

---

**I: Insert Before**

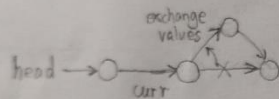
---

This operation has been discussed in detail. Given a new data and the target node, create a new node using the new data and insert the new node before the target node. You should implement this function with  $O(1)$  time complexity.

- Assume the target node is neither head nor tail.

---

```
void InsertBefore (Node *curr, int data)
{
    Node *newNode = new Node( curr->data, curr->next );
    curr->next = newNode;
    curr->data = data;
}
```



**2: Remove Duplication**

Given a linked list in ascending order, you should remove all the nodes with duplicate data. Traverse the list for at most once. For example:

Before:

list: 1 -> 2 -> 2 -> 3 -> 3 -> NULL

After:

list: 1 -> 2 -> 3 -> NULL

```
void RemoveDuplication (List *list)
```

```
{
```

```
    Node *curr = list->head;
```

```
    if (curr == NULL)
```

```
    {
```

```
        return;
```

```
    }
```

```
    while (curr->next != NULL)
```

```
    {
```

```
        if (curr->data == curr->next->data)
```

```
        {
```

```
            Node *p = curr->next->next;
```

```
            if (curr->next == list->tail)
```

```
            {
```

```
                list->tail = curr;
```

```
            }
```

```
            delete curr->next;
```

```
            curr->next = p;
```

```
        }
```

```
    else
```

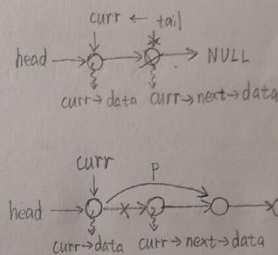
```
    {
```

```
        curr = curr->next;
```

```
    }
```

```
    }
```

```
}
```



**3: Move Head**

Given a non-empty linked list `src` and a linked list `dst`. Move the head node of `src` to the tail of `dst`. For example:

Before:

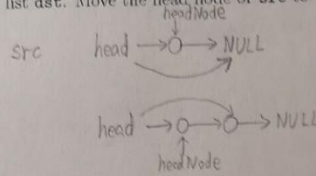
`src: 1 -> 2 -> 3 -> NULL`

`dst: 4 -> 5 -> 6 -> NULL`

After:

`src: 2 -> 3 -> NULL`

`dst: 4 -> 5 -> 6 -> 1 -> NULL`



```
void MoveHead (List *src, List *dst)
{
    Node *headNode = src->head;
    assert(headNode);

    src->head = src->head->next;

    if (headNode->next == NULL)
    {
        src->tail = NULL;
    }

    headNode->next = NULL;

    if (dst->head == NULL)
    {
        dst->head = headNode;
    }
    else
    {
        dst->tail->next = headNode;
    }

    dst->tail = dst->tail->next;
}
```

---

**4: Alternate Split**

---

Given a linked list `src`, split it to two linked list `a` and `b` in alternating order. For example:

Before:

`src: 1 -> 2 -> 3 -> 4 -> 5 -> NULL`

`a: NULL`

`b: NULL`

After:

`src: NULL`

`a: 1 -> 3 -> 5 -> NULL`

`b: 2 -> 4 -> NULL`

- You may assume `src` has more than 4 elements, i.e. after this function, `head` and `tail` of `a` and `b` point to different nodes.
- `a` and `b` are initially empty.
- Use `MoveHead` to implement this function.

---

```
void AlternateSplit (List *src, List *a, List *b)
{
    assert (a == NULL && b == NULL);

    while (src->head != NULL)
    {
        MoveHead (src, a);

        if (src->head != NULL)
        {
            MoveHead (src, b);
        }
    }
}
```

```
void SortedMerge (List *a, List *b, List *dst)
{
```

```
    assert(dst->head == NULL);
```

```
    assert(a->head != NULL && b->head != NULL);
```

```
    while (true)
```

```
    {
```

```
        if (a->head == NULL)
```

```
        {
```

```
            if ( b->head->next != NULL )
```

```
            {
```

```
                MoveHead ( b, dst ) ;
```

```
                SortedMerge ( a, b, dst ) ;
```

```
                b->head = b->tail = NULL;
```

```
            }
```

```
            break ;
```

```
        }
```

```
        if (b->head == NULL)
```

```
        {
```

```
            // similar to above, ignore it
```

```
            ... ..
```

```
        }
```

```
        if (a->head->data < b->head->data)
```

```
        {
```

```
            MoveHead ( a, dst ) ;
```

```
        }
```

```
        else
```

```
        {
```

```
            MoveHead ( b, dst ) ;
```

```
        }
```

```
    }
```

```
}
```

a head → NULL

b head → ○

dst head → ○

a head → ○ →

b head → ○ →

dst head → ○ → ○ →