EE2410 Data Structure Hw #3 (Chapter 4 Linked Lists of textbook)
due date 5/5/2024(Sun.) 23:59
Student No.: ____111060005_____
Name: _____胡昱煊_____

Note: Use MS Word to **edit this file** by directly typing your student number and name in above blanks and your answer to each homework problem right in the **Sol:** blanks as shown below. Then save your file as **Hw3-SNo.pdf**, where SNo is your student number. Submit the **Hw3-SNo.pdf** file via eLearn. The grading will be based on the correctness of your answers to the problems, and the **format requirement**. Fail to comply with the aforementioned format (file name, header, problem, answer, problem, answer,…), will certainly degrade your score. If you have any questions, please feel free to ask. Submit your homework before the deadline (midnight of 4/28). Fail to comply (**late** homework) will have ZERO score. **Copy** homework will have ZERO score (both parties) and SERIOUS consequences.

1. (21%)

Given a template linked list **L** instantiated by the Chain class with a pointer **first** to the first node of the list as shown in Program 4.6 (textbook). The node is a ChainNode object consisting of a template data and link field. **Formulate an algorithm** (pseudo code OK, graph + explanation, C++ code not necessary) which will

(a) count the number of nodes in L. Explain your algorithm properly (using either text or graphs).

(b) change the data field of **the kth node** (the first 1st node start at index 0) of L to the value given by Y. Explain your algorithm properly (using either text or graphs).

(c) perform an insertion to the **immediate before of the kth node** in the list L. Explain your algorithm properly (using either text or graphs).

(d) **delete every other node** of L beginning with node first (i.e., the first, 3rd, 5th,…nodes of L are deleted). Explain your algorithm properly (using either text or graphs).

(e) divides the given list into two sublists of (almost) equal sizes (**divideMid()**). Suppose myList points to the list with elements 34 65 27 89 12 (in this order). The statement: myList.divideMid(subList); divides myList into two sublists: myList points to the list with the elements 34 65 27, and subList points to the sublist with the elements 89v 12. Formulate a step-by-step algorithm to perform this task. Explain your algorithm properly (using either text or graphs).

(f) **deconcatenate** (or **split**) a linked list L into two linked list. Assume the node

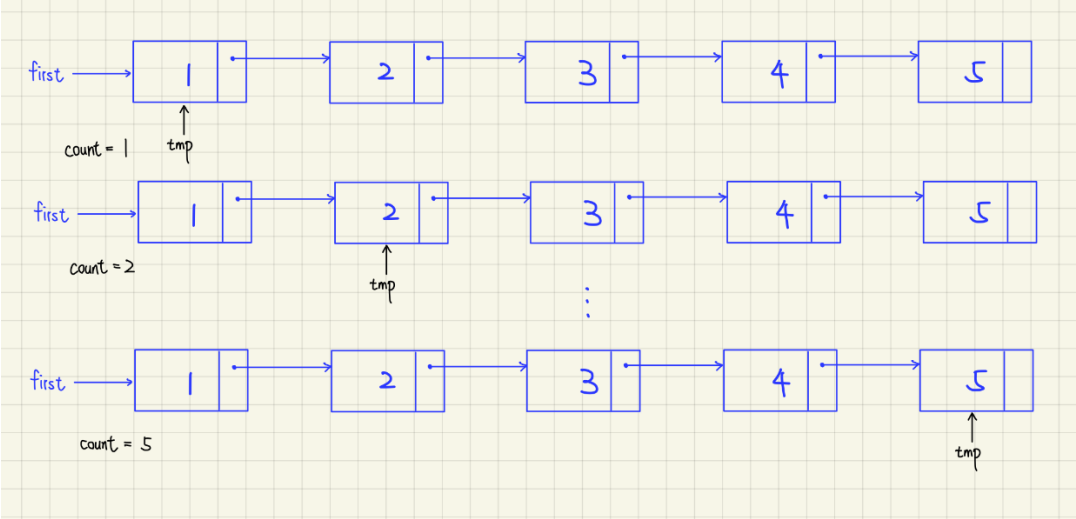
denoted by the pointer variable **split** is to be the first node in the second linked list. Formulate a step-by-step algorithm to perform this task. Explain your algorithm properly (using either text or graphs).

(g) **merge** the two chains: $L_1 = (x_1,x_2,..,x_n)$ and $L_2 = (y_1,y_2,\ldots,y_m)$ together to obtain the chain $L_3 = (x_1,y_1,x_2,y_2,\ldots,x_m,y_m,x_{m+1},..,x_n)$ if n>m and $L_3 = (x_1,y_1,x_2,y_2,\ldots,x_n,y_n,y_{n+1},..,y_m)$ if n<m. Explain your algorithm properly (using either text or graphs).

Sol:

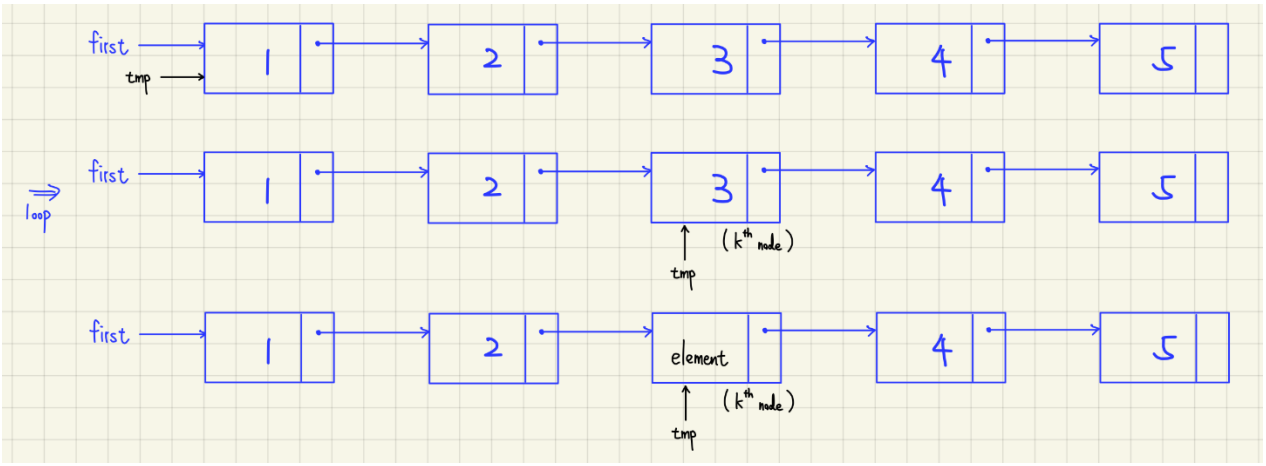
(a)

```
template <class T>
void Chain<T>::length()
{
    int count = 0;
    ChainNode<T> *tmp = first;
    while (tmp != NULL) {
        count++;
        tmp = tmp->link;
    }
    return count;
}
```



