EECS2040 Data Structure Hw #3 (Chapter 4 Linked List)
due date 4/18/2022
by 108011235 陳昭維

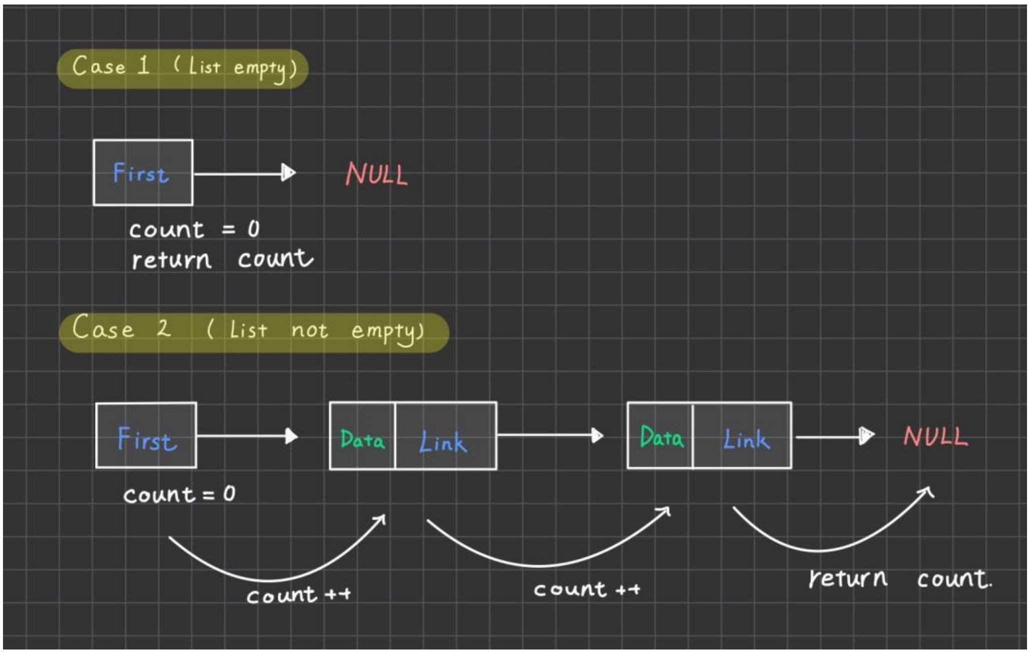
Part 1 (2% of final Grade, due 4/18/2022)

1. (30%) 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.

```
template < class T > class Chain;   // 前向宣告

template < class T >
class ChainNode {
friend class Chain <T>;
private:
    T data;
    ChainNode<T>* link;
};

template <class T>
class Chain {
public:
    Chain( ) {first = 0;} //  建構子將 first 初始化成 0
    //  鏈的處理運算
    .
    .
private:
    ChainNode<T> * first;
}
```

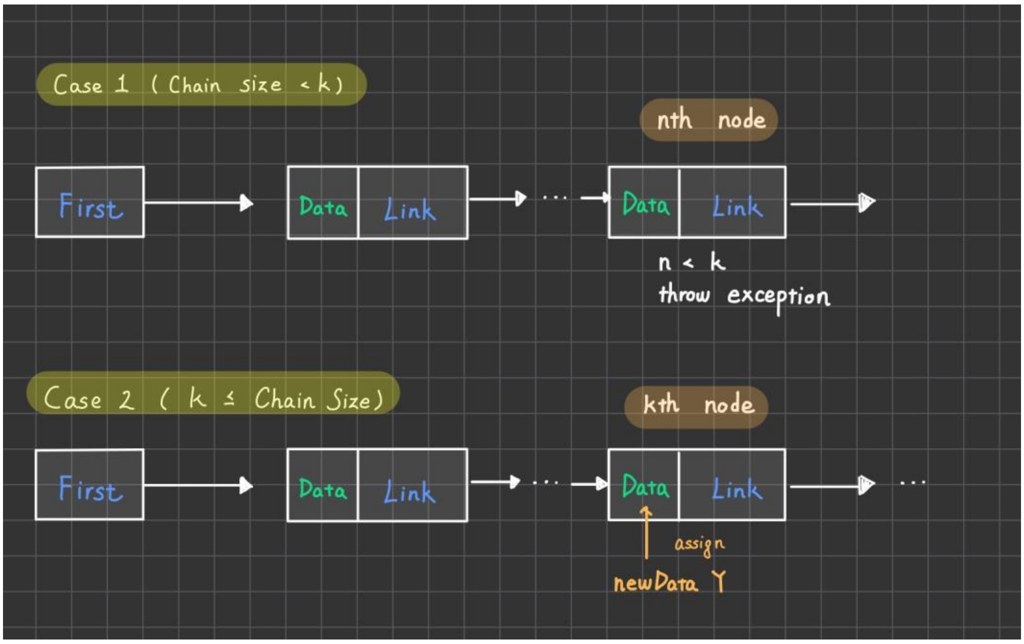
Program 4.6

(a) **Formulate an algorithm** (pseudo code OK, C++ code not necessary) which will count the number of nodes in L. Explain your algorithm properly (using either text or graphs).



