CX1007: TUTORIAL (WEEK 10)

LINKED LIST FUNCTIONS

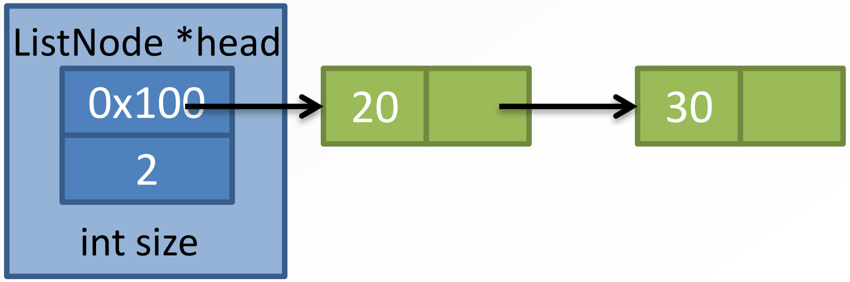
- **Original function prototypes (using ListNode struct):**
  - `void printList(ListNode *head);`
  - `ListNode * findNode(ListNode *head, int index);`
  - `int insertNode(ListNode **ptrHead, int index, int value);`
  - `int removeNode(ListNode **ptrHead, int index);`

- **New function prototypes (Using LinkedList struct):**
  - `void printList(LinkedList *ll);`
  - `ListNode * findNode(LinkedList *ll, int index);`
  - `int insertNode(LinkedList *ll, int index, int value);`
  - `int removeNode(LinkedList *ll, int index);`

LINKEDLIST C STRUCT

- Implementation of Linked List
  - Define another C struct, LinkedList
  - Wrap up all elements that are required to implement the Linked List data structure

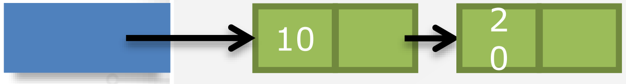
```
typedef struct _linkedlist{
    ListNode *head;
    int size;
} LinkedList;
```



- Why is this useful?
  - Consider the rewritten Linked List functions

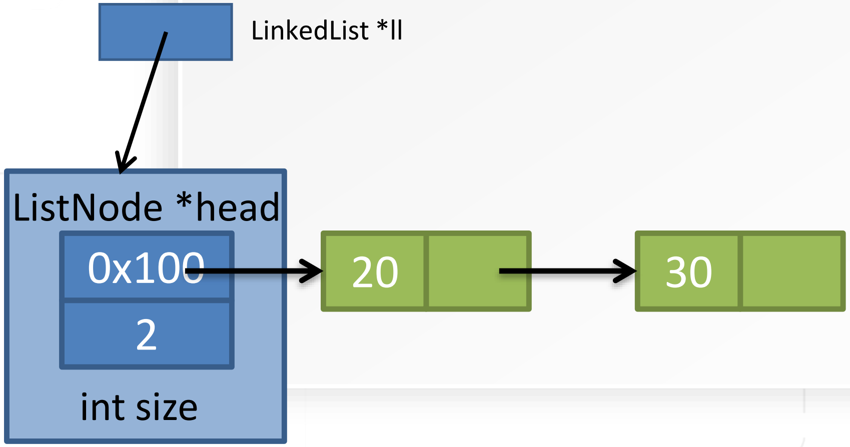
printList() Versions

```
typedef struct _listnode{
  int item;;
  struct _listnode *next;
}LinkedList;
```



```
void printList(ListNode *head){

    if (head == NULL)
        return;

    while (head != NULL){
        printf("%d ", head->item);
        head = head->next;
    }
    printf("\n");
}
```

```
typedef struct _linkedlist{
   int size;
   ListNode *head;
}LinkedList;
```