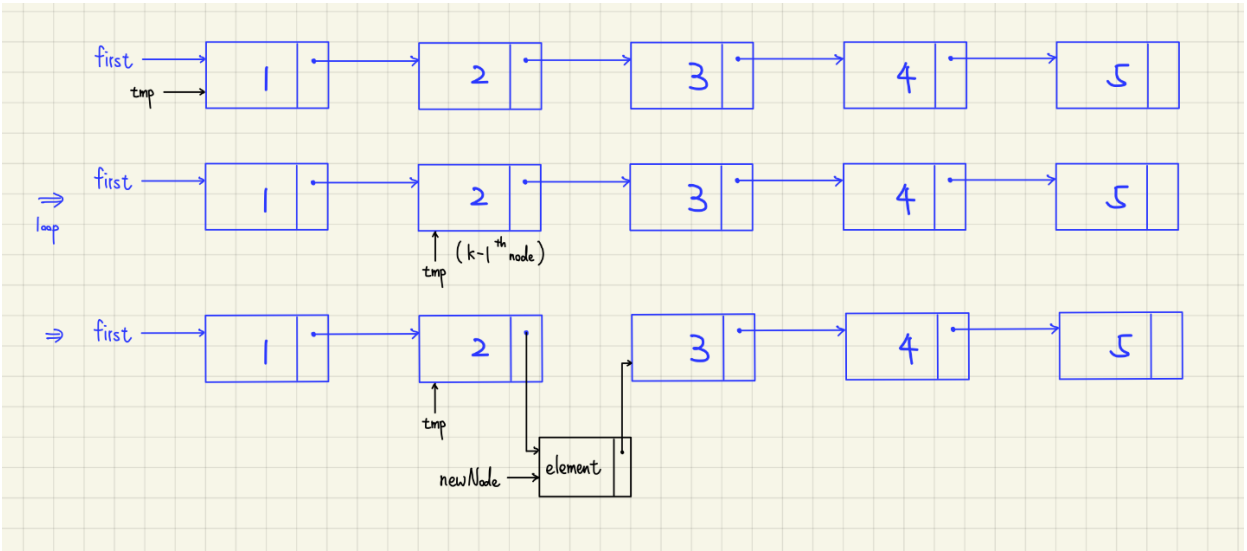
(b)

```
template <class T>
void Chain<T>::changeKthNode(int k, T element)
{
    ChainNode<T> *tmp = first;
    for (int i = 1; i < k; i++) {
        tmp = tmp->link;
    }
    tmp->data = element;
}
```



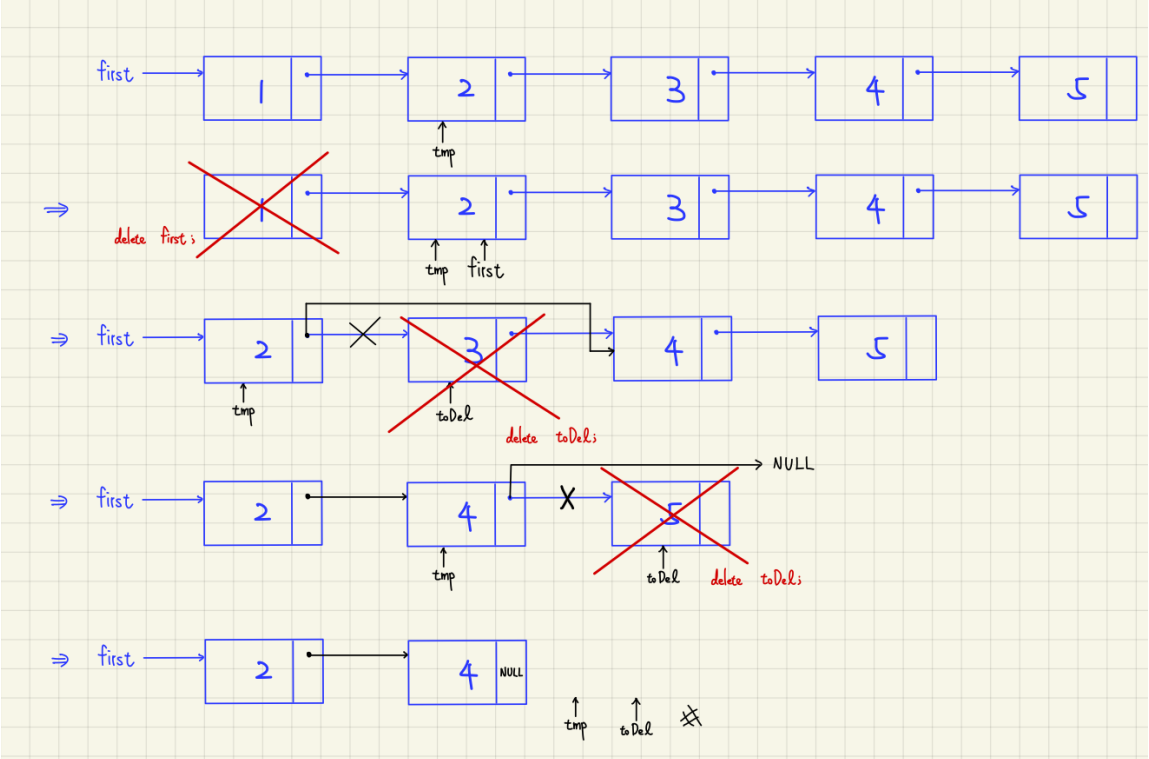
(c)

```
template <class T>
void Chain<T>::insesertBeforeKthNode(int k, T element)
{
    ChainNode<T> *newNode = new ChainNode<T>(element);
    if (k == 1) {
        newNode->link = first;
        first = newNode;
    }
    else {
        ChainNode<T> *tmp = first;
        for (int i = 1; i < k - 1; i++) {
            tmp = tmp->link;
        }
        newNode->link = tmp->link;
        tmp->link = newNode;
    }
}
```



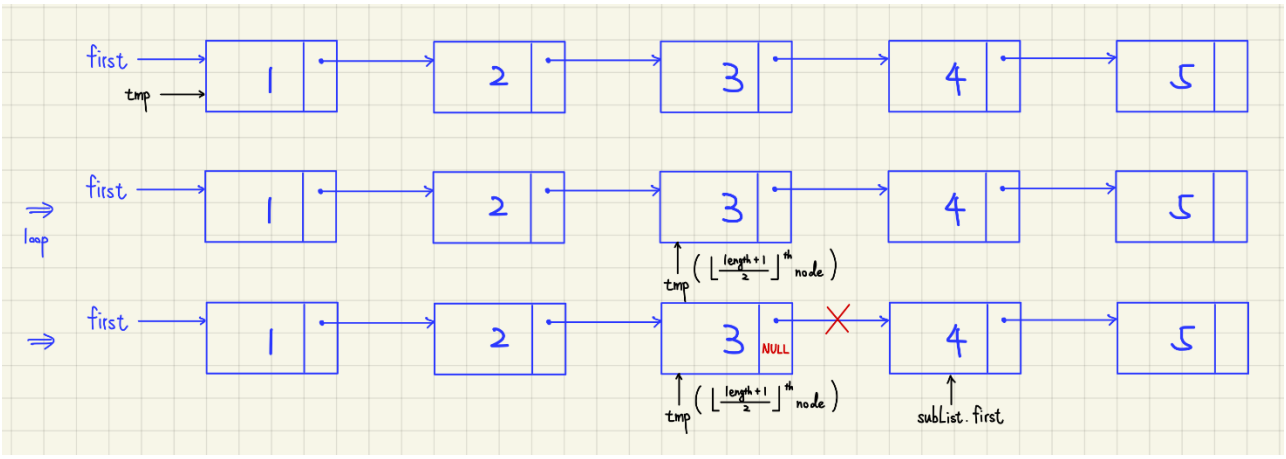
(d)

```
template <class T>
void Chain<T>::deleteOdd()
{
    ChainNode<T> *tmp = first->link;
    ChainNode<T> *toDel;
    delete first;
    first = tmp;
    while (tmp != NULL) {
        toDel = tmp->link;
        tmp->link = toDel->link;
        if (toDel == NULL) {
            break;  // if number of nodes is even
        }
        delete toDel;
        tmp = tmp->link;
    }
}
```