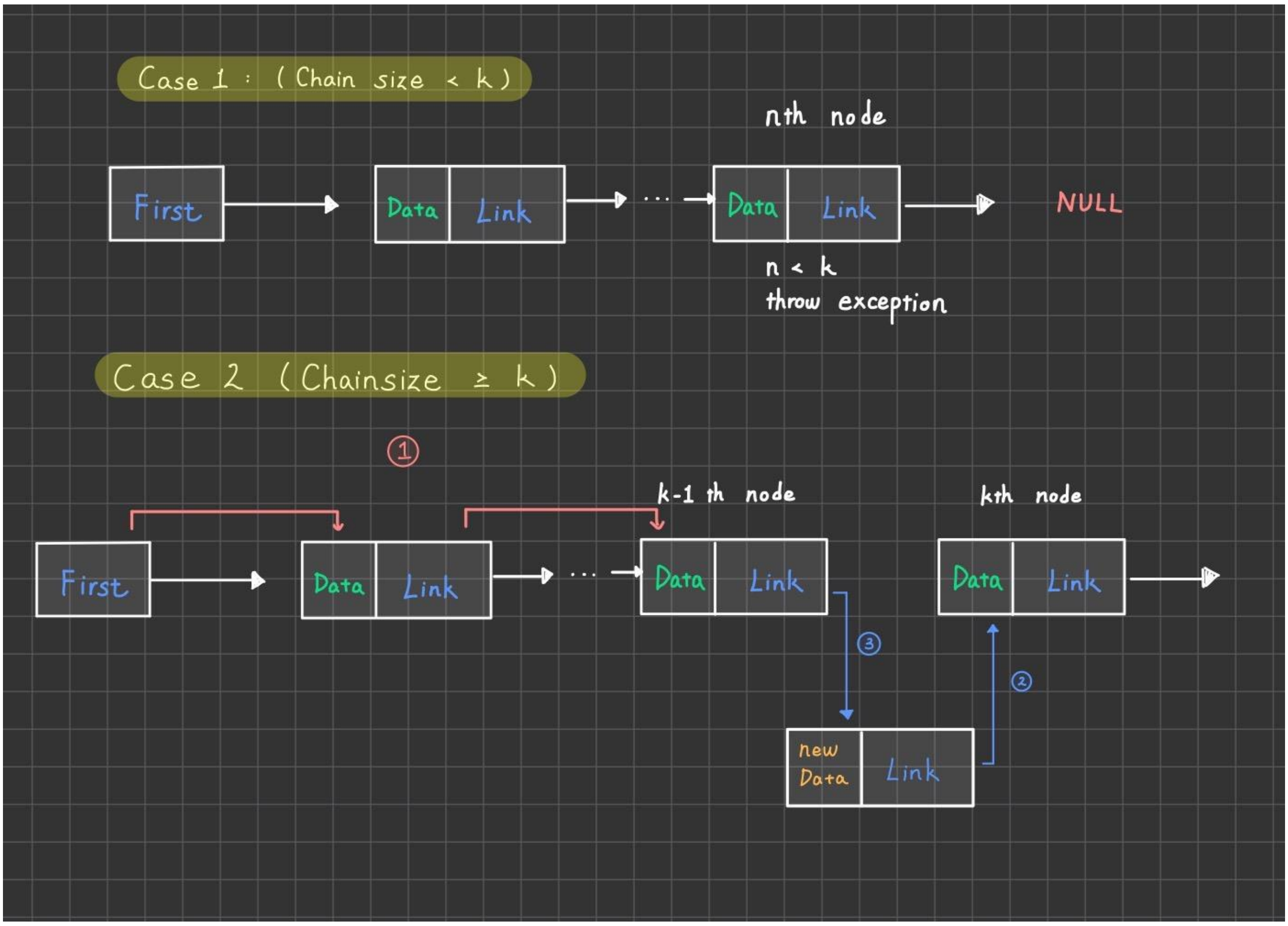

```cpp
template<class T>
int Chain<T>::Size() {
    int count = 0;
    ChainNode<T>* cur = this->first;
    if(cur == NULL)                    // if the Chain has no element
        return 0;
    else {
        while (cur != NULL) {          // while the current node is not NULL
            count ++;                  // count increment
            cur = cur->link;
        }
    }

    return count;
}
```

(b) **Formulate an algorithm** that will 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).



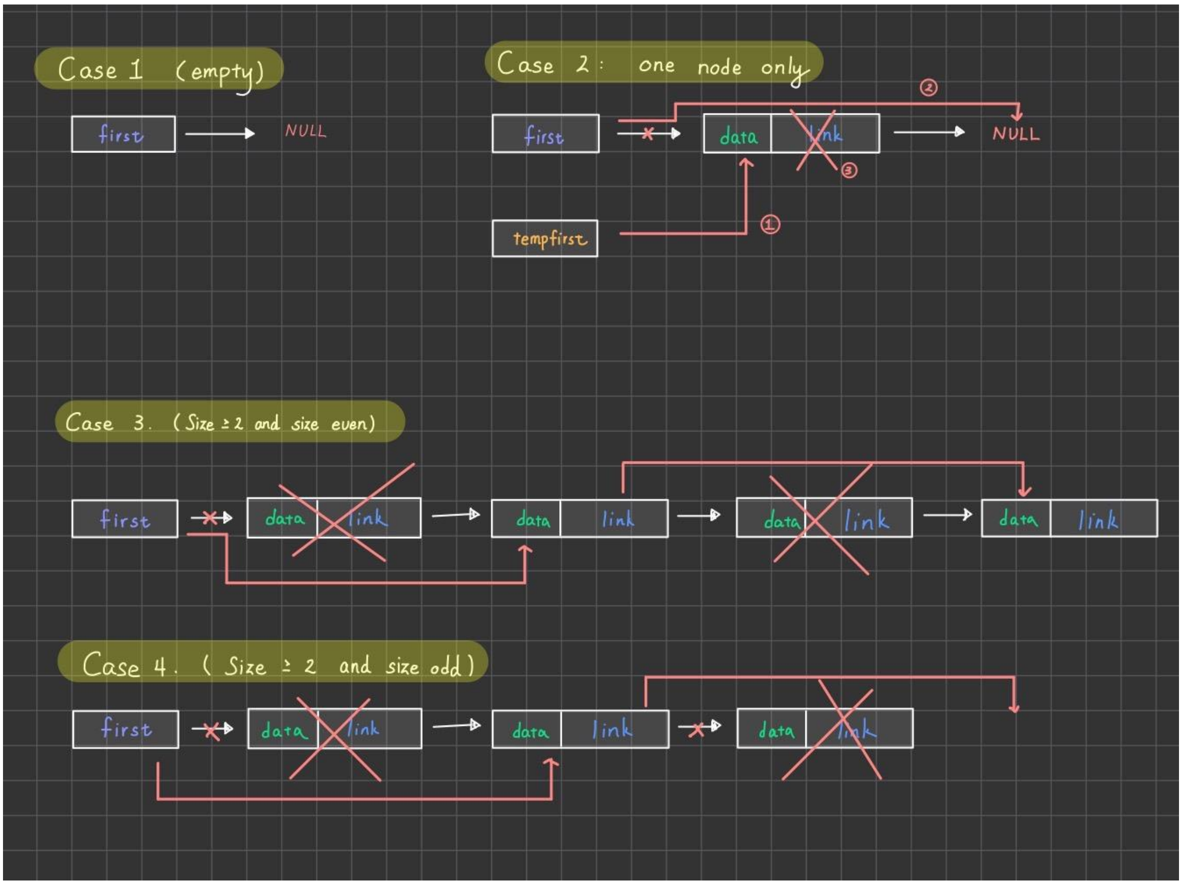

```
template<class T>
void Chain<T>::ReplaceNode(int k, T& newData) {
    ChainNode<T>* cur = this->first;
    for(int i = 1; i < k; i++) {
        if(cur == NULL)          // indicating that the Chain size is smaller than k
            cerr << "Given index is unattainable" << endl;
        else
            cur = cur->link;
    }
    cur->data = newData;       // update new data
}
```