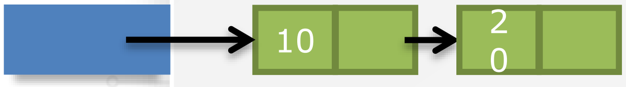


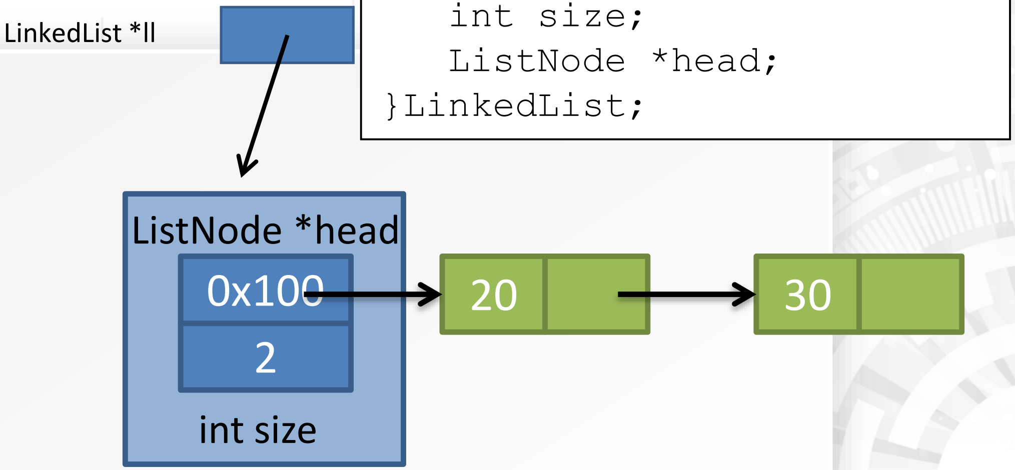
```
void printList(LinkedList *ll){
    ListNode *temp = ll->head;

    if (temp == NULL)
        return;

    while (temp != NULL){
        printf("%d ", temp->item);
        temp = temp->next;
    }
    printf("\n");
}
```

findNode() Versions

```
typedef struct _listnode{
  int item;;
  struct _listnode *next;
}LinkedList;
```

```
typedef struct _linkedlist{
   int size;
   ListNode *head;
}LinkedList;
```