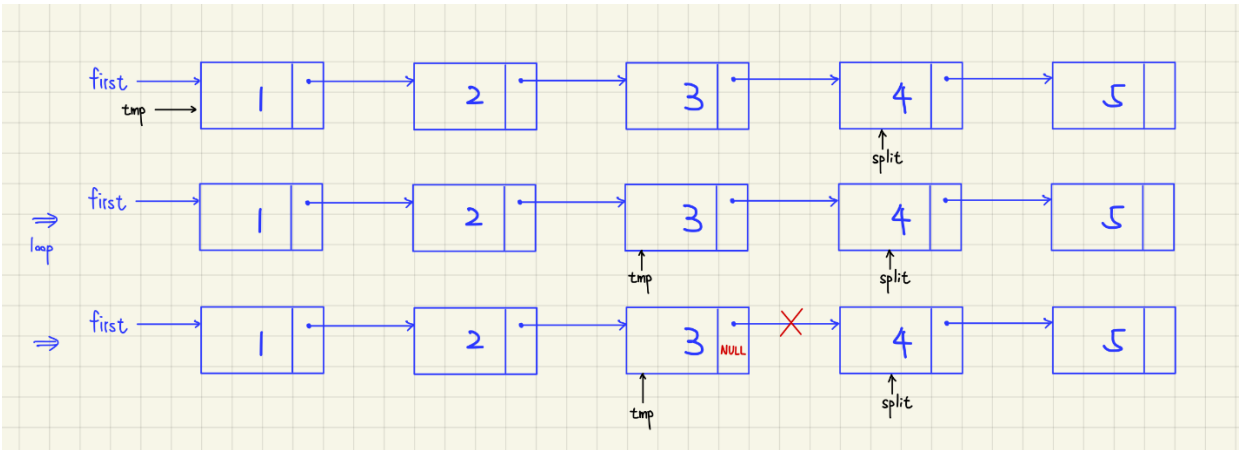
(e)

```
template <class T>
void Chain<T>::divideMid(Chain<T> &subList)
{
    ChainNode<T> *tmp = first;
    int length = length();
    for (int i = 1; i < (length + 1) / 2; i++) {
        tmp = tmp->link;
    }
    subList.first = tmp->link;
    tmp->link = NULL;
}
```



(f)

```
template <class T>
void Chain<T>::split(ChainNode<T> *split)
{
    ChainNode<T> *tmp = first;
    while (tmp->link != split) {
        tmp = tmp->link;
    }
    tmp->link = NULL;
}
```

