6

已知一个单向链表如图所示，试给出一个复制该链表的算法。

```
void copy_linklist(node* origin, node* target) {
    // copy origin to target
    node* ptr = target;
    for(node* p = origin->next; p != NULL; p = p->next) {
        ptr->next = (node*) malloc(sizeof(node));
        ptr = ptr->next;
        ptr->val = p->val;
        ptr->next = NULL;
    }
}
```

14

设有一个带有表头的链式线性表，表头地址为 head，试给出相当于栈的插入操作和删除操作的函数。

```
void stack_push(node* head, int val) {
    node* newnode = (node*) malloc(sizeof(node));
    newnode->val = val;
    newnode->next = head->next;
    head->next = newnode;
}

int stack_top(node* head) {
    return head->next == NULL?NULL:head->next->val;
}

void stack_pop(node* head) {
    if(head->next != NULL) {
        node* temp = head->next;
        head->next = head->next->next;
        free(temp);
    }
}
```

16

设有两个栈共享空间 C[0...m-1]，其中一个栈底设在 C[0] 处，另一个栈底设在 C[m-1] 处，试写一个对任意一个栈 i 做进栈和出栈操作 Push(x, i) 和 Pop(i) 的算法，其中 i = 1, 2。注意：仅当整个空间 C[0...m-1] 占满时才产生上溢。

```
int size1 = 0;
int size2 = 0;
void Push(int x, int target) {
    if(size1 + size2 < m) {
        if(target == 1) {
            size1 ++;
            c[size1 - 1] = x;
        } else {
            size2 ++;
            c[m - size2] = x;
        }
    }
}

void Pop(int target) {
    if(target == 1) {
        size1 --;
    } else {
        size2 --;
    }
}

int Top(int target) {
    if(target == 1) {
        return s[size1 - 1];
    } else {
        return s[SIZE - size2];
    }
}
```

17

试写出计算单向链表长度的算法。

```
int length(node* head) {
    int ans = 0;
    for(node* ptr = head->next; ptr != NULL; ptr = ptr->next) {
        ans ++;
    }
    return ans;
}
```

19

写一个算法将一个单向链表拆分成两个循环链表，并将每个循环链表的长度存入表头结点的数据域中。拆分的规则如下：第一个循环链表包含单向链表的第1、3、5...个结点，而第二个循环链表包含单向链表的第2、4、6...个节点

```
void work(node* ori, node** l1, node** l2) {
    // origin linklist is ori(ori is the head)
    // new linklist is l1, l2
    // now l1 == NULL and l2 == NULL;
    int target = 1;
    for(node* ptr = ori->next; ptr != NULL; ptr = ptr->next) {
        if(target == 1) {
            node* temp = (node*) malloc(sizeof(node));
            temp->val = ptr->val;
            if((*l1) == NULL) {
                (*l1) = temp;
                (*l1)->next = (*l1);
            } else {
                temp->next = (*l1)->next;
                (*l1)->next = temp;
                (*l1) = (*l1)->next;
            }
        } else {
            node* temp = (node*) malloc(sizeof(node));
            temp->val = ptr->val;
            if((*l2) == NULL) {
                (*l2) = temp;
                (*l2)->next = (*l2);
            } else {
                temp->next = (*l2)->next;
                (*l2)->next = temp;
                (*l2) = (*l2)->next;
            }
        }
        target = 3 - target;
    }
}
```