I have discussed two main approaches to solve this problem in C++

1.Two Pointer Approach

temp1: pointer pointing to elements of list 1
temp2: pointer pointing to elements of list 2
resulthead: pointer storing the head of output list

- Initially an important point to check firstly is none of the list is empty. If either of the lists is empty we cannot traverse the list and NULL must be returned to handle the corner cases.

```
if(head1==NULL || head2==NULL)
        return NULL;
```

Three cases are formed :

Case 1: Both the elements of the list are equal

- If temp1==temp2 means that the element is present in both the lists and this must be added to the Intersecting result list.
  --If initially the output list is empty head must be created so there will be two cases
  a. The output list is empty – We need to create the head of the linked list.

```
if(resulthead==NULL)
        {
            Node* newNode=new Node(temp1->data);
            newNode->next=NULL;
            resulthead=newNode;
            temp3=resulthead;
        }
```

b. The output list already contains some elements – New node must be added at the end of the list.

```
else
        {
            Node* newNode=new Node(temp1->data);
            newNode->next=NULL;
            temp3->next=newNode;
            temp3=newNode;
        }
```

At last both the pointers must be moved to point to the next element in the list.

```
temp1=temp1->next;
        temp2=temp2->next;
```

Case 2: Element of list1 is less than element of list 2

To find whether the element in list2 currently being pointed occurs in the sorted list1 we need to move to the next element of list1.

```
else if(temp1->data<temp2->data)
    {
        temp1=temp1->next;
    }
```

Case 3: Element of list1 is greater than element of list 2

To find whether the element in list1 currently being pointed occurs in the sorted list2 we need to move to the next element of list2.

```
else
    {
        temp2=temp2->next;
    }
```

Finally, we return the resulthead.

Analysis of the Algorithm:

1. **Time Complexity:** O (m + n) where m and n are number of nodes in first and second linked lists respectively.
2. **Space Complexity:** O (min (m, n)). The output list can store at most min (m, n) nodes.

2. Recursive Approach

temp1: pointer pointing to elements of list 1

temp2: pointer pointing to elements of list 2
temp: pointer storing the head of output list

- Initially an important point to check firstly is none of the list is empty. If either of the lists is empty we cannot traverse the list and NULL must be returned to handle the corner cases.

```
if(head1==NULL || head2==NULL)
        return NULL;
```

Case 1: Element of list1 is less than element of list 2

A recursive call is made by passing the next element of list1 to check whether it matches the current element of list2.

```
if(temp1->data<temp2->data)
        return findIntersection (temp1->next, temp2);
```

Case 2: Element of list1 is greater than element of list 2

A recursive call is made by passing the next element of list2 to check whether it matches the current element of list1.

```
if(temp1->data>temp2->data)
        return findIntersection (temp1, temp2->next);
```

Case 3: Element of list1 is equal to element of list 2

A new node is created and added to temp list.

```
Node* temp=new Node(temp1->data);
    temp->next=findIntersection (temp1->next, temp2->next);
```

Finally, the list is returned.

Analysis of the Algorithm:

1. **Time Complexity:** O (m + n) where m and n are number of nodes in first and second linked lists respectively.
2. **Space Complexity:** O (max (m, n)). The output list can store at most max (m, n) nodes.