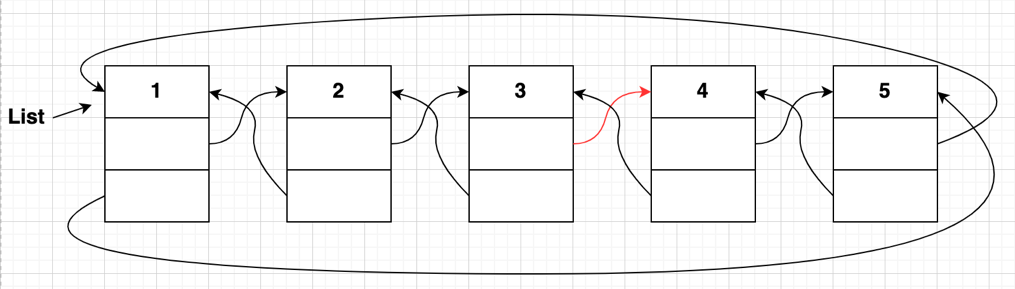
|  |
| --- |
| **CHAPTER 4 HOMEWORK** |

**PROBLEM 1:**

1. **list->next->next->next = list->prev;**



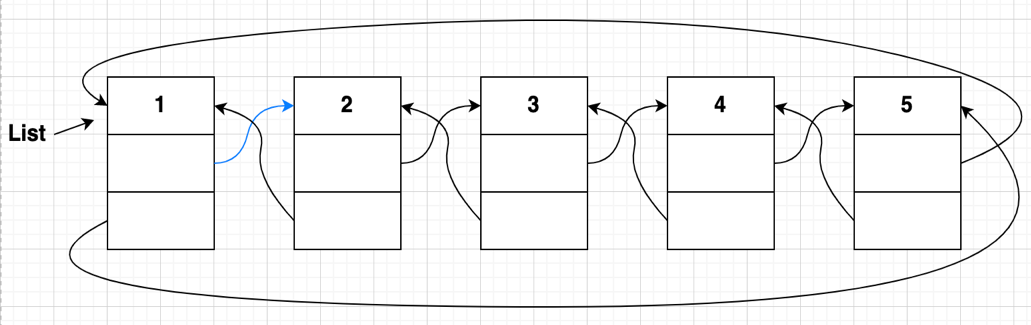
1. **list->prev->prev->prev = list->next->next->next->prev;**

(No change in linked list)

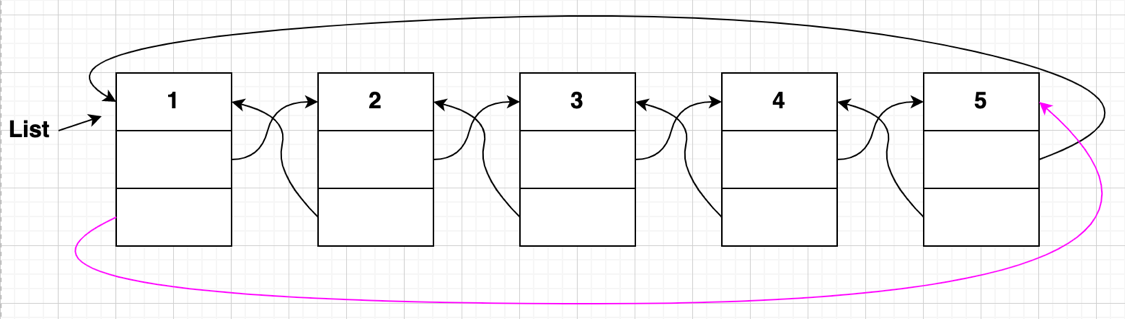
1. **list->next->next->next->prev = list->prev->prev->prev;**

(No change in linked list)

