Check if two linked lists merge. If so, where?

Asked 15 years, 2 months ago Modified 3 years, 11 months ago

Viewed 51k times

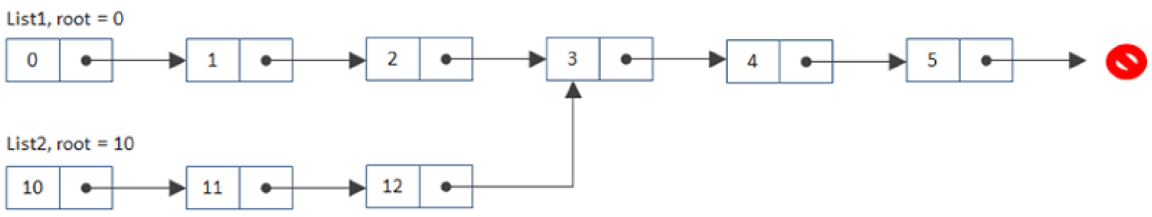
115

This question may be old, but I couldn't think of an answer.

Say, there are two lists of different lengths, **merging at a point**; how do we know where the merging point is?

Conditions:

1. We don't know the length
2. We should parse each list only once.



algorithm linked-list data-structures

Share

Improve this question

Follow

edited May 29, 2014 at 3:10

Richard

60.8k ● 39 ● 194 ● 273

asked Oct 20, 2009 at 11:51

rplusg
3,446 ● 6 ● 37 ● 51

merge means from that point there will be only one list.
– rplusg Oct 20, 2009 at 11:57

is modification of the list allowed? – Artelius Oct 20, 2009 at 11:59

1 I'm pretty sure it doesn't work without modification of the list. (Or just copying it somewhere else to avoid the restriction to parse it only once.) – Georg Schölly Oct 20, 2009 at 13:40

2 Might have been the point. Damn interviewers! Hehe
– Kyle Rosendo Oct 20, 2009 at 16:38

1 I have an interesting proposal... assuming the common tail of the list is infinitely long. How can you find the node intersection using constant memory? – Akusete Aug 18, 2010 at 6:24

27 Answers

Sorted by: Highest score (default)

189

The following is by far the greatest of all I have seen - O(N), no counters. I got it during an interview to a candidate S.N. at VisionMap.

Make an interating pointer like this: it goes forward every time till the end, and then jumps to the beginning of the opposite list, and so on. Create two of these, pointing to two heads. Advance each of the pointers by 1 every time, until they meet. This will happen after either one or two passes.

I still use this question in the interviews - but to see how long it takes someone to understand why this solution works.

Share Improve this answer Follow

edited Oct 21, 2013 at 7:53

answered Feb 19, 2013 at 11:16

Pavel Radzivilovsky
19.1k ● 5 ● 59 ● 69

---

2 This is a good answer, but you have to go through the lists twice which violates condition #2. – tster May 29, 2014 at 18:05

---

1 By opposite list, do you mean traversing the pointers of the same list in the opposite direction, or in other words the `prev` pointer of a `doubly linked list` ? – Tuxdude Feb 24, 2015 at 22:02

---

2 I find this solution quite elegant, if a merge point is guaranteed to be present. It will not work to detect merge points, as if one is not present it will loop infinitely. – alternate direction Mar 13, 2016 at 4:16

---

4 That's super brilliant! Explanation: we have 2 lists: `a-b-c-x-y-z` and `p-q-x-y-z`. path of first pointer `a,b,c,x,y,z,p,q,x`, path of second pointer `p,q,x,y,z,a,b,c,x` – Nikolai Golub Dec 11, 2016 at 19:43

---

23 Brilliant. For those who didn't understand, count the number of nodes traveled from head1-> tail1 -> head2 -> intersection point and head2 -> tail2-> head1 -> intersection point. Both will be equal(Draw diff types of linked lists to verify this). Reason is both pointers have to travel same distances

head1-> IP + head2->IP before reaching IP again. So by the time it reaches IP, both pointers will be equal and we have the merging point. – adev Jul 28, 2017 at 5:37

98

Pavel's answer requires modification of the lists *as well as* iterating each list twice.

Here's a solution that *only* requires iterating each list twice (the first time to calculate their length; if the length is given you only need to iterate once).