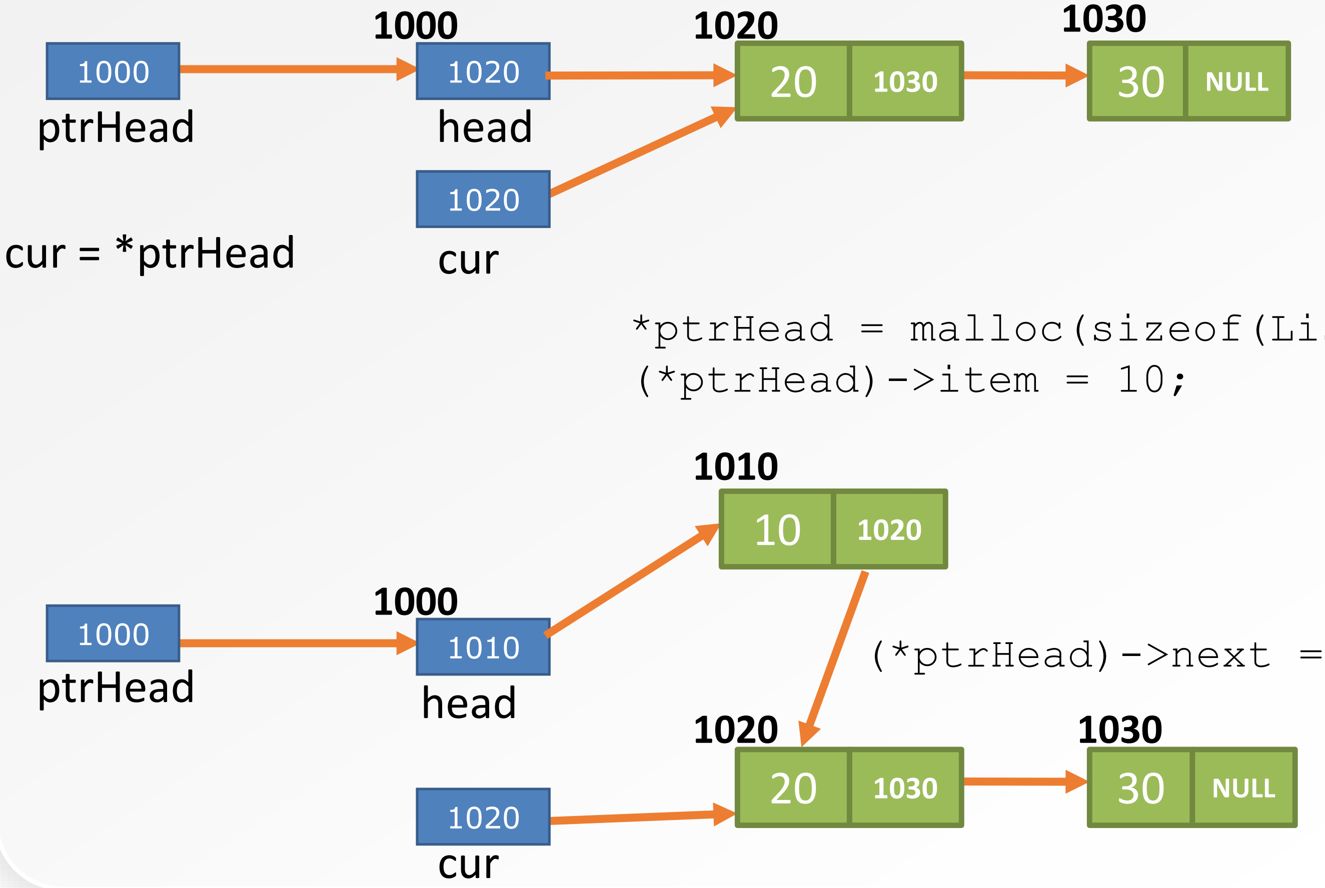


```
ListNode * findNode(
    ListNode *head, int index){

    if (head == NULL || index < 0)
        return NULL;

    while (index > 0){
        head = head->next;
        if (head == NULL)
            return NULL;
        index--;
    }
    return head;
}
```

```
ListNode * findNode(
    LinkedList *ll, int index){
    ListNode *temp = ll->head;
    if (temp == NULL || index < 0)
        return NULL;

    while (index > 0){
        temp = temp->next;
        if (temp == NULL)
            return NULL;
        index--;
    }
    return temp;
}
```

INSERTING A NODE AT THE FRONT

```
int insertNode(ListNode **ptrHead, int index, int value)
```



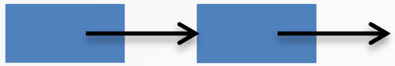
```
cur = *ptrHead
```

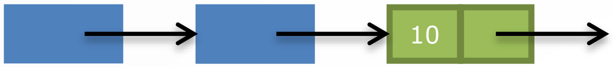
```
*ptrHead = malloc(sizeof(ListNode));
(*ptrHead)->item = 10;
```

```
(*ptrHead)->next = cur;
```

insertNode() Using ListNode STRUCT

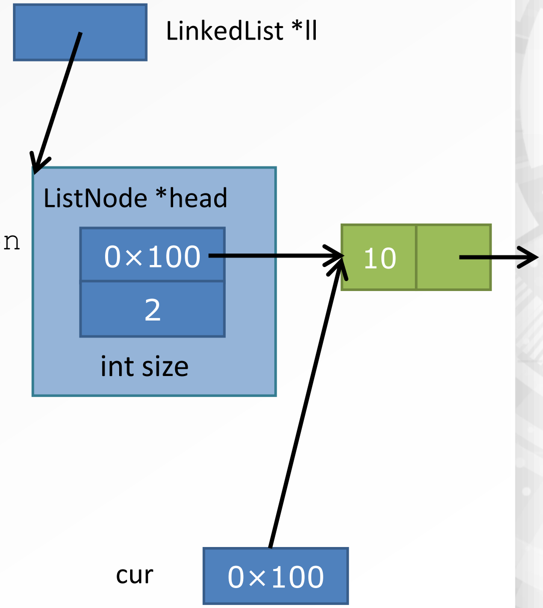
```
int insertNode(ListNode **ptrHead, int index, int value){

    ListNode *pre, *cur;

    // If empty list or inserting first node, need to update head pointer
    if (*ptrHead == NULL || index == 0){
        cur = *ptrHead;
        *ptrHead = malloc(sizeof(ListNode));
        (*ptrHead)->item = value;
        (*ptrHead)->next = cur;
        return 0;
    }

    // Find the nodes before and at the target position
    // Create a new node and reconnect the links
    if ((pre = findNode(*ptrHead, index-1)) != NULL){
        cur = pre->next;
        pre->next = malloc(sizeof(ListNode));
        pre->next->item = value;
        pre->next->next = cur;
        return 0;
    }

    return -1;
}
```