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



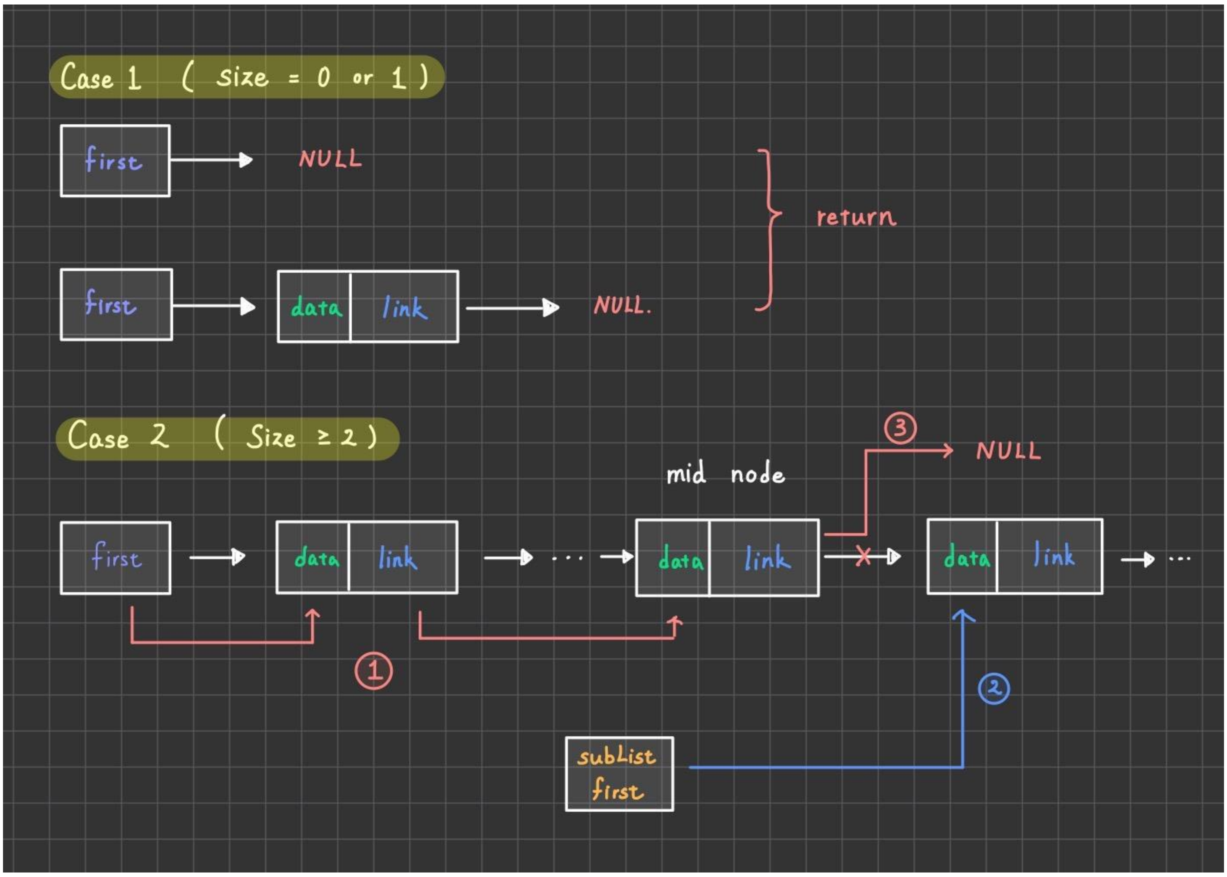

```
template <class T>
void Chain<T>::insert(int k, T& newData) {
    ChainNode<T>* cur = this->first;
    ChainNode<T>* newNode = new ChainNode<T>;
    newNode->data = newData;                    // initializing a new node
    newNode->link = NULL;

    for(int i = 1; i < k - 1; i++) {
        if(cur == NULL)                         // Chain size smaller than k
            cerr << "Given index is unattainable";
        else
            cur = cur->link;
    }

    newNode->link = cur->link;                  // New node's next node equals to kth node
    cur->link = newNode;                        // k-1 th node's next node is new node.
}
```

(d) **Formulate an algorithm** that will **delete every other odd 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).