The idea is to ignore the starting entries of the longer list (merge point can't be there), so that the two pointers are an equal distance from the end of the list. Then move them forwards until they merge.

```
lenA = count(listA) //iterates list A
lenB = count(listB) //iterates list B

ptrA = listA
ptrB = listB

//now we adjust either ptrA or ptrB so that they
are equally far from the end
while(lenA > lenB):
    ptrA = ptrA->next
    lenA--
while(lenB > lenA):
    prtB = ptrB->next
    lenB--

while(ptrA != NULL):
    if (ptrA == ptrB):
        return ptrA //found merge point
    ptrA = ptrA->next
    ptrB = ptrB->next
```

This is asymptotically the same (linear time) as my other answer but probably has smaller constants, so is probably faster. But I think my other answer is cooler.

Share Improve this answer Follow

answered Oct 20, 2009 at 22:16

Artelius
49k ● 13 ● 92 ● 106

2 +1 like this and also doesnt need any modification to the list, also most of the linked-list implementations usually provide for length – keshav84 Sep 4, 2010 at 8:28

5 We have too many Pavels. My solution does not require modifying the list. – Pavel Radzivilovsky Mar 22, 2015 at 13:15

Good answer. What will the time complexity for this though. 0(n + m) ? where n = nodes in list 1 , m = nodes in list 2 ? – Vihaan Verma Jan 16, 2017 at 13:55

instead of moving both of the pointers in both list: we can just see if the diff >= small of two path, if yes, then move in small list by small value else move in small list by diff + 1 value; if diff is 0 then last node is the answer. – Vishal Anand Jun 29, 2017 at 1:20

In fact this can even be used to calculate if there is a merge point as once the end of one list is reached we could just store the end node and compare once the other list reaches it end. As we only create a virtual cycle and not a real one, this works well. – LionsAd Feb 26, 2018 at 23:45

If

39

- by "modification is not allowed" it was meant "you may change but in the end they should be restored", and
- we could iterate the lists exactly *twice*

the following algorithm would be the solution.

First, the numbers. Assume the first list is of length `a+c` and the second one is of length `b+c`, where `c` is the length of their common "tail" (after the mergepoint). Let's denote them as follows:

```
x = a+c
y = b+c
```

Since we don't know the length, we will calculate `x` and `y` without additional iterations; you'll see how.

Then, we iterate each list and reverse them while iterating! If both iterators reach the merge point at the same time, then we find it out by mere comparing. Otherwise, one pointer will reach the merge point before the other one.

After that, when the other iterator reaches the merge point, it won't proceed to the common tail. Instead will go back to the former beginning of the list that had reached merge-point before! So, before it reaches the end of the changed list (i.e. the former beginning of the other list), he will make `a+b+1` iterations total. Let's call it `z+1`.

The pointer that reached the merge-point first, will keep iterating, until reaches the end of the list. The number of iterations it made should be calculated and is equal to `x`.

Then, this pointer iterates back and reverses the lists again. But now it won't go back to the beginning of the list it originally started from! Instead, it will go to the beginning of the other list! The number of iterations it made should be calculated and equal to `y`.

So we know the following numbers:

```
x = a+c
y = b+c
z = a+b
```

From which we determine that

```
a = (+x-y+z)/2
b = (-x+y+z)/2
c = (+x+y-z)/2
```

Which solves the problem.

Share Improve this answer Follow

edited Oct 20, 2009 at 14:41

answered Oct 20, 2009 at 12:47

P Shved

99.1k ● 19 ● 127 ● 167

2 Comment to question states modification of list not allowed! – Skizz Oct 20, 2009 at 13:21

1 I like this answer (very creative). The only problem I have with it is that it assumes you know the length of both lists. – tster Oct 20, 2009 at 13:49

you cannot modify list, and we don't know the length--these are the constraints...any how, thanks for a creative answer. – rplusg Oct 20, 2009 at 14:15

2 @tster , @calvin , the answer doesn't assume, we need the length. It can be calculated inline. Adding explanations to my answers. – P Shved Oct 20, 2009 at 14:28

2 @Forethinker hashing visited nodes and/or marking them as seen requires O(list length) memory, while many solutions (including mine, however imperfect and complicated it is) require O(1) memory. – P Shved Jul 23, 2013 at 18:02

37

Well, if you know that they will merge:

Say you start with:

```
A-->B-->C
        |
        V
1-->2-->3-->4-->5
```

1) Go through the first list setting each next pointer to NULL.