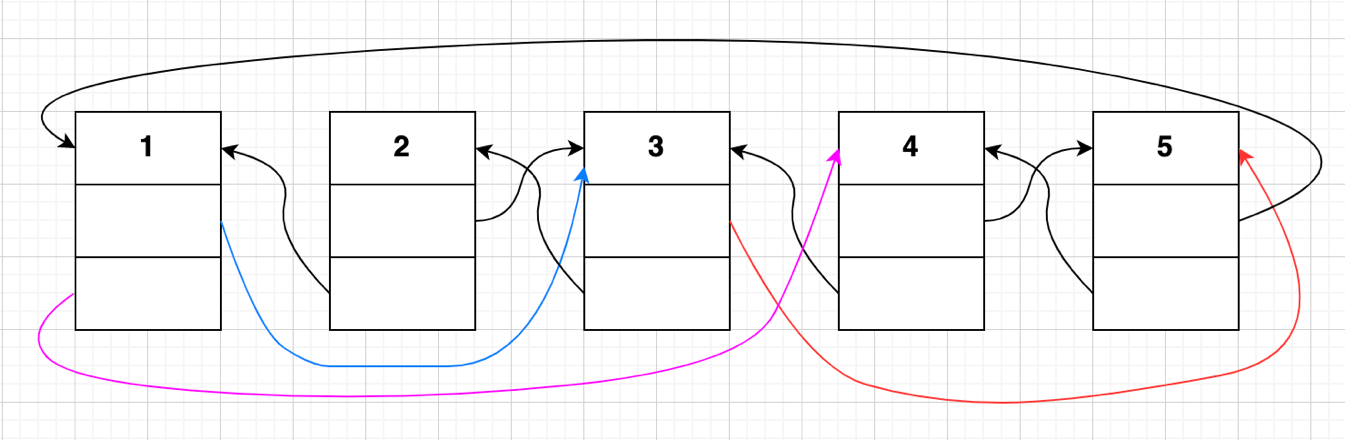
1. **list->next = list->next->next;**

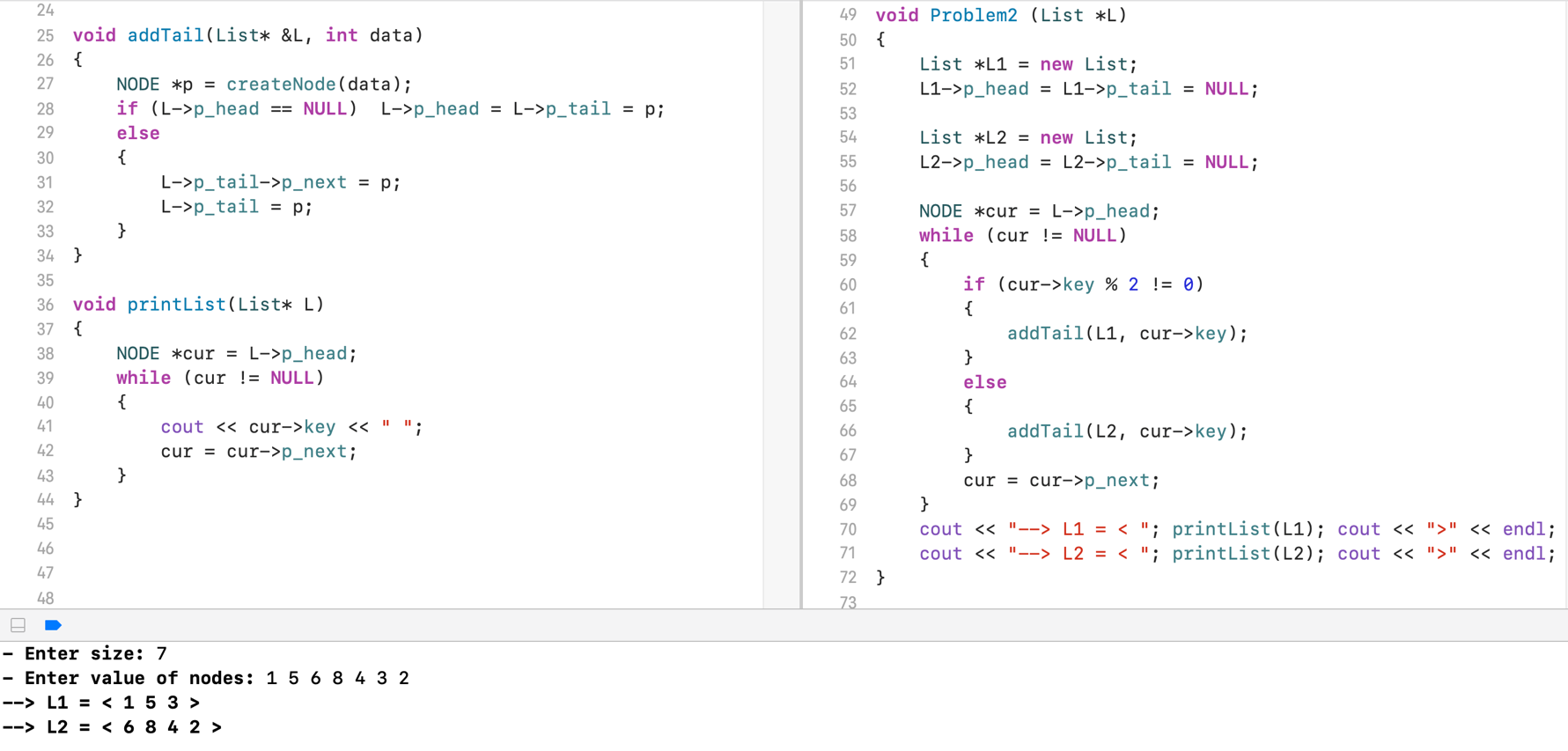


1. **list->next->prev->next = list->next->next->next;**



Danh sách liên kết sau khi thực hiện a, b, c, d, e



**PROBLEM 2:**

**PROBLEM 3:**

**struct** NODE

{

**int** key;

NODE \*next;

};

**struct** List

{

NODE \*head;

NODE \*tail;

};

NODE\* createNode(**int** data)

{

NODE \*p = **new** NODE;

p->key = data;

p->next = **NULL**;

**return** p;

}

**//a. Count the number of nodes in L**

**void** CountNumberOfNode (List \*L){

NODE \*cur = L->head;

**int** count = 0;

**while** (cur != **NULL**)

{

count++;

cur = cur->next;

}

cout << count << endl;

}

**//b. Search for the ith node in L, return the address of that node if found**

NODE \*getAtIndex (List \*L, **int** index)

{

NODE \*cur = L->head;

**for** (**int** i = 0; i < index; i++)

{

cur = cur->next;

}

**return** cur;

}

**//c. Insert a node x after a node k in L**

**void** addTail(List\* &L, NODE \*p)

{

**if** (L->head == **NULL**) L->head = L->tail = p;

**else**

{

L->tail->next = p;

L->tail = p;

}

}

**void** addAfter (List \*L, NODE \*k, NODE \*x)

{

**if** (L->head == **NULL**) addTail(L, x);

**if** (L->head->next == **NULL**) addTail(L, x);

**else**

{

x->next = k->next;

k->next = x;

}

}

**//d. Delete the node before the node k in L**

**void** deleteNodeBefore (List \*L, NODE \*k)

{

NODE \*cur = L->head;

**while** (cur->next->next->key != k->key)

{

cur = cur->next;

}

NODE \*p = cur->next;

cur->next = k;

**delete** p;

}

**//e. Reverse L**

**void** ReverseList (List \*L)

{