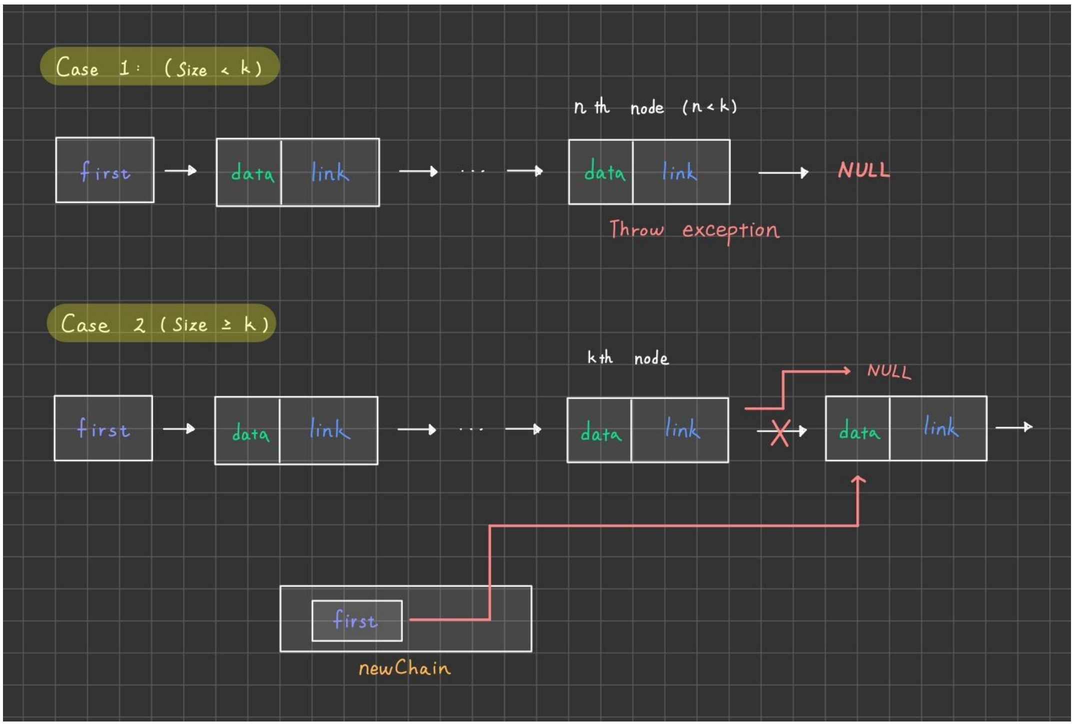

```
template <class T>
void Chain<T>::deleteAllOddNode() {

    if(first) {                      // if the link is not empty
        ChainNode<T>* tempFirst = first;
        first = first->link;
        delete tempFirst;            // delete the #1 node of the chain

        if(first) {                  // if there is more than 1 node in the chain
            ChainNode<T>* evenNode = first;
            ChainNode<T>* oddNode = first->link;

            while (evenNode && oddNode)  // while the #even and #odd node exists
            {
                evenNode->link = oddNode->link;
                delete oddNode;

                evenNode = evenNode->link;

                if(evenNode)    // if #even node is not the end of the list
                    oddNode = evenNode->link;
            }
        }
    }
}
```

(e) Formulate an algorithm divideMid that will divides the given list into two sublists of (almost) equal sizes. 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 89 12. Formulate a step-by-step algorithm to perform this task. Explain your algorithm properly (using either text or graphs).



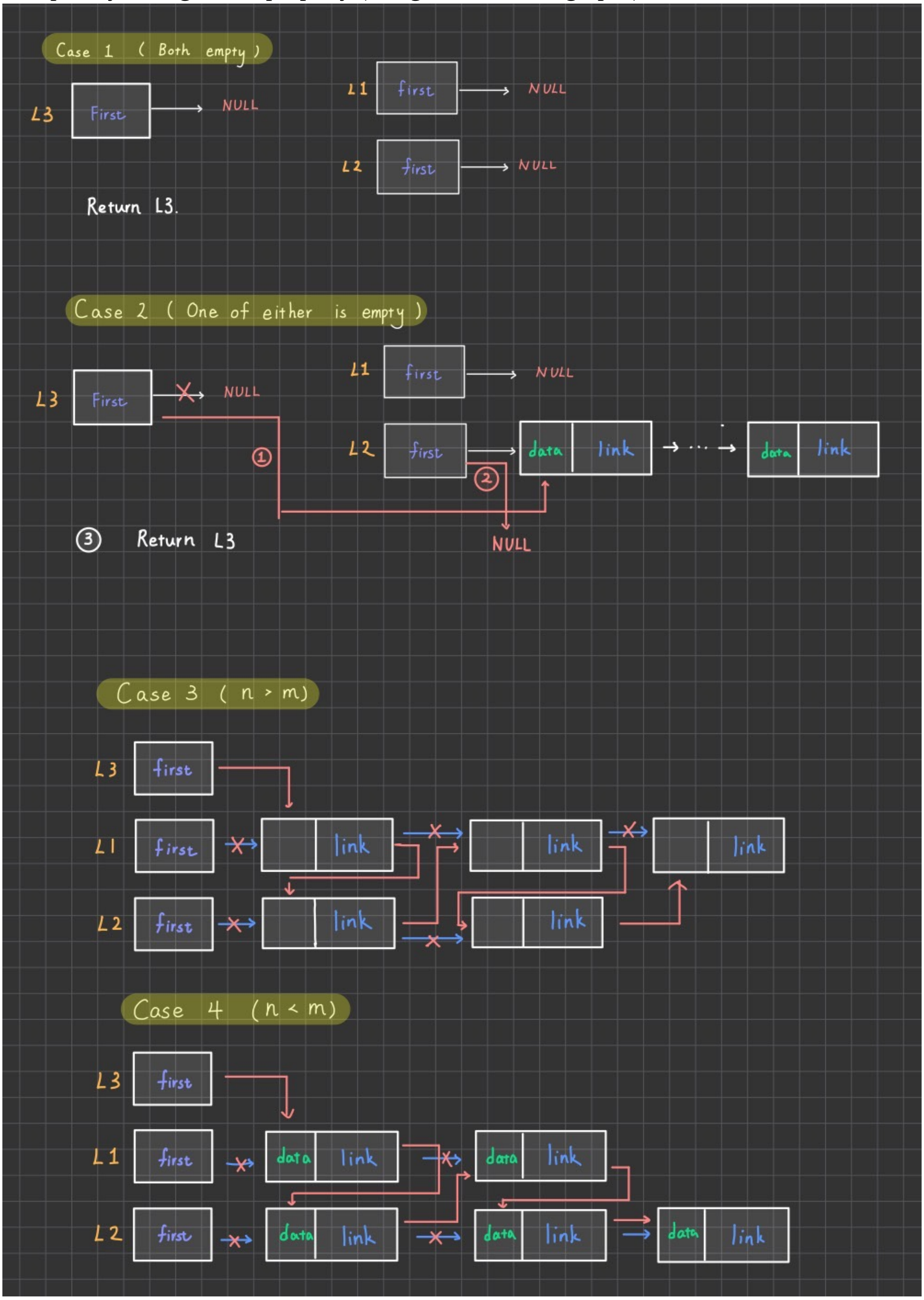

```
template <class T>
void Chain<T>::divideMid(Chain<T>* subList) {
    int count = this->Size();
    int mid = (count/2 + count%2);  // get mid num from Size

    ChainNode<T>* cur = this->first;

    if(count == 0 || count == 1)    // if size is 0 or 1, then nothing to divide
        return;

    for (int i = 1; i < mid; i++) { // reaches mid node
        cur = cur->link;
    }

    subList->first = cur->link;     // the sublist starts from the next node of mid node
    cur->link = NULL;

}
```