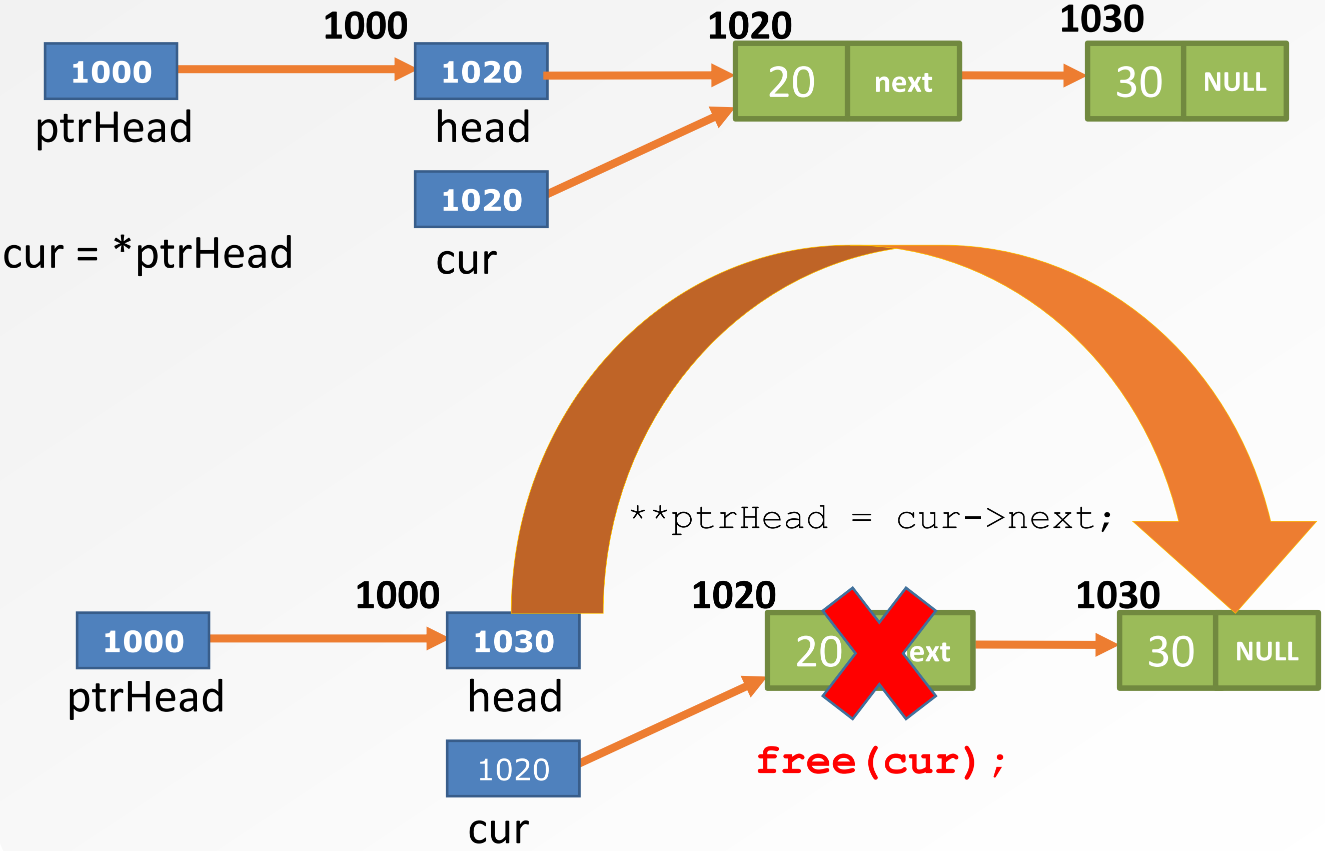
insertNode() LinkedList STRUCT

```c
int insertNode(LinkedList *ll, int index, int value){
    ListNode *pre, *cur;

    if (ll == NULL || index < 0 || index > ll->size + 1)
          return -1;
// If empty list or inserting first node, need to update head pointer
    if (ll->head == NULL || index == 0){
       cur = ll->head;
       ll->head = malloc(sizeof(ListNode));
       ll->head->item = value;
       ll->head->next = cur;
       ll->size++;
       return 0;
    }
// Find the nodes before and at the target position
// Create a new node and reconnect the links
    if ((pre = findNode(ll, index - 1)) != NULL){
       cur = pre->next;
       pre->next = malloc(sizeof(ListNode));
       pre->next->item = value;
       pre->next->next = cur;
       ll->size++;
       return 0;
    }
    return -1;
}
```