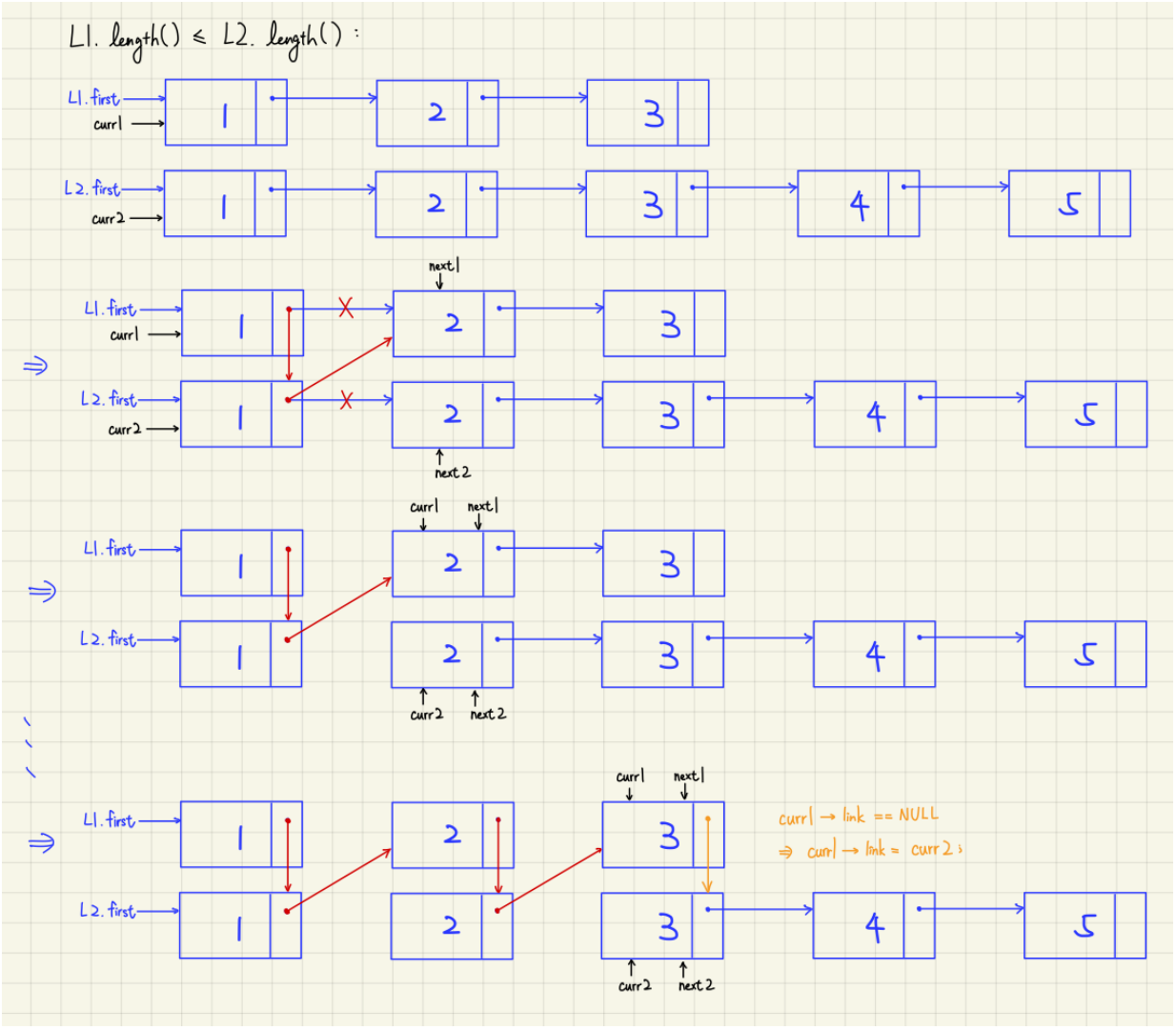


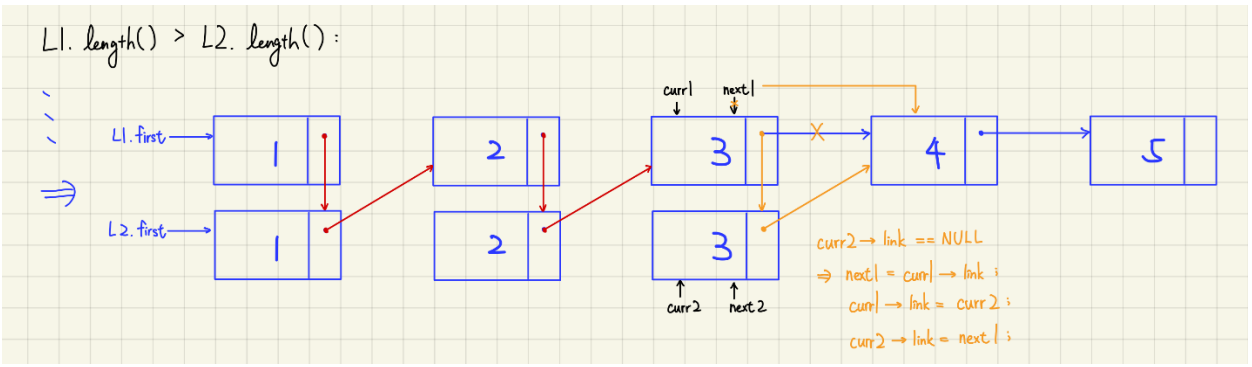
(g)

```
template <class T>
void Chain<T>::merge(Chain<T> &L2)
{
    ChainNode<T> *curr1 = first;
    ChainNode<T> *curr2 = L2.first;
    ChainNode<T> *next1, *next2;

    while (curr1->link != NULL && curr2->link != NULL) {
        next1 = curr1->link;
        next2 = curr2->link;
        curr1->link = curr2;
        curr2->link = next1;
        curr1 = next1;
        curr2 = next2;
    }

    if (curr1->link == NULL) {
        curr1->link = curr2;
    }
    else {
        next1 = curr1->link;
        curr1->link = curr2;
        curr2->link = next1;
    }
}
```





2. (18%)

Given a **circular linked list L** instantiated by class CircularList containing a private data member, **first** pointing to the first node in the circular list as shown in Figure 4.14.

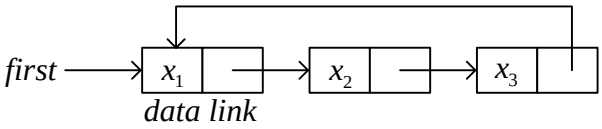


Fig. 4.14 A circular linked list

formulate algorithms (pseudo code OK, graph + explanation, C++ code not necessary) to

(a) count the number of nodes in the circular list. Explain your algorithm properly (using either text or graphs)

(b) insert a new node at the front of the list. Discuss the time complexity of your algorithm. Explain your algorithm properly (using either text or graphs)

(c) insert a new node at the back (right after the last node) of the list. Discuss the time complexity of your algorithm. Explain your algorithm properly (using either text or graphs)

(d) delete the first node of the list. Discuss the time complexity of your algorithm. Explain your algorithm properly (using either text or graphs)

(e) delete the last node of the list. Discuss the time complexity of your algorithm. Explain your algorithm properly (using either text or graphs).

(f) Repeat (a) – (e) above and (b) – (g) in Problem 1 above if the circular list is modified as shown in Figure 4.16 below by introducing a dummy node, header.

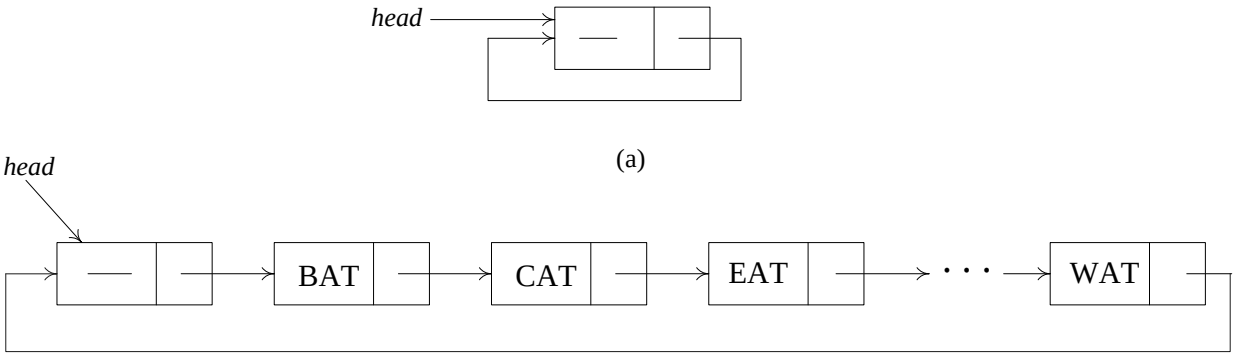
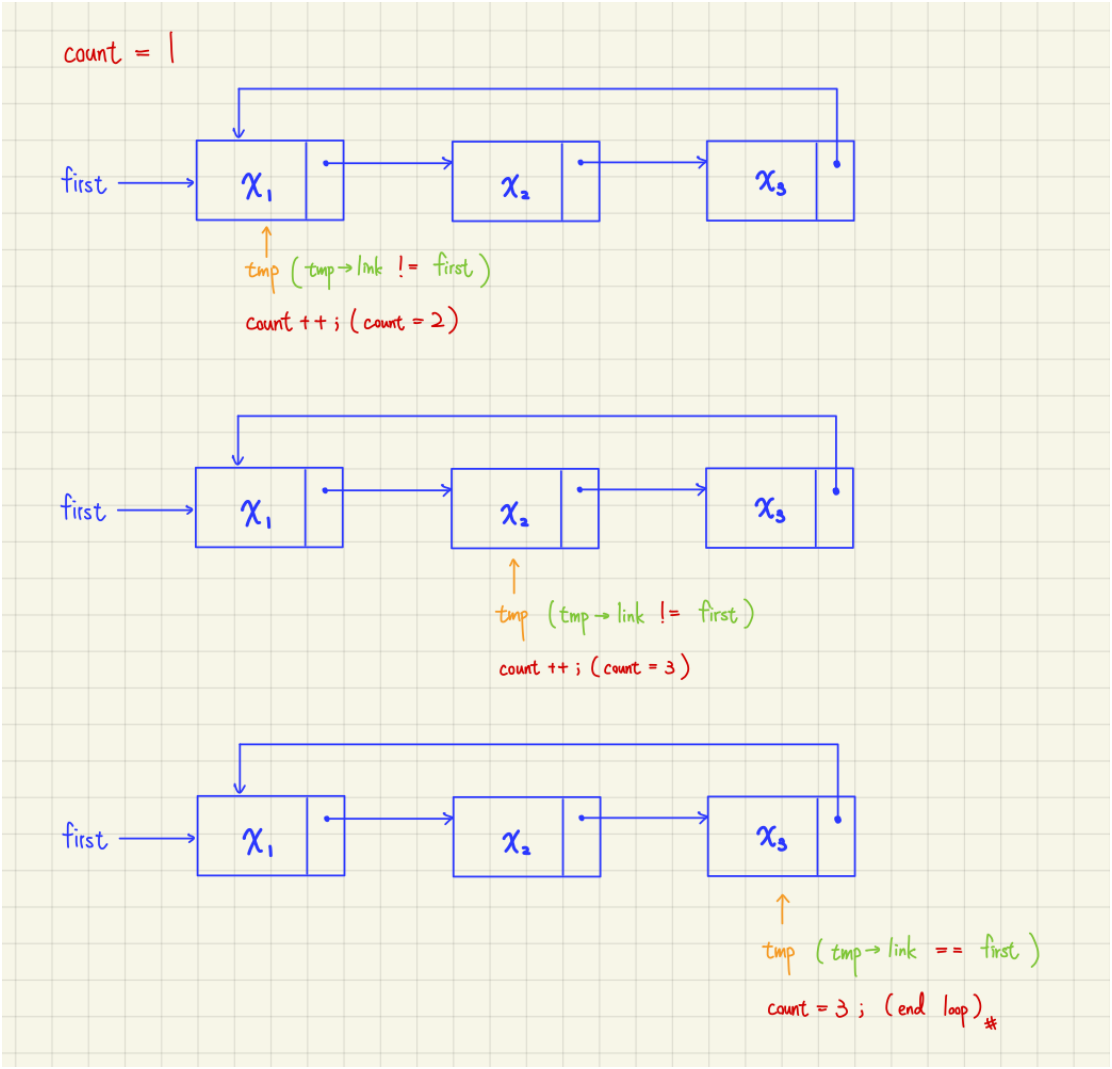