(f) **Formulate an algorithm** that will **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).



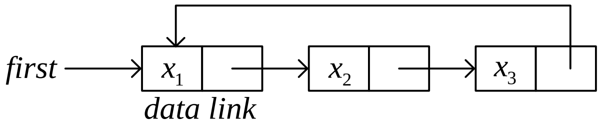

```
template <class T>
Chain<T>* Chain<T>::deconcatenate(int k) {
    ChainNode<T>* cur = this->first;
    Chain<T>* newChain = new Chain<T>;
    int count = 0;
    while(cur->link != NULL) {
        if(count == k - 1) {
            newChain->first = cur->link;
            cur->link = NULL;
            return newChain;
        }

        else {
            cur = cur->link;
            count ++;
        }
    }

    cerr << "Given index is unattainable." ;
    return NULL;
}
```

(g) Assume L_1 and L_2 are two chains: $L_1 = (x_1,x_2,..,x_n)$ and $L_2 = (y_1,y_2,\ldots,y_m)$, respectively. **Formulate an algorithm** that can **merge** the two chains 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).



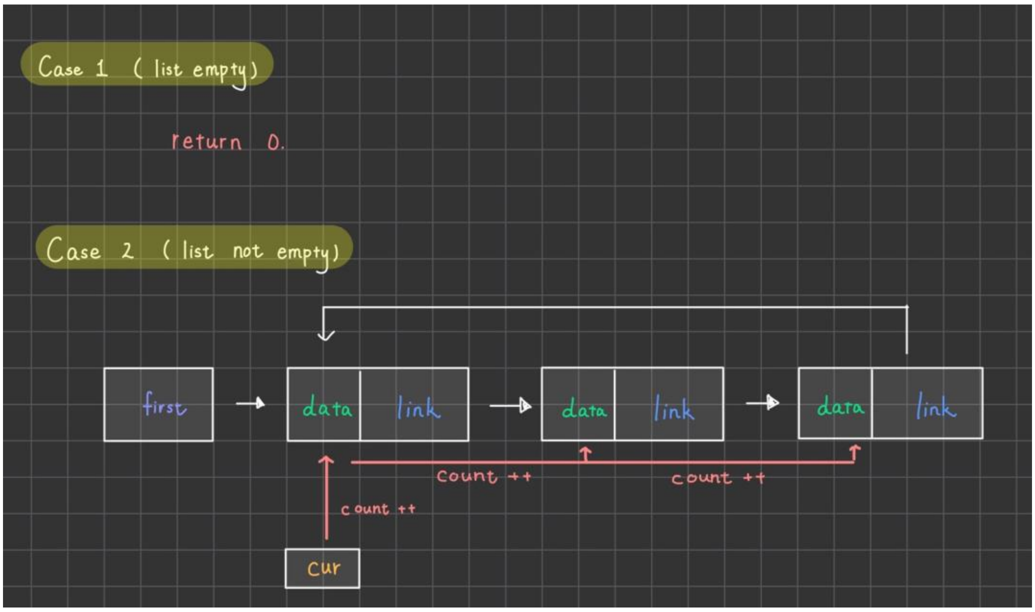
```cpp
template <class T>
Chain<T>* Chain<T>::merge(Chain<T>* L2) {
    Chain<T>* L3 = new Chain<T>;

    if(!first && !L2->first)    // both chain is empty return empty L3
        return L3;

    if(!first) {                // case of L1 empty
        L3->first = L2->first;
        L2->first = NULL;
        return L3;
    }

    if(!L2->first) {            // case of L2 empty
        L3->first = this->first;
        this->first = NULL;
        return L3;
    }

    ChainNode<T>* N1 = this->first->link;
    ChainNode<T>* N2 = L2->first;
    ChainNode<T>* cur = NULL;
    L3->first = this->first;
    cur = L3->first;

    while(N1 != NULL && N2 != NULL) {  //traverses until reaches one end of the chain
        cur->link = N2;
        cur = cur->link;
        N2 = N2->link;

        cur->link = N1;
        cur = cur->link;
        N1 = N1->link;
    }

    if(N1)
        cur->link = N1;   // append the rest of the list to the merged list
    if(N2)
        cur->link = N2;   // append the rest of the list to the merged list

    this->first = NULL;
    L2->first = NULL;
    return L3;      You, yesterday • update 0409 …
}

#endif
```

2. (55%) 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.