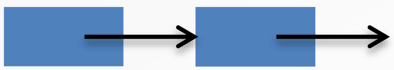
REMOVING A NODE AT THE FRONT

```
int removeNode(ListNode **ptrHead, int index);
```



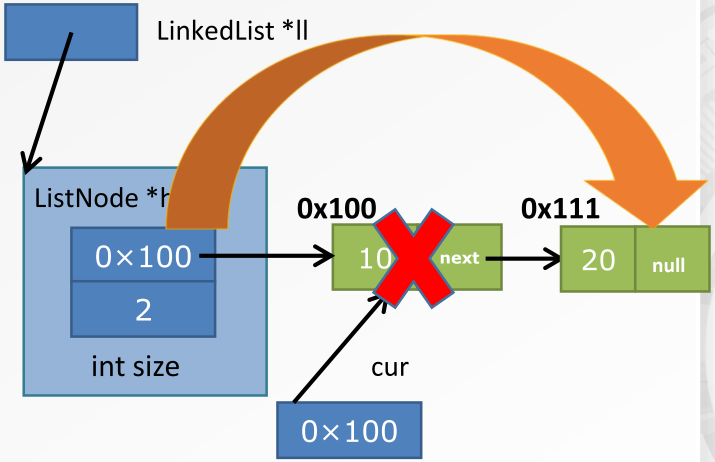
removeNode() Using ListNode STRUCT

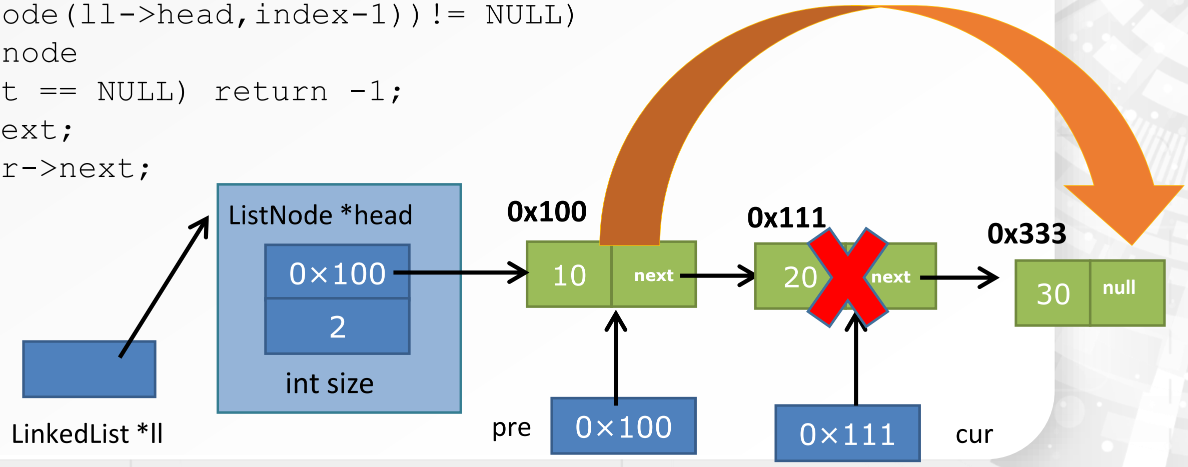
```
 int removeNode(ListNode **ptrHead, int index){
     ListNode *pre, *cur;
     // Sanity check for empty list
     if (*ptrHead == NULL)
          return -1;
     // If removing first node, need to update head pointer
     if (index == 0){
          cur = *ptrHead;
          *ptrHead = cur->next;
          free(cur);
          return 0;
     }
     // Find the nodes before and after the target position
     // Free the target node and reconnect the links
     if ( (pre = findNode(*ptrHead, index-1))!= NULL){
          if (pre->next == NULL) return -1;
          cur = pre->next;
          pre->next=cur->next;
          free(cur);
          return 0;
          }
     return -1;
}
```