Figure 4.16 Circular list with a header node

Sol:

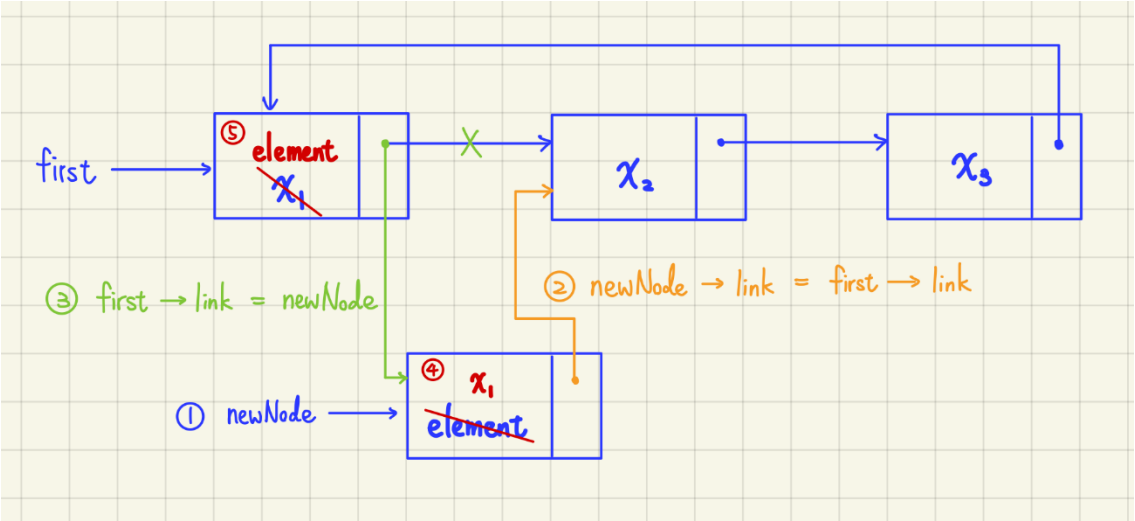
(a)

```
template <class T>
void CircularList<T>::length()
{
    int count = 1;
    CircularNode<T> *tmp = first;
    while (tmp->link != first) {
        count++;
        tmp = tmp->link;
    }
    return count;
}
```



(b)

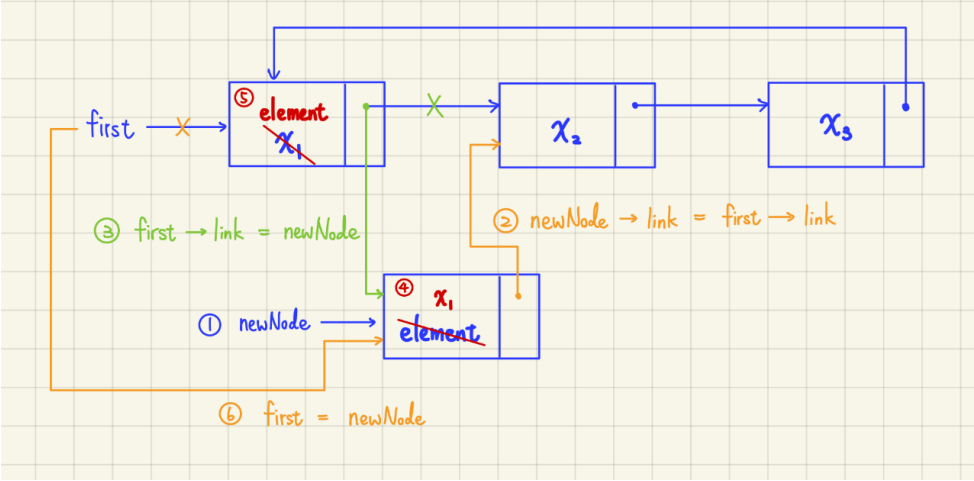
```
template <class T>
void CircularList<T>::insertFront(T element)
{
    CircularNode<T> *newNode = new CircularNode<T>(element);
    if (first == NULL) {
        first = newNode;
        first->link = first;
    }
    else {
        newNode->link = first->link;
        first->link = newNode;
        newNode->data = first->data;
        first->data = element;
    }
}
```



Time complexity: O(1)

(c)

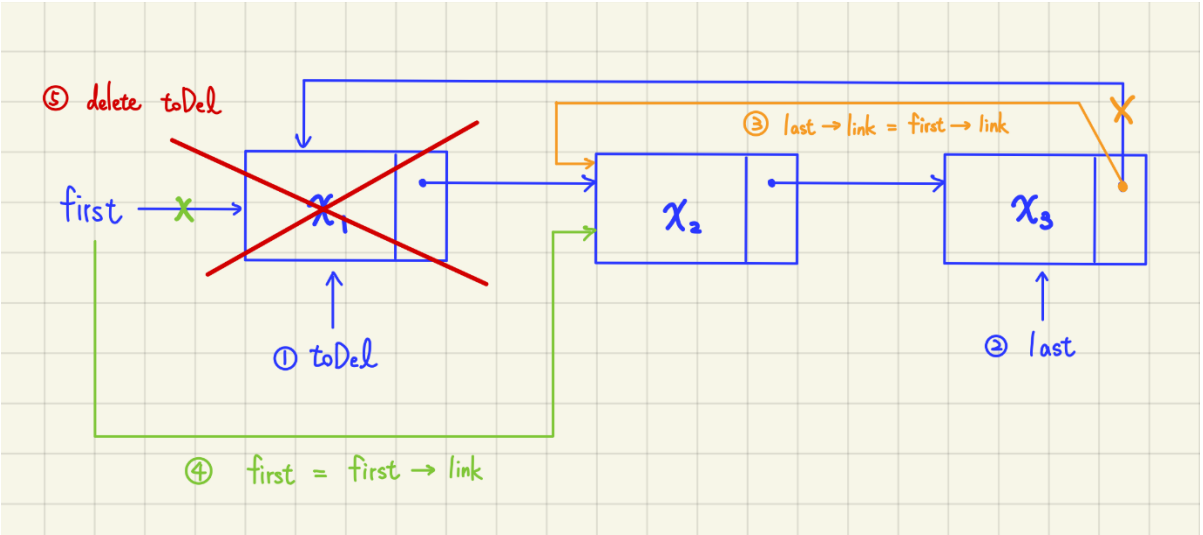
```
template <class T>
void CircularList<T>::insertBack(T element)
{
    CircularNode<T> *newNode = new CircularNode<T>(element);
    if (first == NULL) {
        first = newNode;
        first->link = first;
    }
    else {
        newNode->link = first->link;
        first->link = newNode;
        newNode->data = first->data;
        first->data = element;
        first = newNode;
    }
}
```