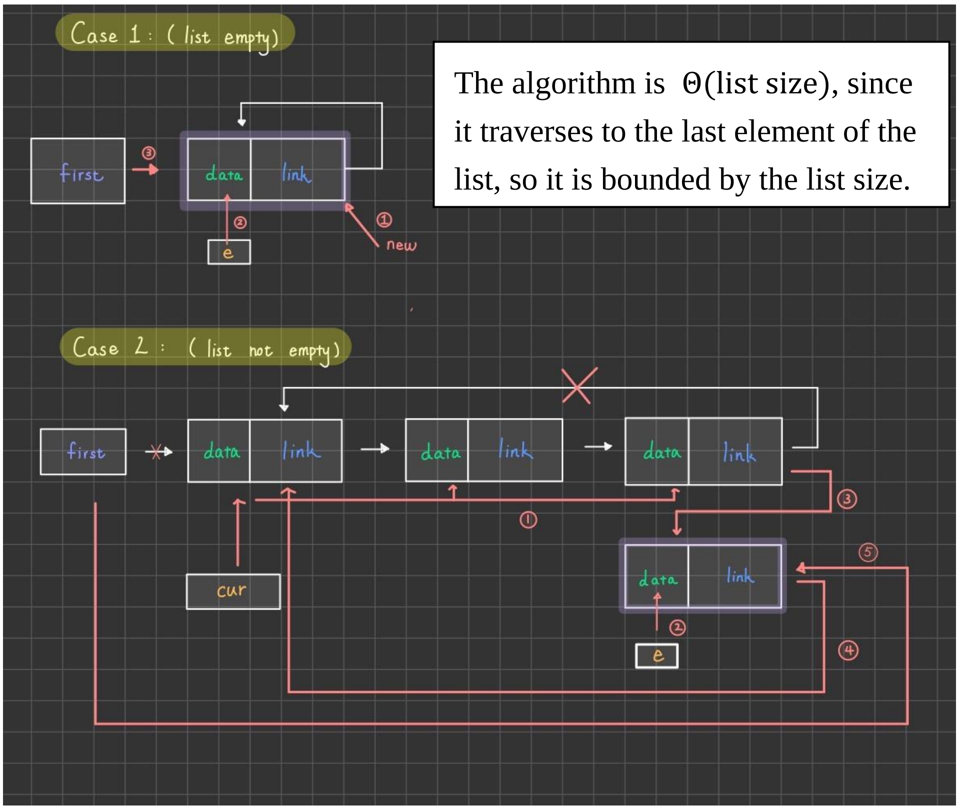
formulate algorithms (pseudo code OK, C++ code not necessary) to

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



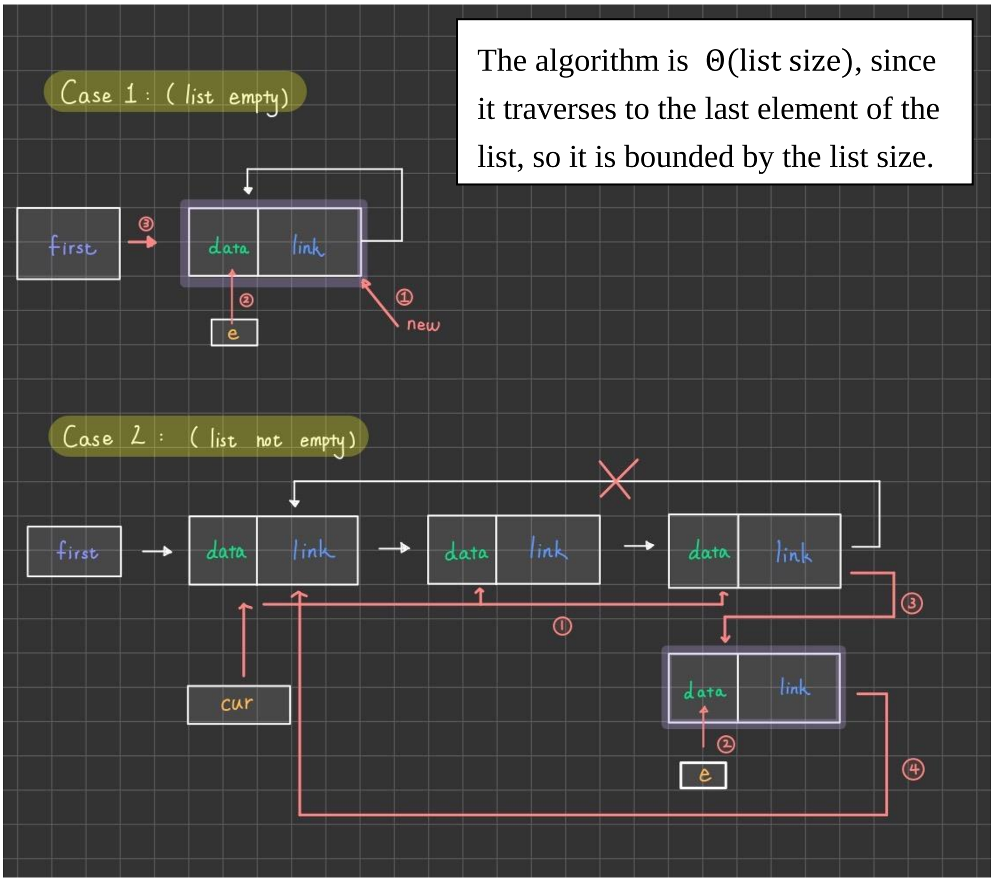

```
template <class T>
int CircularList<T>::Size() {
    if(!first)
        return 0;

    int count = 0;
    ListNode<T>* cur = first;
    do {
        count ++;
        cur = cur->link;
    } while(cur != first);

    return count;
}
```

