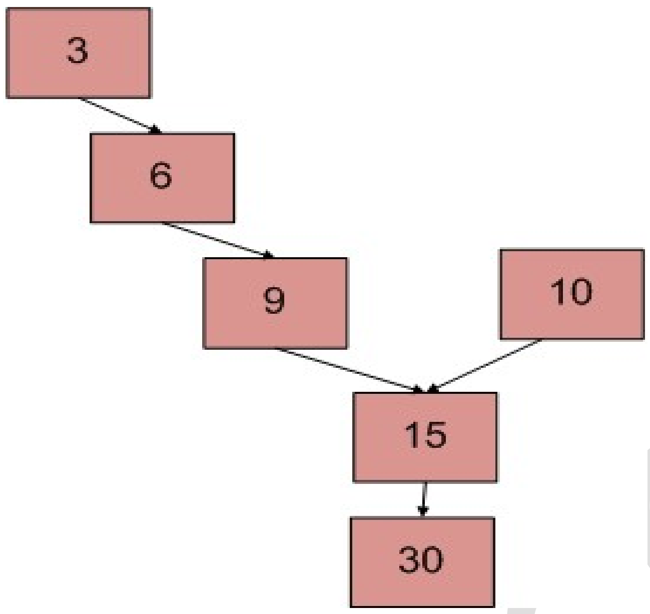
Find a point of intersection in a Y - shaped linked list.



Approach 1 :-

Brute force approach – In this approach we traverse the one linked list and check for each node present in it whether it is present in other Linked list.

Function Code :-

```
int intersectPoint(Node* head1, Node* head2)

{

   // Your Code Here

   Node *p1=head1, *p2=head2;

   while(p1)

   {

      p2=head2;

      while(p2 && p2!=p1)
```

```
            p2=p2->next;
        if(p1==p2)
            return p1->data;
        p1=p1->next;
    }
    return -1;
}
```

This approach takes **O(m * n)** time where m is the length of 1st linked list and n is the length of other LL and **O(1)** auxillary space complexity.

Aproach 2 :-

As the above approach is taking so much of time by traversing the 2nd LL again and again so we can improvise it using another approach.

Using stack - In this approach we travel each LL and store the address of each node in the stack and then when we have traversed each LL we will pop out the stack till we reach the last same pop out address. The last pop out address is the required address of Node.

Function Code :-

```
int intersectPoint(Node* head1, Node* head2)
{
    // Your Code Here
    Stack <Node *> k1,k2;
    Node *p1=head1, *p2=head2;
    while(p1)
    {
        k1.push(p1);
        p1=p1->data;
    }
    while(p2)
    {
        k2.push(p2);
        p2=p2->data;
    }
    p1=0;
    while(k1.top()==k2.top())
    {
        p1=k1.top();
        k1.pop();
        k2.pop();
    }
    if(p1)
        return p1->data;
    return -1;
}
```

This approach takes **O(m + n)** time as well as **O(m + n)** auxillary space complexity.

Approach 3 :-

As the above approach is using O(m + n) auxillary space by storing the address of each node so we will improvise it using another approach.

Using difference of node :- In this approach we will traverse both the LL and calculate the length of each LL. Now we will calculate the differrence of number of nodes in both the LL and traverse that number of nodes in bigger LL. Now both the LLs are equidistant from the point of interesection. Now we will traverse both the LL simultaneously till the address of Nodes of both the LL become same.

Code :-

```
#include<iostream>

#include<stdio.h>

using namespace std;


/* Link list Node */

struct Node

{

   int data;

   struct Node *next;

   Node(int x)

   {

      data = x;

      next = NULL;

   }

};


int intersectPoint(struct Node* head1, struct Node* head2);


Node* inputList(int size)

{

   if(size==0) return NULL;


   int val;
```

```
    cin>> val;

    Node *head = new Node(val);
    Node *tail = head;

    for(int i=0; i<size-1; i++)
    {
        cin>>val;
        tail->next = new Node(val);
        tail = tail->next;
    }

    return head;
}

/* Driver program to test above function*/
int main()
{
    int T,n1,n2,n3;

    cin>>T;
    while(T--)
    {
        cin>>n1>>n2>>n3;
        Node* head1 = inputList(n1);

        Node* head2 = inputList(n2);

        Node* common = inputList(n3);


        Node* temp = head1;

        while(temp!=NULL && temp->next != NULL)

            temp = temp->next;

        if(temp!=NULL) temp->next = common;


        temp = head2;

        while(temp!=NULL && temp->next != NULL)

            temp = temp->next;

        if(temp!=NULL) temp->next = common;


        cout << intersectPoint(head1, head2) << endl;

    }

    return 0;

}


// } Driver Code Ends



/* Linked List Node
```

```
struct Node {
  int data;
  struct Node *next;
  Node(int x) {
    data = x;
    next = NULL;
  }
}; */

/* Should return data of intersection point of two linked
   lists head1 and head2.
   If there is no intersecting point, then return -1. */
int intersectPoint(Node* head1, Node* head2)
{
    // Your Code Here
    int c1=0,c2=0;
    Node *p1=head1, *p2=head2;
    while(p1)
    {
        p1=p1->next;
        c1++;
    }
    while(p2)
    {
```

```
    p2=p2->next;
    c2++;
}
int d=abs(c1-c2);
p1=head1;
p2=head2;
if(c1>c2)
{
    while(d--)
        p1=p1->next;
}
else
{
    while(d--)
        p2=p2->next;
}
while(p1 && p2 && p1!=p2)
{
    p1=p1->next;
    p2=p2->next;
}
if(p1 && p2)
{
    return p1->data;
    }
    else
        return -1;
}
```

This approach takes **O(m + n)** time only and **O(1)** auxillary space complexity.