Time complexity: O(1)

(d)

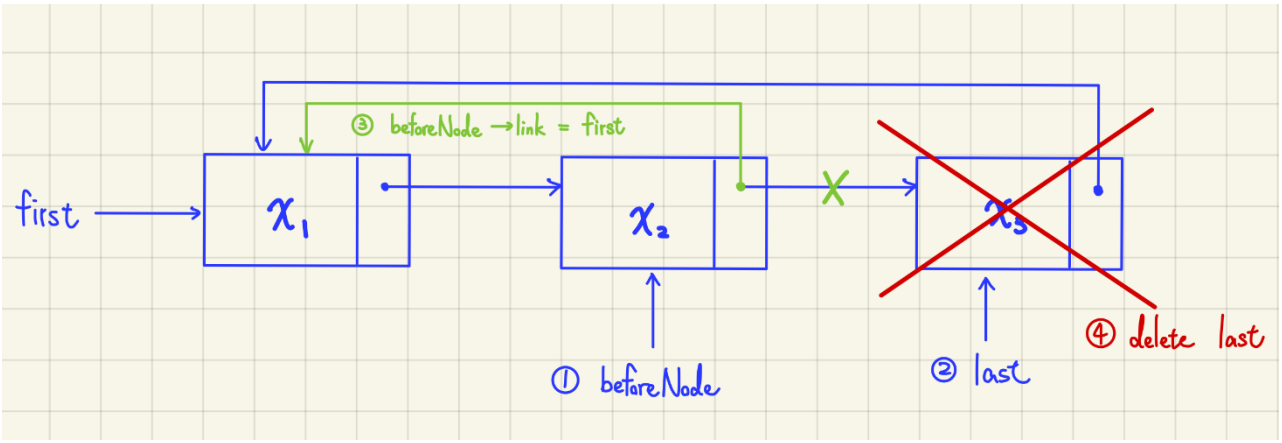
```
template <class T>
void CircularList<T>::deleteFront()
{
    if (first->link == first) {
        delete first;
        first = NULL;
    }
    else {
        CircularNode<T> *toDel = first;
        CircularNode<T> *last = first;
        while (last->link != first) {
            last = last->link;
        }
        last->link = first->link;
        first = first->link;
        delete toDel;
    }
}
```



Time complexity: O(n)

(e)

```
template <class T>
void CircularList<T>::deleteBack()
{
    if (first->link == first) {
        delete first;
        first = NULL;
    }
    else {
        CircularNode<T> *beforeNode = first;
        CircularNode<T> *last;
        while (beforeNode->link->link != first) {
            beforeNode = beforeNode->link;
        }
        last = beforeNode->link;
        beforeNode->link = first;
        delete last;
    }
}
```



Time complexity: O(n)

(f)

2a)

```
template <class T>
void CircularList<T>::length()
{
    int count = 0;
    ChainNode<T> *tmp = head;
    while (tmp->link != head) {
        count++;
        tmp = tmp->link;
    }
    return count;
}
```

2b)

```
template <class T>
void CircularList<T>::insertFront(T data)
{
    ChainNode<T> *newNode = new ChainNode(data);
    newNode->link = head->link;
    head->link = newNode;
}
```

Time complexity: O(1)

2c)

```
template <class T>
void CircularList<T>::insertBack(T data)
{
    ChainNode<T> *newNode = new ChainNode(data);
    ChainNode<T> *tmp = head;
    while (tmp->link != head) {
        tmp = tmp->link;
    }
    tmp->link = newNode;
    newNode->link = head;
}
```

Time complexity: O(n)

2d)

```
template <class T>
void CircularList<T>::deleteFront()
{
    ChainNode<T> *toDel = head->link;
    head->link = toDel->link;
    delete toDel;
}
```

Time complexity: O(1)

2e)

```
template <class T>
void CircularList<T>::deleteBack()
{
    ChainNode<T> *tmp = head;
    ChainNode<T> *toDel;
    while (tmp->link->link != head) {
        tmp = tmp->link;
    }
    toDel = tmp->link;
    tmp->link = head;
    delete toDel;
}
```

Time complexity: O(n)

1b)

```
template <class T>
void CircularList<T>::changeKthNode(int k, T data)
{
    ChainNode<T> *tmp = head;
    for (int i = 0; i < k; i++) {
        tmp = tmp->link;
    }
    tmp->data = data;
}
```

1c)

```
template <class T>
void CircularList<T>::insertBeforeKthNode(int k, T data)
{
    ChainNode<T> *newNode = new ChainNode(data);
    ChainNode<T> *tmp = head;
    for (int i = 0; i < k - 1; i++) {
        tmp = tmp->link;
    }
    newNode->link = tmp->link;
    tmp->link = newNode;
}
```

1d)

```
template <class T>
void CircularList<T>::deleteOdd()
{
    ChainNode<T> *tmp = head->link->link;
    ChainNode<T> *toDel = head->link;
    delete toDel;
    head->link = tmp;
    while (tmp != head) {
        toDel = tmp->link;
        if (toDel == head) {
            break; // if number of nodes is even
        }
        tmp->link = toDel->link;
        delete toDel;
        tmp = tmp->link;
    }
}
```

1e)

```
// subList is an empty list
template <class T>
void CircularList<T>::divideMid(CircularList<T> &subList)
{
    ChainNode<T> *tmp = head;
    ChainNode<T> *end = head;
    int length = length();
    for (int i = 0; i < (length + 1) / 2; i++) {
        tmp = tmp->link;
    }
    while (end->link != head) {
        end = end->link;
    }
    subList.head->link = tmp;
    end->link = subList.head;    // wrap around subList
    tmp->link = head;            // wrap around the original list
}
```

1f)

```
template <class T>
void CircularList<T>::split(CircularList<T> *split)
{
    ChainNode<T> *tmp = head;
    while (tmp->link != split){
        tmp = tmp->link;
    }
    tmp->link = head;
}
```

1g)

```
// merged list is stored in L1 with L1.merge(L2)
template <class T>
void CircularList<T>::merge(CircularList<T> &L2)
{
    ChainNode<T> *curr1 = head->link;
    ChainNode<T> *curr2 = L2.head->link;
    ChainNode<T> *next1, *next2;

    while (curr1->link != head && curr2->link != L2.head) {
        next1 = curr1->link;
        next2 = curr2->link;
        curr1->link = curr2;
        curr2->link = next1;
        curr1 = next1;
        curr2 = next2;
    }

    // if L1 is shorter
    if (curr1->link == head) {
        curr1->link = curr2;
        while (curr2->link != L2.head) {
            curr2 = curr2->link;
        }
        curr2->link = head;     // wrap around
    }
    // if L2 is shorter
    else {
        next1 = curr1->link;
        curr1->link = curr2;
        curr2->link = next1;
        while (next1->link != head) {
            next1 = next1->link;
        }
        next1->link = head;    // wrap around
    }
}
```