(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)




```
template <class T>
void CircularList<T>::InsertFront(const T& e) {
    if(!first) {
        ListNode<T>* newNode = new ListNode<T>;
        newNode->data = e;
        newNode->link = newNode;
        first = newNode;
        return;
    }

    ListNode<T>* cur = first;

    while(cur->link != first) {
        cur = cur->link;
    }

    ListNode<T>* newNode = new ListNode<T>;
    newNode->data = e;
    newNode->link = first;
    cur->link = newNode;
    first = newNode;
}
```

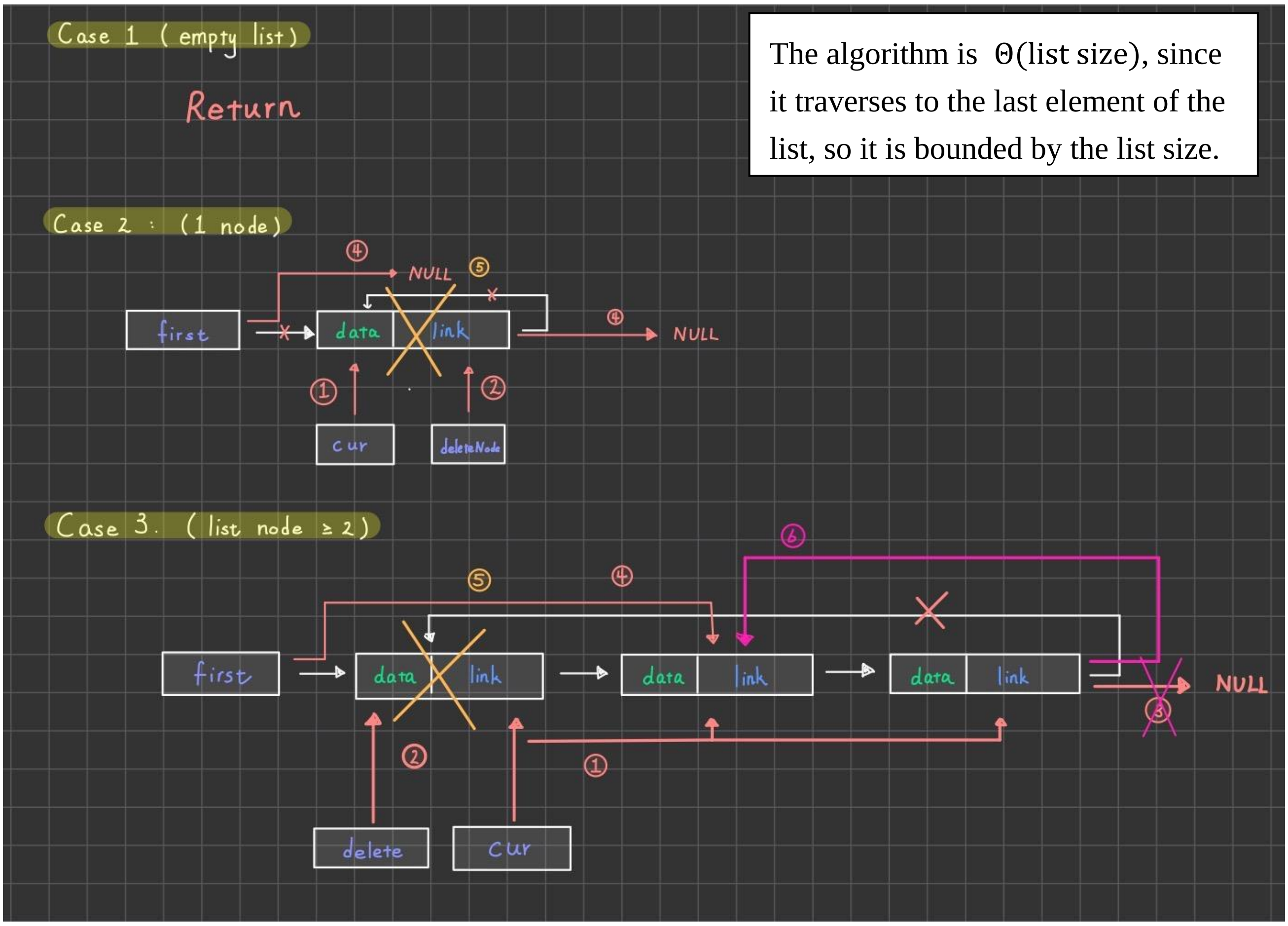
(c) insert a new node at the back (right after the last node) of the list. Discuss the time complexity of your algorithm. Explain your algorithm properly (using either text or graphs)




```cpp
template <class T>
void CircularList<T>::InsertBack(const T& e) {
    if(!first) {
        ListNode<T>* newNode = new ListNode<T>;
        newNode->data = e;
        newNode->link = newNode;
        first = newNode;
    }

    else {
        ListNode<T>* cur = first;
        while(cur->link != first) {
            cur = cur->link;
        }
        ListNode<T>* newNode = new ListNode<T>;
        newNode->data = e;
        newNode->link = first;
        cur->link = newNode;
    }
}
```

