0, and the function f, return k and n, where k is the value the cycle starts and n is the period of the cycle.

1: **function** FLOYD-CYCLE-DETECTION(x_0, f)
2: $p \leftarrow q \leftarrow x_0$ ▷ p: the slow (tortoise), q: the fast (hare)
3: **repeat**
4: $p \leftarrow f(p)$
5: $q \leftarrow f(f(q))$
6: **until** $p = q$ ▷ Loop until converge

7: $k \leftarrow 0$
8: $q \leftarrow x_0$ ▷ Reset the fast
9: **while** $p \neq q$ **do** ▷ Loop to the connection point
10: $p \leftarrow f(p)$
11: $q \leftarrow f(q)$

```
        k ← k + 1

    n ← 0
    repeat                                          ▷ Traverse the circle
        q ← f(q)
        n ← n + 1
    until q = p
    return (k, n)
```

```
function Brent-Cycle-Detection(x_0, f)
    n ← 0, m ← 1
    p ← q ← x_0
    repeat
        if m = n then
            p ← q                                   ▷ Reset the start point
            n ← 0
            m ← 2m
        q = f(q)
        n ← n + 1
    until p = q                                     ▷ Loop until converge

    p ← q ← x_0                                     ▷ Reset
    loop n times                                    ▷ Make distance |qp| = n
        q ← f(q)

    k ← 0
    while p ≠ q do                                  ▷ Loop to the connection point
        p ← f(p)
        q ← f(q)
        k ← k + 1
    return (k, n)
```

6 Functional approach

6.1 Floyd's algorithm

The cycle detection solution can be defined in purely functional form. For Floyd's method, we need define a function $converge(a, b)$. It applies function f to a, and applies f twice to b. This function recursively finds the converge point where $a = b$.

$$converge(a, b) = \begin{cases} a & : \quad a = b \\ converge(f(a), f(f(b))) & : \quad otherwise \end{cases} \tag{4}$$

The converge point is given as $x_i = converge(x_1, x_2)$, where $x_1 = f(x_0)$ and $x_2 = f(x_1)$. After that, we start iterating from x_0 and x_i in parallel to find the

connection point x_k.

$$connect(a, b, k) = \begin{cases} (k, a) & : \quad a = b \\ connect(f(a), f(b), k + 1) & : \quad otherwise \end{cases} \tag{5}$$

Because iterating f on x_0 of k times will arrive at the connection point A, and so as from the converge point x_i. This function returns a pair of values $(k, x_k) = connect(x_0, x_i, 0)$, where the third parameter zero is the initial value for accumulating k.

The connection point x_k is the place from where we can traverse the cycle to count for n.

$$traverse(a) = \begin{cases} 1 & : \quad a = x_k \\ 1 + traverse(f(a)) & : \quad otherwise \end{cases} \tag{6}$$

The final Floyd's algorithm can be realized as below.

$$floyd(x_0, f) = (k, n) \tag{7}$$

Where

$$\begin{cases} (k, x_k) = connect(x_0, converge(f(x_0), f(f(x_0))), 0) \\ n = traverse(x_k) \end{cases}$$

The following Haskell example program implements this algorithm.

```
findCycle x0 f = (k, n) where
  (k, p) = connect x0 (converge (f x0) (f $ f x0)) 0
  n = traverse (f p)
  converge a b | a == b = a
               | otherwise = converge (f a) (f $ f b)
  connect a b cnt | a == b = (cnt, a)
                  | otherwise = connect (f a) (f b) (cnt + 1)
  traverse a | a == p = 1
             | otherwise = 1 + traverse (f a)
```

In the environments that support lazy evaluation, the solution can also be realized using infinite series. Denote $X = \{x_0, x_1, x_2, ...\}$, and $X' = \{x_1, x_2, ...\}$. Taking the even number indexed elements yields $Y = \{x_0, x_2, x_4, ...\}$, and let $Y' = \{x_2, x_4, ...\}$ be the rest elements in Y. Y contains all the possible values that applies f twice from x_0. If we pairing X' and Y', as $zip(X', Y') = \{(x_1, x_2), (x_2, x_4), ...(x_j, x_{2j}), ...\}$, we can examine the pairs one by one till the converge point where the two values in the pair are equal. Dropping all the examined pairs, we will have an infinite series of pairs from the converge point

$$\begin{aligned} zip(C, D) &= \{(x_i, x_i), (x_{i+1}, x_{2(i+1)}), ...\} \\ &= dropWhile(\lambda_{(x_j, x_k)} \cdot x_j \neq x_k, zip(X', Y')) \end{aligned} \tag{8}$$