removeNode() LinkedList STRUCT

```
int removeNode2(LinkedList *ll, int index) {
        ListNode *cur, *pre;

        if (ll->head ==NULL) return -1;

        if (index==0)
        {
                cur=ll->head;
                ll->head =cur->next;
                free(cur);
                ll->size --;
                return 0;
        }

        if ( (pre = findNode(ll->head,index-1))!= NULL)
        {//not the first node
                if (pre->next == NULL) return -1;
                cur = pre->next;
                pre->next=cur->next;
                free(cur);
                ll->size--;
                return 0;
        }

        return -1;
}
```

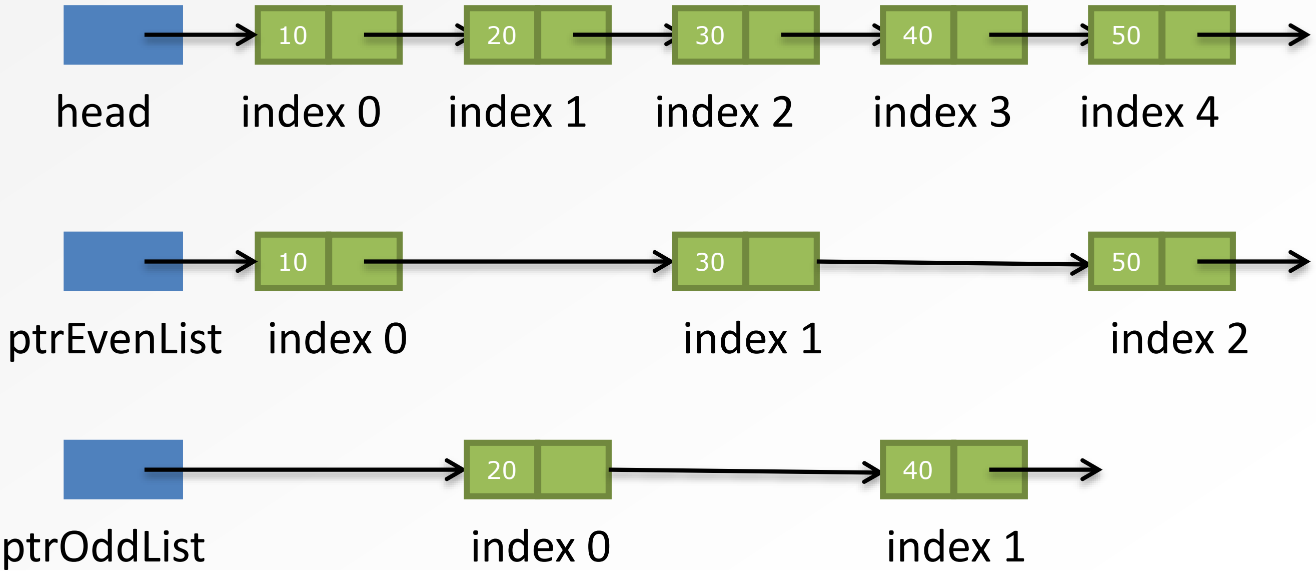




SPLIT(LISTNODE *HEAD, LISTNODE **PTREVENLIST, LISTNODE **PTRODDLIST);