Now you have:

```
A   B   C

1-->2-->3   4   5
```

2) Now go through the second list and wait until you see a NULL, that is your merge point.

If you can't be sure that they merge you can use a sentinel value for the pointer value, but that isn't as elegant.

Share Improve this answer Follow

answered Oct 20, 2009 at 12:43

tster
18.2k ● 6 ● 56 ● 72

4 However, you destroy the list in the process, never to be used again :P – Kyle Rosendo Oct 20, 2009 at 12:46

@Kyle Rozendo , well, my solution changes lists in the way they can be restored after processing. But this is more clear demonstration of the concept – P Shved Oct 20, 2009 at 12:55

I didn't see that modification of the list was not allowed. I'll give it a think, but nothing is coming to mind without storing every node seen. – tster Oct 20, 2009 at 13:20

11 C'mon, that's the correct answer! We just need to adjust the question :) – P Shved Oct 20, 2009 at 13:28

28 Excellent algorithm to create memory leaks. – Karoly Horvath Apr 28, 2013 at 7:42

14

If we could iterate lists exactly twice, than I can provide method for determining merge point:

- iterate both lists and calculate lengths A and B
- calculate difference of lengths C = |A-B|;
- start iterating both list simultaneously, but make additional C steps on list which was greater
- this two pointers will meet each other in the merging point

Share Improve this answer Follow

answered Oct 26, 2009 at 17:09

rachvela
730 ● 4 ● 11

8

Here's a solution, computationally quick (iterates each list once) but uses a lot of memory:

```
for each item in list a
  push pointer to item onto stack_a

for each item in list b
  push pointer to item onto stack_b

while (stack_a top == stack_b top) // where top is the item to be popped next
  pop stack_a
  pop stack_b

// values at the top of each stack are the items prior to the merged item
```

Share Improve this answer

Follow

edited Jul 3, 2012 at 14:13

user142162

answered Oct 20, 2009 at 13:12

Skizz

71k ● 10 ● 74 ● 109

2 That's the equivalent of processing a list twice. – Georg Schölly Oct 20, 2009 at 13:40

I suppose that, technically, you're doing stuff with the lists twice, but it's a significant improvement on Kyle Rozendo's solution. Now, if 'processing the list' is defined as 'reading the link value and following the pointer' it could be argued that it does process the list once - it reads each link value once, stores it and then compares them. – Skizz Oct 20, 2009 at 13:56

Is definitely going to be faster than mine, no doubt. – Kyle Rosendo Oct 20, 2009 at 16:43

8

You can use a set of Nodes. Iterate through one list and add each Node to the set. Then iterate through the second list and for every iteration, check if the Node exists in the set. If it does, you've found your merge point :)

Share Improve this answer

Follow

answered Feb 8, 2013 at 7:14

isyi

1,302 ● 1 ● 11 ● 11

I'm afraid (because of Ω(n) additional space) this is the only approach (not sort of rebuilding the list(s) and) not parsing a

list more than once. Detecting a loop in the list is trivial for the first list (check if node in set) - use any loop detection method on the second list to ensure termination. (The interview question *may* have been about listening *carefully* to a problem statement, and *not* jumping in to use a hammer you happen to know to hit something not quite a nail.)
– greybeard Oct 18, 2016 at 5:00

6

This arguably violates the "parse each list only once" condition, but implement the tortoise and hare algorithm (used to find the merge point and cycle length of a cyclic list) so you start at List A, and when you reach the NULL at the end you pretend it's a pointer to the beginning of list B, thus creating the appearance of a cyclic list. The algorithm will then tell you exactly how far down List A the merge is (the variable 'mu' according to the Wikipedia description).

Also, the "lambda" value tells you the length of list B, and if you want, you can work out the length of list A during the algorithm (when you redirect the NULL link).

