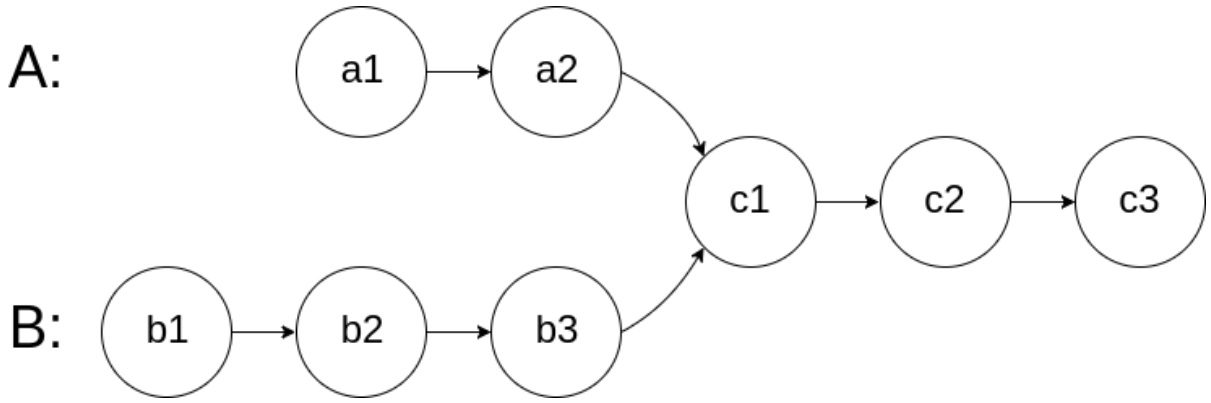
Striver's SDE Sheet

Day 4: Linked List Part-II

Intersection of Two Linked Lists

Given the heads of two singly linked-lists headA and headB, return *the node at which the two lists intersect*. If the two linked lists have no intersection at all, return null.

For example, the following two linked lists begin to intersect at node c1:



Ans:

Observation and approach :

Approach 1

1. Find the length of both the linked lists.
2. Traverse the bigger linked list until the remaining nodes count becomes equal to the smaller one's.
3. Traverse both the heads together. If both of them are same then the intersecting node is found.

Code:

```
int length(ListNode *head){
        int len = 0;
        while(head){
            len++;
            head = head->next;
        }
        return len;
```

```
}

ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
    if(!headA || !headB) return NULL;

    //step1
    int lenA = length(headA), lenB = length(headB);

    //step2
    if(lenA>lenB){
        while(lenA>lenB){
            headA = headA->next;
            lenA--;
        }
    }
    else if(lenA<lenB){
        while(lenA<lenB){
            headB = headB->next;
            lenB--;
        }
    }

    //step 3
    while(headA && headB){
        if(headA==headB) return headA;
        headA = headA->next;
        headB = headB->next;
    }
    return NULL;
}
```

Approach 2

In this approach, we can simply convert this problem into a loop problem.

1. Find the tail.
2. Connect the tail with any of the head which creates a loop.
3. Using the other head, find intersection point of the loop.
4. Undo the loop, by setting tail->next = NULL.
5. Return the intersection node.

Code:

```
ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
    //getting the tail
    ListNode* tail = headA;
    while(tail->next){
        tail = tail->next;
    }

    //creating a loop
    tail->next = headA;

    //detecting and finding the intersection
    ListNode *fast = headB, *slow = headB;

    while(fast && fast->next){
        slow = slow->next;
        fast = fast->next->next;

        if(slow==fast) {
            slow = headB;
            while(slow!=fast){
                slow = slow->next;
                fast = fast->next;
            }
            //undoing the loop
            tail->next = NULL;
```

```
            return slow;
        };
    }
    tail->next = NULL;
    return NULL;
}
```

TUF approach:

```
ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
         if (headA==NULL || headB==NULL)
        {
            return NULL;
        }
        ListNode *ptrA=headA, *ptrB=headB;
        while(ptrA!=NULL || ptrB!=NULL)
        {
            if(ptrA==NULL)
                ptrA=headB;
            if(ptrB==NULL)
                ptrB=headA;
            if (ptrA==ptrB)
                return ptrA;
            ptrA=ptrA->next;
            ptrB=ptrB->next;
        }
        return NULL;
    }
```

Linked List Cycle

Given head, the head of a linked list, determine if the linked list has a cycle in it.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer.
Internally, pos is used to denote the index of the node
that tail's next pointer is connected to. Note that pos is not passed as a parameter.

Return true *if there is a cycle in the linked list*. Otherwise, return false.

Ans:

Standard fast and slow approach :

```
    bool hasCycle(ListNode *head) {
          if(head == NULL)
            return false;
        ListNode *fast = head;
        ListNode *slow = head;



        while(fast != NULL && fast ->next != NULL)
        {
            fast = fast->next->next;
            slow = slow->next;
            if(fast == slow)
                return true;
        }
        return false;
    }
```

using hash table :

I used unordered set in this question, because i need only one parameter which is reference to node. While head is not nullptr, check map for current node is in already in table or not.

If set has current node reference, it means there is a cycle here. If not we can reach to tail of the list, after that current node becomes nullptr and exits the loop and returns false.