Thus $C = \{x_i, x_{i+1}, ...\}$. At this stage, we can pairing X and C, as $zip(X, C) = \{(x_0, x_i), (x_1, x_{i+1}), ...\}$, to search the connection point where the two values in the pair are equal. Then we break the infinite series into two parts Z_1, Z_2, the

first part contains elements before the connection point A. The length of it is $k = |Z_1|$. And we can count the cycle from the second part.

$$(Z_1, Z_2) = span(\lambda_{(x_i,x_j)} \cdot x_i \neq x_j, zip(X, C)) \tag{9}$$

In order to find the cycle in $Z_2 = \{(x_k, x_k), (x_{k+1}, x_{k+1}), ...\}$, we examine from the second pair one by one till meeting (x_k, x_k) again.

$$n = 1 + |takeWhile(\lambda_{(x_i,x_j)} \cdot x_i = x_j = x_k, Z_2')| \tag{10}$$

Where Z_2' contains all the pairs from the second one in Z_2. The following Haskell example program combines these ideas.

```
neq (x, y) = x /= y

findCycle' x0 f = (length sec, length cycle) where
  xs@(x:xs') = iterate f x0
  ys@(y:ys') = iterate (f.f) x0
  converge = fst $ unzip $ dropWhile neq (zip xs' ys')
  (sec, (z:zs)) = span neq (zip xs converge)
  cycle = z : takeWhile (z /=) zs
```

6.2 Brent's algorithm

In Brent's algorithm, the cycle period n is calculated first. We define a function $converge(a, b, n, m)$. Let m be the power of 2 series $1, 2, 4, 8, ...$, and check if it converges within m recursions.

$$converge(a, b, n, m) = \begin{cases} converge(a, f(a), 1, 2m) & : \quad n = m \\ n & : \quad a = b \\ converge(a, f(a), n + 1, m) & : \quad otherwise \end{cases} \tag{11}$$

If $n = m$, it can't converge in m steps. We reset the start point to a, the period counter n to 1, and try again in next $2m$ steps; When $a = b$, it converges. n records the circumference; Other wise, we step ahead. The cycle length n is given as $n = converge(x_0, f(x_0), 1, 1)$.

With n calculated, we can get k by finding the connection point $k = connect(x_0, f^n(x_0))$, where $f^n(x_0)$ means applying function f to x_0 with n times.

$$connect(a, b) = \begin{cases} 0 & : \quad a = b \\ 1 + connect(f(a), f(b)) & : \end{cases} \tag{12}$$

The following Haskell example program implements Brent's algorithm.

```
detectCycle x0 f = (k, n) where
  n = converge x0 (f x0) 1 1
  q = foldr ($) x0 (replicate n f)
  k = connect x0 q
  connect p q | p == q = 0
```

```
            | otherwise = 1 + connect (f p) (f q)
converge p q n m | n == m = converge q (f q) 1 (2*m)
                 | p == q = n
                 | otherwise = converge p (f q) (n + 1) m
```

Brent's algorithm can be realized with infinite series as well in the programming environment that supports lazy evaluation. We start m from 1, then 2, 4, ... For each value, we search the converge point in a section series of length m. For any given infinite values $Y = \{y_1, y_2, y_3, ...\}$, let $Y' = \{y_2, y_3, ...\}$. We split Y' at position m, $(Y_1, Y_2) = splitAt(m, Y')$. Then we search y_1 in Y_1. If we fail to find it, we need recursively search the converge point in a section of length $2m$ in Y_2. Otherwise the position of y_1 in Y_1 is the length of the cycle.

$$converge(Y, m) = \begin{cases} i & : \quad y_1 = y_i, y_i \in Y_1 \\ converge(Y_2, 2m) & : \quad y_1 \notin Y_1 \end{cases} \tag{13}$$

With n calculated, we can get k like below.

$$k = |takeWhile(\lambda_{(x_i, x_j)} \cdot x_i \neq x_j, zip(\{x_0, x_1, ...\}, \{x_n, x_{n+1}, ...\}))| \tag{14}$$

The following Haskell example program implements this method by using infinite series and lazy evaluation.

```
detectCycle' x0 f = (k, n) where
  xs = iterate f x0
  n = converge xs 1
  k = length $ takeWhile neq (zip xs (drop n xs))
  converge (x:xs) m = let
      (ys, zs) = splitAt m xs in
      case elemIndex x ys of
        Nothing → converge zs (2*m)
        (Just idx) → idx + 1
```

The complete ANSI C and Haskell example programs with test are distributed with this article.

References

[1] Alexander Stepanov, Paul McJones. "Elements of Programming". Section 2.3, Chapter 2. Addison-Wesley Professional; (2009)

[2] Donald E. Knuth. "The Art of Computer Programming, Volume 2: Seminumerical Algorithms (3rd Edition)". Addison-Wesley Professional; (1997)

[3] Cycle detection. Wikipedia. http://en.wikipedia.org/wiki/Cycle_detection