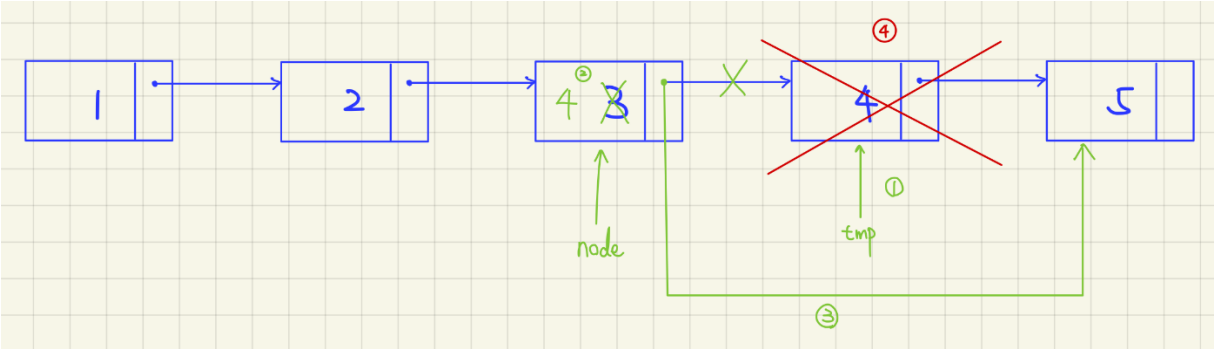
3. (10%)

Suppose we have a pointer to a node in a **singly linked list** that is guaranteed not to be the last node in the list. We do not have pointers to any other nodes (except by following links). Describe an **O(1)** algorithm that logically removes the value stored in such a node from the linked list, maintaining the integrity of the linked list.

Sol:

```
1   void removeNode(ListNode *node)
2   {
3       ListNode *tmp = node->next;
4       node->val = tmp->val;
5       node->next = tmp->next;
6       delete tmp;
7   }
```

explanation in graph:



4. (5%)

Let x be a node in a **singly linked circular list**. Write a C++ function to delete the data in this node. Following the deletion, the number of nodes in the list is one less than before deletion. **Your function must run in O(1) time.**

Sol:

```
9   void deleteNode(CircularListNode x)
10  {
11      CircularListNode *tmp = x.next;
12      x.val = tmp->val;
13      x.next = tmp->next;
14      delete tmp;
15  }
```

5. (5%)

One way to implement a queue is to use a circular linked list. In a circular linked list, the last node's next pointer points at the first node. Assume the list **does not contain a header** and that we can maintain, at most, **one iterator corresponding to a node** in the list. For which of the following representations can all basic queue operations be performed in constant worst-case time? Justify your answers.

a. Maintain an iterator that corresponds to the first item in the list.

b. Maintain an iterator that corresponds to the last item in the list.

Sol:

b. Maintain an iterator that corresponds to the last item in the list.
Since the link field of the last node points to the first node, which means that we can access the first node by last->link.

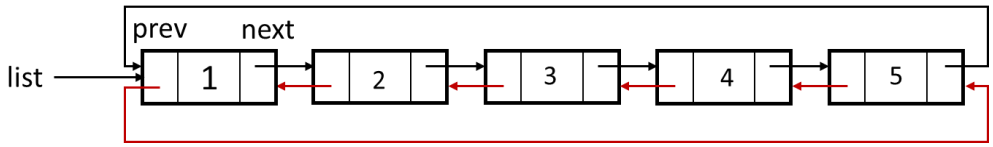
6. (10%)
Suppose that a singly linked list is implemented with both a header and a tail node. Describe constant-time algorithms to
a. insert item x before position p (given by an iterator)
b. remove the item stored at position p (given by an iterator)
Sol:
(a)

```
template <class T>
void Chain<T>::insert(T x, ChainNode<T> *p)
{
    ChainIterator<T> iterator = ChainIterator<T>(p);
    ChainNode<T> *newNode = new ChainNode<T>(x);
    newNode->link = iterator.current->link;
    --iterator; // move iterator to the previous node
    iterator.current->link = newNode;
}
```

(b)

```
template <class T>
void Chain<T>::remove(ChainNode<T> *p)
{
    ChainIterator<T> iterator = ChainIterator<T>(p);
    ChainNode<T> *toDel;
    --iterator; // move iterator to the previous node
    toDel = iterator.current->link;
    iterator.current->link = toDel->link;
    delete toDel;
}
```

7. (5%)
Assume that a circular doubly linked list has been created, as shown below. After each of the following assignments, indicate changes made in the list by showing which links have been modified. The second assignment should make changes in the list modified by the first assignment, and so on.



1)list->next->next->next = list->prev;

2)list->prev->prev->prev = list->next->next->next->prev;

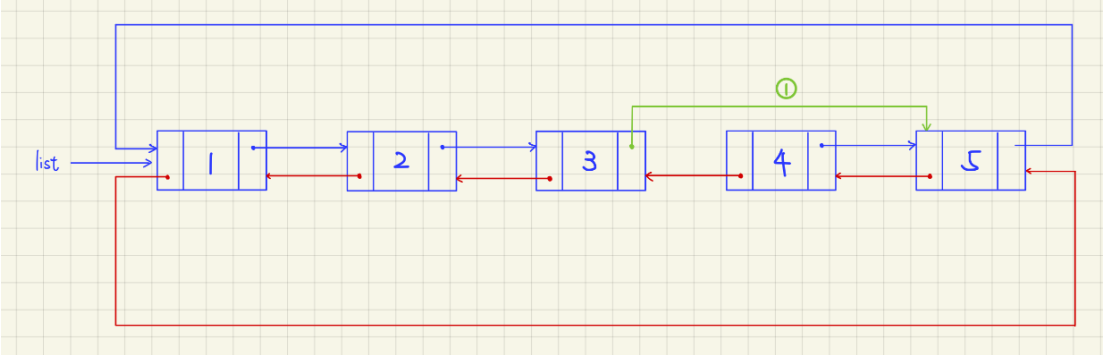
3)list->next->next->next->prev = list->prev->prev->prev;

4)list->next = list->next->next;

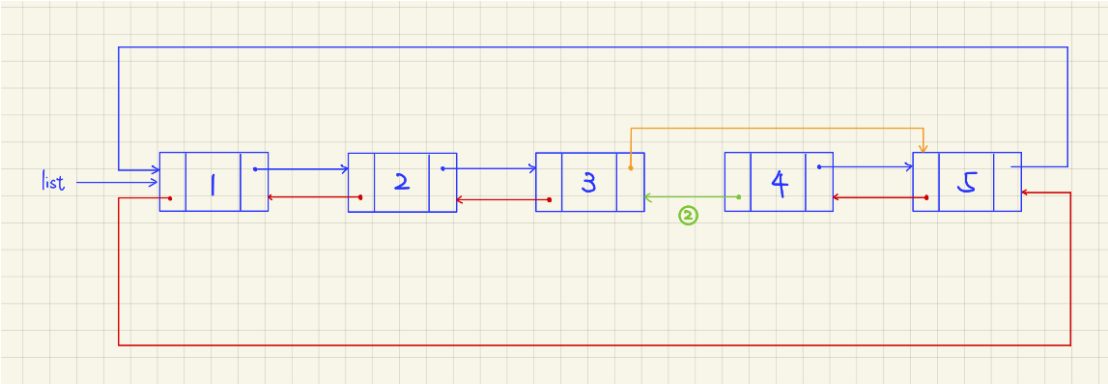
5)list->next->prev->next = list->next->next->next;