NODE \*fol = **new** NODE;

NODE \*prev = **NULL**;

NODE \*cur = L->head;

**while** (cur != **NULL**)

{

fol = cur->next;

cur->next = prev;

prev = cur;

cur = fol;

}

L->head = prev;

}

**PROBLEM 4:**

**bool** HaveTheSameSize(List \*L1, List \*L2)

{

**int** count1 = 0;

NODE \*p1 = L1->head;

**while** (p1 != **NULL**)

{

count1++;

p1 = p1->next;

}

**int** count2 = 0;

NODE \*p2 = L2->head;

**while** (p2 != **NULL**)

{

count2++;

p2 = p2->next;

}

**if** (count1 == count2)

{

**return** 1;

}

**return** 0;

}

**bool** PROBLEM4 (List \*L1, List \*L2)

{

**if** (HaveTheSameSize(L1, L2))

{

NODE \*p = L1->head;

NODE \*q = L2->head;

**while** (p != **NULL** || q != **NULL**)

{

**if** (p->key == q->key)

{

p = p->next;

q = q->next;

}

**else**

{

**return** 0;

}

}

**return** 1;

}

**return** 0;

}

**PROBLEM 9:**

**struct** NODE

{

**int** key;

NODE \*pNext;

};

**struct** Stack

{

NODE \*pHead;

NODE \*pTail;

};

NODE\* createNode(**int** data)

{

NODE \*p = **new** NODE;

p->key = data;

p->pNext = **NULL**;

**return** p;

}

**void** Initializer (Stack &S)

{

S.pHead = S.pTail = **NULL**;

}

**void** Push (Stack &S, **int** data)

{

NODE \*p = createNode(data);

**if** (S.pHead == **NULL**) {S.pHead = p; S.pTail = p;}

**else**

{

S.pTail->pNext = p;

S.pTail = p;

}

}

**void** Pop (Stack &S)

{

**if** (S.pHead == **NULL**) **return**;

**if** (S.pHead->pNext == **NULL**) S.pHead = **NULL**;

**else** {

NODE \*cur = S.pHead;

**while** (cur->pNext->pNext != **NULL**)

{

cur = cur->pNext;

}

cur->pNext = **NULL**;

}

}