Share Improve this answer Follow

edited Oct 20, 2009 at 13:02

answered Oct 20, 2009 at 12:11

Artelius
49k ● 13 ● 92 ● 106

Pretty much what I said, just with fancier names. :P
– Kyle Rosendo Oct 20, 2009 at 12:13

Not at all. This solution is O(n) in operations and O(1) in memory usage (in fact only requires two pointer variables). – Artelius Oct 20, 2009 at 12:46

Yea, should have deleted my prior comment as my solution changed a bit. Hehe. – Kyle Rosendo Oct 20, 2009 at 12:48

But I don't see how that was applicable in the first place? – Artelius Oct 20, 2009 at 12:55

Your explanation did, not the algorithm itself. Perhaps I view it differently, but hey. – Kyle Rosendo Oct 20, 2009 at 13:04

3

Maybe I am over simplifying this, but simply iterate the smallest list and use the last nodes `Link` as the merging point?

So, where `Data->Link->Link == NULL` is the end point, giving `Data->Link` as the merging point (at the end of the list).

EDIT:

Okay, from the picture you posted, you parse the two lists, the smallest first. With the smallest list you can maintain the references to the following node. Now, when you parse the second list you do a comparison on the reference to find where Reference [i] is the reference at LinkedList[i]->Link. This will give the merge point. Time to explain with pictures (superimpose the values on the picture the OP).

You have a linked list (references shown below):

`A->B->C->D->E`

You have a second linked list:

`1->2->`

With the merged list, the references would then go as follows:

`1->2->D->E->`

Therefore, you map the first "smaller" list (as the merged list, which is what we are counting has a length of 4 and the main list 5)

Loop through the first list, maintain a reference of references.

The list will contain the following references `Pointers { 1, 2, D, E }`.

We now go through the second list:

```
-> A - Contains reference in Pointers? No, move on
-> B - Contains reference in Pointers? No, move on
-> C - Contains reference in Pointers? No, move on
-> D - Contains reference in Pointers? Yes, merge
point found, break.
```

Sure, you maintain a new list of pointers, but thats not outside the specification. However the first list is parsed exactly once, and the second list will only be fully parsed if there is no merge point. Otherwise, it will end sooner (at the merge point).

Share Improve this answer Follow

edited Oct 20, 2009 at 12:28

answered Oct 20, 2009 at 11:53

Kyle Rosendo
25.3k ● 8 ● 82 ● 118

Well changes slightly from what I wanted to say at first, but from what the OP seems to want, this will do the trick.
– Kyle Rosendo Oct 20, 2009 at 12:32

It's clearer now. But linear in memory use. I don't like that.
– Artelius Oct 20, 2009 at 12:32

The question didn't ask for more, otherwise the entire process can be multithreaded. This is still a simplistic "top level" view of the solution, the code can be implemented any number of ways. :) – Kyle Rosendo Oct 20, 2009 at 12:37

1 Uh, what? Multithreading is a way of better utilising processing power, not reducing the total processing power an algorithm requires. And saying the code can be implemented in any number of ways is just an excuse. – Artelius Oct 20, 2009 at 12:51

1 This really bends the 'parse each list only once' to near breaking point. All you're doing is copying one list and then checking the other list against the copy. – Skizz Oct 20, 2009 at 13:17

3

I have tested a merge case on my FC9 x86_64, and print every node address as shown below:

```
Head A 0x7fffb2f3c4b0
0x214f010
0x214f030
0x214f050
0x214f070
0x214f090
0x214f0f0
0x214f110
0x214f130
0x214f150
0x214f170


Head B 0x7fffb2f3c4a0
0x214f0b0
0x214f0d0
0x214f0f0
0x214f110
0x214f130
0x214f150
0x214f170
```

Note becase I had aligned the node structure, so when malloc() a node, the address is aligned w/ 16 bytes, see the least 4 bits. The least bits are 0s, i.e., 0x0 or 000b. So if your are in the same special case (aligned node address) too, you can use these least 4 bits. For example when travel both lists from head to tail, set 1 or 2 of the 4 bits of the visiting node address, that is, set a flag;

```
next_node = node->next;
node = (struct node*)((unsigned long)node |
0x1UL);
```

Note above flags won't affect the real node address but only your SAVED node pointer value.

Once found somebody had set the flag bit(s), then the first found node should be the merge point. after done, you'd restore the node address by clear the flag bits you had set. while an important thing is that you should be careful when iterate (e.g. node = node->next) to do clean. remember you had set flag bits, so do this way

```
real_node = (struct node*)((unsigned long)node) &
~0x1UL);
real_node = real_node->next;
node = real_node;
```

Because this proposal will restore the modified node addresses, it could be considered as "no modification".