Write a function split() that copies the contents of a linked list into two other linked lists. The function prototype is given below:

split(ListNode *head,
ListNode **ptrEvenList,
ListNode **ptrOddList);



split(ListNode *head, ListNode **ptrEvenList, ListNode **ptrOddList);

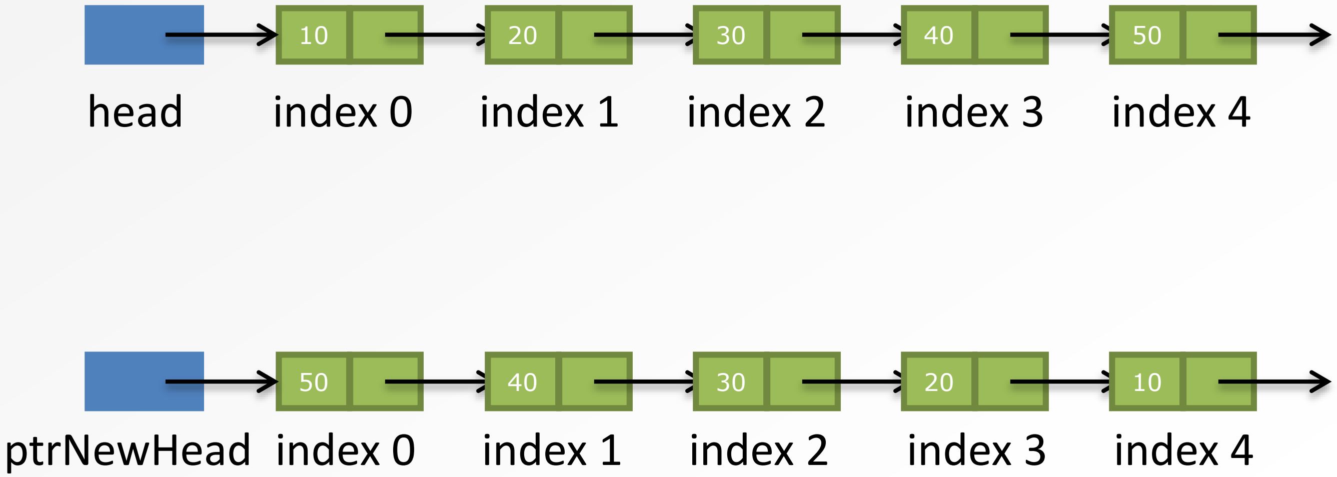
```
int split(ListNode *head, ListNode **ptrEvenList, ListNode **ptrOddList)
{
          int even = 1, evenSize = 0, oddSize = 0;
          ListNode *cur=head;

          if (cur == NULL)
                    return -1;
          while (cur!= NULL){
                    if (even==1){
                              insertNode(ptrEvenList, evenSize, cur->num);
                              evenSize++;
                    }
                    else{
                              insertNode(ptrOddList, oddSize, cur->num);
                              oddSize++;
                    }
          cur = cur ->next;
          even = -even;
          }
          return 0;
}
```

int insertNode(ListNode **ptrHead, int index, int value)

DUPLICATEREVERSE()

Write a function duplicateReverse() that creates a duplicate of a linked list with the nodes stored in reverse. The function prototype is given below:

```
int duplicateReverse(ListNode *head, ListNode **ptrNewHead);
```



duplicateReverse()

```
int duplicateReverse(ListNode *head, ListNode **ptrNewHead){
        ListNode *cur=head;

        if (cur == NULL) return -1;
        // Simply traverse the list and insert each visited
        // node into the new list at index 0 each time

        while (cur != NULL){

                if (insertNode(ptrNewHead, 0, cur->num) == -1)
                        return -1;
                cur = cur ->next;
        }
        return 0;
}
```

```
int insertNode(ListNode **ptrHead, int index, int value)
```