Code:

```
bool hasCycle(ListNode *head) {
        unordered_set<ListNode*> set;
        while (head!=nullptr) //until head is null
        {
            if (set.count(head)==0) //check for repetition
            {
                set.insert(head); //if not, add reference to set
            }
            else //if repetition is found
            {
                return true;
            }
            head=head->next; //go next node
        }
        //if we can reach here, means that there is no cycle
        return false;
    }
```

Reverse Nodes in k-Group

Given the head of a linked list, reverse the nodes of the list k at a time, and return *the modified list*.

k is a positive integer and is less than or equal to the length of the linked list. If the number of nodes is not a multiple of k then left-out nodes, in the end, should remain as it is.

You may not alter the values in the list's nodes, only nodes themselves may be changed.

Ans:

Observation and approach:

Combination reverse linked list and some what deletion of node mean that pointer part

Code:

```
int length (ListNode* head)
    {
        int len = 0;
        while(head != NULL)
        {
            len++;
            head = head -> next;
        }
        return len;
    }


    ListNode* reverseKGroup(ListNode* head, int k) {
        int len = length(head);
        if(len < k) //As mentioned in aue, if len < k don't reverse
        {
            return head;
        }
        int cnt = 0;
        ListNode* curr = head; //1 --- After 1st step, curr = 2
        ListNode* prev = NULL; //NULL
        ListNode* forward = NULL;
```

```
        while(curr != NULL && cnt < k)
        {
            forward = curr -> next; //2 --- 3
            curr -> next = prev;
            prev = curr; //prev = 1 --- prev = 2
            curr = forward; // curr = 2 --- curr = 3
            cnt++;
        }
        if(forward != NULL)
        {
            head -> next = reverseKGroup(forward, k); //Recursively
calling for remaining nodes
        }
        //I've stored it in head -> next bcz, head = 1 and I've
coneected it with 4, head of the new LL


        return prev; // return prev bcz, 2 is the head of our final LL
and it is stored in prev
    }
```

Palindrome Linked List

Given the head of a singly linked list, return true *if it is a*

Palindrome or false *otherwise*.

Ans:

Observation and approach :

Just reverse the list and maintain a pointer to check if both equal then return true

We can use slow and fast pointer to solve the problem

Code:

```
bool isPalindrome(ListNode* head) {
         ListNode *slow=head;
        ListNode *fast=head;
        while(fast->next!=NULL && fast->next->next!=NULL){
            slow=slow->next;
            fast=fast->next->next;
        }
        ListNode *mid=slow->next;
        ListNode *curr=mid;
        ListNode *prev=NULL;
        while(curr!=NULL){
            ListNode *next=curr->next;
            curr->next=prev;
            prev=curr;
            curr=next;
        }
        while(prev!=NULL && head!=mid){
            if(head->val!=prev->val)
                return false;
            prev=prev->next;
            head=head->next;
        }
        return true;
    }
```

Linked List Cycle II

Given the head of a linked list, return *the node where the cycle begins. If there is no cycle*, *return* null.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer.
Internally, pos is used to denote the index of the node that
tail's next pointer is connected to (0-indexed). It is -1 if there is no cycle. Note that pos is not passed as a parameter.

Do not modify the linked list.

Ans:

Observation and approach:

Fast and slow approach

Code:

```
 ListNode *detectCycle(ListNode *head) {
        unordered_set<ListNode*>s;
        ListNode* curr=head;
        while(curr!=NULL){
            if(s.find(curr)!=s.end()){
                return curr;
            }else{
                s.insert(curr);
                curr=curr->next;
            }
        }
        return NULL;
    }
```

Flattening a Linked List

Given a Linked List of size N, where every node represents a sub-linked-list and contains two pointers:
(i) a next pointer to the next node,
(ii) a bottom pointer to a linked list where this node is head.
Each of the sub-linked-list is in sorted order.
Flatten the Link List such that all the nodes appear in a single level while maintaining the sorted order.

Note: The flattened list will be printed using the bottom pointer instead of the next pointer.
For more clarity have a look at the printList() function in the driver code.

Ans:

code:

```
Node * Merge ( Node* r1, Node* r2)
{  r1->next=NULL;
    Node* temp =new Node(0);
    Node* res=temp;

    while(r1 && r2)
    {
        if(r1->data<=r2->data)
        {
            temp->bottom=new Node(0);
            temp->bottom->data=r1->data;
             r1=r1->bottom;

        }
        else{
             temp->bottom=new Node(0);
             temp->bottom->data=r2->data;
             r2=r2->bottom;
        }
        temp=temp->bottom;
    }
```

```
    while(r1)
    {     temp->bottom=new Node(0);
          temp->bottom->data=r1->data;
          r1=r1->bottom;
          temp=temp->bottom;

    }

    while(r2)
    {    temp->bottom=new Node(0);
          temp->bottom->data=r2->data;
          r2=r2->bottom;
          temp=temp->bottom;



    }
    return res->bottom;
}
Node *flatten(Node *root)
{   if(root==NULL || root->next==NULL) return root;
    if(root->next->next==NULL)
    {
      root=Merge(root,root->next);
      root->next=NULL;
      return root;
    }
    return Merge(root,flatten(root->next));
   }
```

$ MOST ODF THE PROBLEM ARE SOLVED BY FAST AND SLOW POINTER METHODS

$ MOST OF THE PROBLEM (98%) ARE PROCESS BASED QUESTION MEANS JUST HAVE TO DO WHAT EVER IT SAY (brute and {fast and slow start})