Share Improve this answer Follow

edited Oct 21, 2009 at 2:02

answered Oct 20, 2009 at 15:54

Test
1,727 ● 1 ● 11 ● 10

+1, this is what naturally comes to mind with "iterate only once" dunno why this never got up voted! Beautiful solution.
– jman Jul 8, 2013 at 2:50

3

There can be a simple solution but will require an auxilary space. The idea is to traverse a list and store each address in a hash map, now traverse the other list and match if the address lies in the hash map or not. Each list is traversed only once. There's no modification to any list. Length is still unknown. Auxiliary space used: O(n) where 'n' is the length of first list traversed.

Share Improve this answer Follow

answered Aug 5, 2018 at 5:31

Vikas Agarwal
82 ● 5

2

this solution iterates each list only once...no modification of list required too..though you may complain about space..

1) Basically you iterate in list1 and store the address of each node in an array(which stores unsigned int value)

2) Then you iterate list2, and for each node's address ---> you search through the array that you find a match or not...if you do then this is the merging node

```
//pseudocode
//for the first list
p1=list1;
unsigned int addr[];//to store addresses
i=0;
while(p1!=null){
  addr[i]=&p1;
  p1=p1->next;
}
int len=sizeof(addr)/sizeof(int);//calculates
length of array addr
//for the second list
p2=list2;
while(p2!=null){
  if(search(addr[],len,&p2)==1)//match found
  {
    //this is the merging node
    return (p2);
  }
  p2=p2->next;
}

int search(addr,len,p2){
  i=0;
  while(i<len){
    if(addr[i]==p2)
      return 1;
    i++;
  }
 return 0;
}
```

Hope it is a valid solution...

Share Improve this answer Follow

edited Mar 7, 2011 at 18:58

answered Mar 7, 2011 at 18:53

rajya vardhan
1,131 ● 4 ● 17 ● 29

This pretty much iterates one of the lists more than once, though in form of an array instead of the list itself. – syockit Jun 12, 2016 at 13:30

1

There is no need to modify any list. There is a solution in which we only have to traverse each list once.

1. Create two stacks, lets say stck1 and stck2.
2. Traverse 1st list and push a copy of each node you traverse in stck1.

3. Same as step two but this time traverse 2nd list and push the copy of nodes in stck2.
4. Now, pop from both stacks and check whether the two nodes are equal, if yes then keep a reference to them. If no, then previous nodes which were equal are actually the merge point we were looking for.

Share Improve this answer Follow

answered Jun 9, 2018 at 13:49

ABVash

26 ● 3

1

```
int FindMergeNode(Node headA, Node headB) {
  Node currentA = headA;
  Node currentB = headB;

  // Do till the two nodes are the same
  while (currentA != currentB) {
    // If you reached the end of one list start at
the beginning of the other
    // one currentA
    if (currentA.next == null) {
      currentA = headA;
    } else {
      currentA = currentA.next;
    }
    // currentB
    if (currentB.next == null) {
      currentB = headB;
    } else {
      currentB = currentB.next;
    }
  }
  return currentB.data;
}
```

Share Improve this answer

Follow

edited Apr 20, 2020 at 12:49

mr NAE

3,372 ● 1 ● 19 ● 38

answered Nov 3, 2019 at 7:37

Fahad Israr

1,229 ● 11 ● 10

In its original revision, this just spelled out the highest voted answer (Pavel Radzivilovsky, 2013). – greybeard Apr 20, 2020 at 15:40

1

We can use two pointers and move in a fashion such that if one of the pointers is null we point it to the head of the other list and same for the other, this way if the list lengths are different they will meet in the second pass. If length of list1 is n and list2 is m, their difference is d=abs(n-m). They will cover this distance and meet at the merge point.
Code:

```
int findMergeNode(SinglyLinkedListNode* head1,
SinglyLinkedListNode* head2) {
    SinglyLinkedListNode* start1=head1;
    SinglyLinkedListNode* start2=head2;
    while (start1!=start2){
        start1=start1->next;
        start2=start2->next;
        if (!start1)
        start1=head2;
        if (!start2)
        start2=head1;
    }
    return start1->data;
}
```

Share Improve this answer

Follow

answered Jul 4, 2020 at 14:45

aditya singh

11 ● 3

▲

0

▼

Here is naive solution , No neeed to traverse whole lists.

if your structured node has three fields like

```
struct node {
    int data;
    int flag;  //initially set the flag to zero
for all nodes
    struct node *next;
};
```

say you have two heads (head1 and head2) pointing to head of two lists.