Sol:

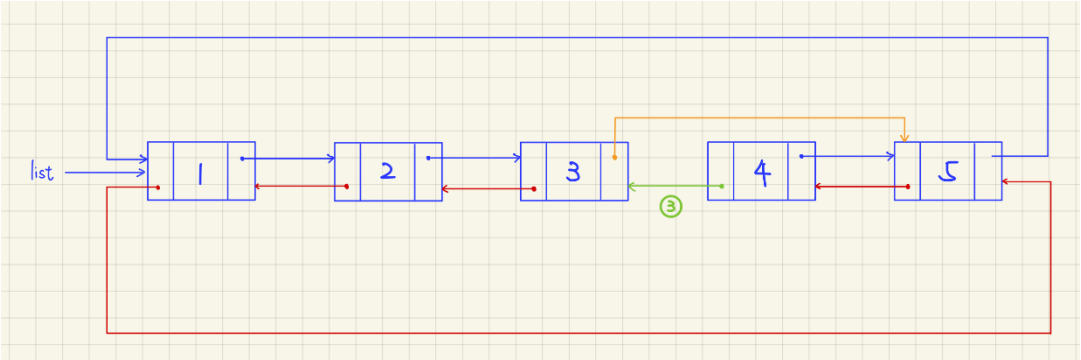
(1)



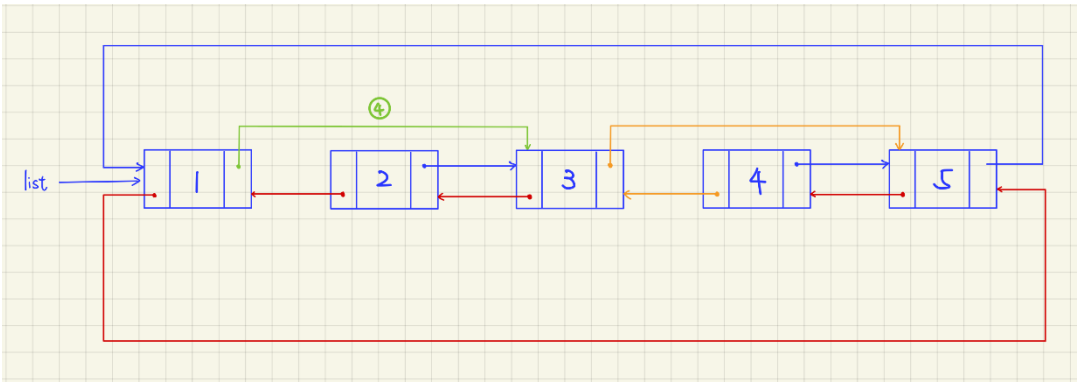
(2)



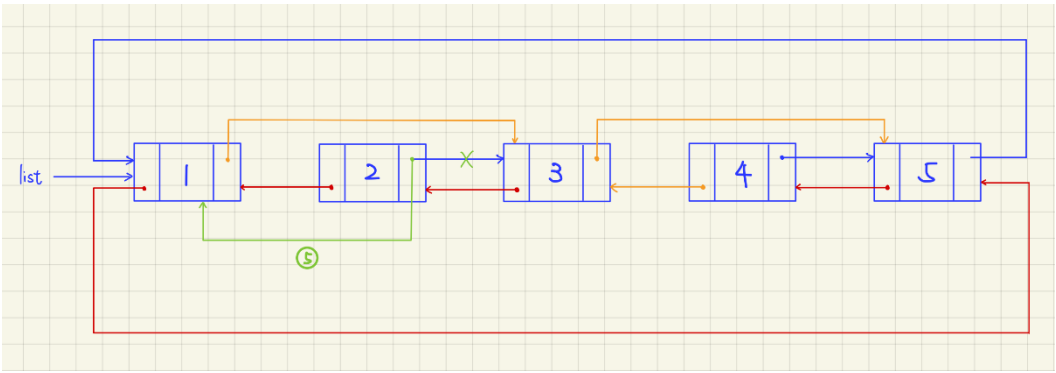
(3)



(4)



(5)



8. (6%)

Show what is produced by the following C++ code. Assume the node is in the usual data-link form with the data of type int. (list and ptr are pointers of type nodeType.)

a.
```
list = new nodeType;
list->data = 10;
ptr = new nodeType;
ptr->data = 13;
ptr->link = NULL;
list->link = ptr;
ptr = new nodeType;
ptr->data = 18;
ptr->link = list->link;
list->link = ptr;
cout << list->data << " " << ptr->data << " ";
ptr = ptr->link;
cout << ptr->data << endl;
```

b.
```
list = new nodeType;
list->data = 20;
ptr = new nodeType;
ptr->data = 28;
ptr->link = NULL;
list->link = ptr;
ptr = new nodeType;
ptr->data = 30;
ptr->link = list;
list = ptr;
ptr = new nodeType;
ptr->data = 42;
ptr->link = list->link;
list->link = ptr;
ptr = List;
while (ptr != NULL)
{
    cout << ptr->data << endl;
    ptr = ptr->link;
}
```

Sol:

(a)

10 18 13

(b)

30

42

20

28

9. (10%)

What is the output of the following program segment?

```
list<int> intList;
ostream_iterator<int> screen(cout, " ");
list<int>::iterator listIt;
```

```
intList.push_back(5);
intList.push_front(23);
intList.push_front(45);
intList.pop_back();
intList.push_back(35);
intList.push_front(0);
intList.push_back(50);
intList.push_front(34);
copy(intList.begin(), intList.end(), screen);
cout << endl;
listIt = intList.begin();
intList.insert(listIt,76);
++listIt;
++listIt;
intList.insert(listIt,38);
intList.pop_back();
++listIt;
++listIt;
intList.erase(listIt);
intList.push_front(2 * intList.back());
intList.push_back(3 * intList.front());
copy(intList.begin(), intList.end(), screen);
cout << endl;
```

Sol:

```
// if insert(listIt,76) means insert 76 before the node listIt points to
// and listIt will point to 76 after insertion
34 0 45 23 35 50
70 76 34 38 0 45 23 35 210

// if insert(listIt,76) means insert 76 after the node listIt points to
// and listIt will point to 76 after insertion
34 0 45 23 35 50
70 34 76 0 45 38 23 35 210
```

10. (10%)

Suppose the input is:

18 30 4 32 45 36 78 19 48 75 -999

What is the output of the following C++ code? (The class unorderedLinkedList is as is name.)

```
unorderedLinkedList<int> list;
unorderedLinkedList<int> copyList;
int num;
cin >> num;
while (num != -999)
{
    if (num % 5 == 0 || num % 5 == 3)
       list.insertFirst(num);
    else
       list.insertLast(num);
    cin >> num;
}
list.print(); //print out the list node data one by one.
cout << endl;
copyList = list;
copyList.deleteNode(78);
copyList.deleteNode(35);
cout << "Copy List = ";
copyList.print();
cout << endl;
```

Sol:

75 48 78 45 30 18 4 32 36 19

Copy List =75 48 45 30 18 4 32 36 19