Traverse both the list at same pace and put the flag =1(visited flag) for that node ,

```
  if (node->next->field==1)//possibly longer list
will have this opportunity
      //this will be your required node.
```

Share Improve this answer

Follow

answered Feb 27, 2012 at 8:37

Anil Arya

3,110 ● 7 ● 45 ● 69

0

How about this:

1. If you are only allowed to traverse each list only once, you can create a new node, traverse the first list to have every node point to this new node, and traverse the second list to see if any node is pointing to your new node (that's your merge point). If the second traversal doesn't lead to your new node then the original lists don't have a merge point.
2. If you are allowed to traverse the lists more than once, then you can traverse each list to find our their lengths and if they are different, omit the "extra" nodes at the beginning of the longer list. Then just traverse both lists one step at a time and find the first merging node.

Share Improve this answer Follow

answered Jan 30, 2013 at 5:19

user2024069
1

1. not only modifies but destructs the first list. 2. is suggested time and again. – greybeard Aug 23, 2016 at 6:49

0

Steps in Java:

1. Create a map.
2. Start traversing in the both branches of list and Put all traversed nodes of list into the Map using some

unique thing related to Nodes(say node Id) as Key and put Values as 1 in the starting for all.

3. When ever first duplicate key comes, increment the value for that Key (let say now its value became 2 which is > 1.

4. Get the Key where the value is greater than 1 and that should be the node where two lists are merging.

Share Improve this answer Follow

edited Apr 28, 2013 at 7:55

Jens Erat
38.6k ● 16 ● 84 ● 98

answered Apr 28, 2013 at 7:36

King KB
1

1 What if we have cycle in the merged part? – Rohit Mar 2, 2014 at 18:29

But for the error handling cycles, this looks very much like isyi's answer. – greybeard Oct 18, 2016 at 5:09

0

We can efficiently solve it by introducing "isVisited" field. Traverse first list and set "isVisited" value to "true" for all nodes till end. Now start from second and find first node where flag is true and Boom ,its your merging point.

Share Improve this answer Follow

answered Oct 17, 2016 at 18:23

Riya kathil
19 ● 5

0

Step 1: find lenght of both the list Step 2 : Find the diff and move the biggest list with the difference Step 3 : Now both list will be in similar position. Step 4 : Iterate through list to find the merge point

```
//Psuedocode
def findmergepoint(list1, list2):
lendiff = list1.length() > list2.length() :
list1.length() - list2.length() ? list2.lenght()-
list1.lenght()
biggerlist = list1.length() > list2.length() :
list1 ? list2  # list with biggest length
smallerlist = list1.length() < list2.length() :
list2 ? list1 # list with smallest length


# move the biggest length to the diff position to
level both the list at the same position
for i in range(0,lendiff-1):
    biggerlist = biggerlist.next
#Looped only once.
while ( biggerlist is not None and smallerlist is
not None ):
    if biggerlist == smallerlist :
        return biggerlist #point of intersection
return None // No intersection found
```

Share Improve this answer Follow

edited Nov 14, 2016 at 12:07

answered Nov 14, 2016 at 11:52

svaithin

75 ● 1 ● 7

(I liked the list with each item starting a line better. Consider using a spelling checker.) – greybeard Nov 14, 2016 at 12:08

0

```
int FindMergeNode(Node *headA, Node *headB)
{
    Node *tempB=new Node;
    tempB=headB;
   while(headA->next!=NULL)
       {
       while(tempB->next!=NULL)
           {
           if(tempB==headA)
               return tempB->data;
           tempB=tempB->next;
       }
       headA=headA->next;
       tempB=headB;
   }
    return headA->data;
}
```

Share Improve this answer Follow

answered May 16, 2017 at 7:05

Yachana Yachana

1

You need to add some explanation to your answer. Code only answers may get deleted. – rghome May 16, 2017 at 7:14

This is just a brute force approach comparing every single element of first list to every element of first list. that gives a time complexity of O(n*m). – Amit Dube Jun 13, 2021 at 8:45

0

Use Map or Dictionary to store the addressess vs value of node. if the address alread exists in the Map/Dictionary then the value of the key is the answer. I did this:

```
int FindMergeNode(Node headA, Node headB) {

Map<Object, Integer> map = new HashMap<Object,
Integer>();

while(headA != null || headB != null)
{
    if(headA != null &&
map.containsKey(headA.next))
    {
        return map.get(headA.next);
    }

    if(headA != null && headA.next != null)
    {
         map.put(headA.next, headA.next.data);
         headA = headA.next;
    }

    if(headB != null &&
map.containsKey(headB.next))
    {
        return map.get(headB.next);
    }

    if(headB != null && headB.next != null)
    {
        map.put(headB.next, headB.next.data);
        headB = headB.next;
    }
}

return 0;
}
```

Share Improve this answer

Follow

answered Jun 29, 2017 at 1:24

Vishal Anand

1,461 ● 1 ● 16 ● 37

0

A O(n) complexity solution. But based on an assumption.

assumption is: both nodes are having only positive integers.

logic : make all the integer of list1 to negative. Then walk through the list2, till you get a negative integer. Once found => take it, change the sign back to positive and return.

```
static int findMergeNode(SinglyLinkedListNode
head1, SinglyLinkedListNode head2) {

    SinglyLinkedListNode current = head1; //head1
is give to be not null.

    //mark all head1 nodes as negative
    while(true){
        current.data = -current.data;
        current = current.next;
        if(current==null) break;
    }

    current=head2; //given as not null
    while(true){
        if(current.data<0) return -current.data;
        current = current.next;
    }

}
```

Share Improve this answer

Follow

answered Sep 28, 2018 at 19:08

user3828943

11 ● 1

0

You can add the nodes of `list1` to a hashset and the loop through the second and if any node of `list2` is already present in the set .If yes, then thats the merge node

```
static int findMergeNode(SinglyLinkedListNode
head1, SinglyLinkedListNode head2) {
    HashSet<SinglyLinkedListNode> set=new
HashSet<SinglyLinkedListNode>();
    while(head1!=null)
    {
        set.add(head1);
        head1=head1.next;
    }
    while(head2!=null){
        if(set.contains(head2){
            return head2.data;
        }
    }
    return -1;
}
```

Share Improve this answer

Follow

edited Jul 12, 2020 at 12:52

Boken

5,390 ● 10 ● 38 ● 45

answered Jul 12, 2020 at 12:32

C.B. Shivananda
1

0

Solution using javascript

```
var getIntersectionNode = function(headA, headB) {

    if(headA == null || headB == null) return null;

    let countA = listCount(headA);
    let countB = listCount(headB);

    let diff = 0;
    if(countA > countB) {

        diff = countA - countB;
        for(let i = 0; i < diff; i++) {
            headA = headA.next;
        }
    } else if(countA < countB) {
        diff = countB - countA;
        for(let i = 0; i < diff; i++) {
            headB = headB.next;
        }
    }

    return getIntersectValue(headA, headB);
};

function listCount(head) {
    let count = 0;
    while(head) {
        count++;
        head = head.next;
    }
    return count;
}
```

```
function getIntersectValue(headA, headB) {
    while(headA && headB) {
        if(headA === headB) {
            return headA;
        }
        headA = headA.next;
        headB = headB.next;
    }
    return null;
}
```

Share Improve this answer

Follow

answered Aug 9, 2020 at 7:45

Rama Adaikkalam
713 ● 1 ● 7 ● 15

-1

If editing the linked list is allowed,

1. Then just make the next node pointers of all the nodes of list 2 as null.
2. Find the data value of the last node of the list 1. This will give you the intersecting node in single traversal of both the lists, with "no hi fi logic".

Share Improve this answer

Follow

answered Aug 10, 2020 at 10:15

Devvrat Joshi
1 ● 1

-1

Follow the simple logic to solve this problem: Since both pointer A and B are traveling with **same speed**. To meet both at the same point they must be cover the **same distance**. and we can achieve this by **adding the length of a list to another**.

Share Improve this answer Follow

answered Jan 6, 2021 at 5:22

Uday Kumar
1 ● 1