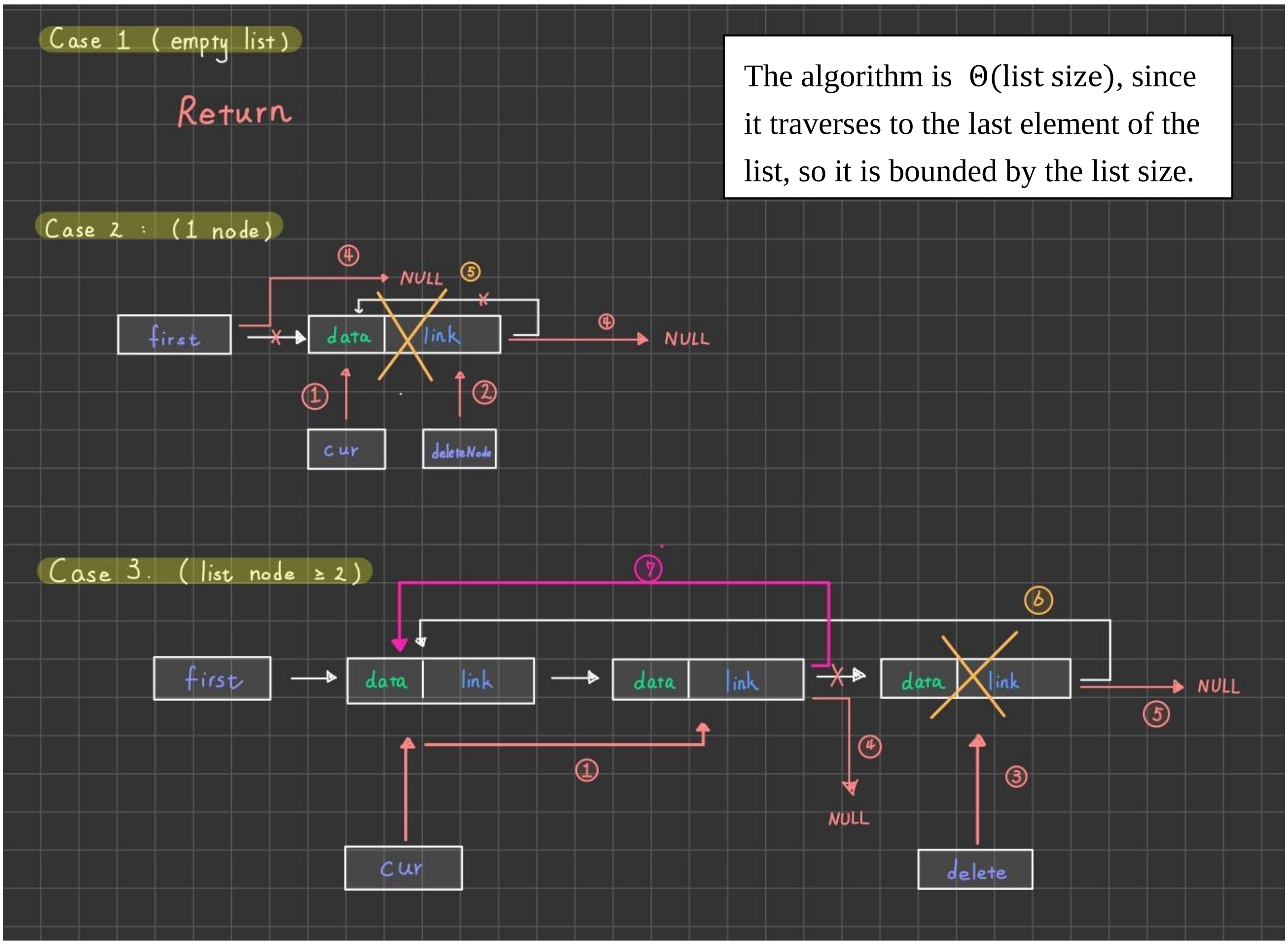
(d) delete the first node of the list. Discuss the time complexity of your algorithm. Explain your algorithm properly (using either text or graphs)

The algorithm is Θ(list size), since it traverses to the last element of the list, so it is bounded by the list size.



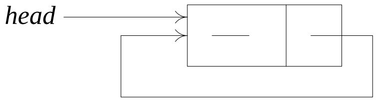

```cpp
template <class T>
void CircularList<T>::DeleteFront() {
    if(!first)
        return;                 // return if nothing to delete
    ListNode<T>* cur = first;   // use the ListNode pointer to traverse the list.
    while(cur->link != first) { // traverse until we reach the last node from first
        cur = cur->link;
    }

    ListNode<T>* deleteNode = cur->link; // assign a pointer to the first node
    cur->link = NULL;                    // let the last node points to NULL
    first = first->link;                 // first now then becomes the next node to first
    delete deleteNode;                   // delete the first node

    if(first)                            // if there is more than one node before delete
        cur->link = first;               // connects the last node to the new first.
}
```

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

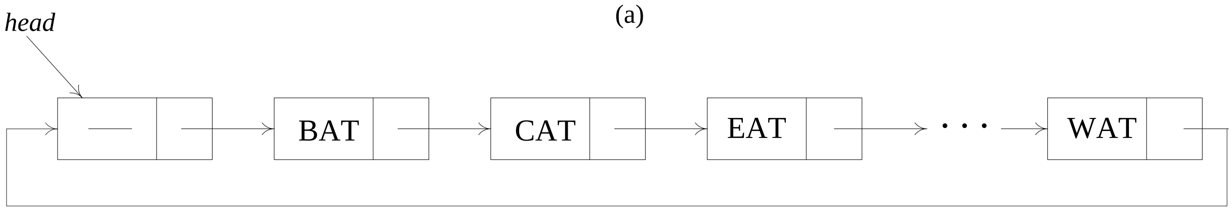



```
template <class T>
void CircularList<T>::DeleteBack() {
    if(!first)
        return;                     // return if nothing to delete
    ListNode<T>* cur = first;       // use the ListNode pointer to traverse the list.
    while(cur->link->link != first) { // traverse until we reach the second last node from first
        cur = cur->link;
    }

    ListNode<T>* deleteNode = cur->link;    // assign a pointer to the last node
    cur->link = NULL;                       // let the link to the last node break
    deleteNode->link = NULL;                // let the last node break it's link

    if(cur == deleteNode)                   // if there is only one node
        first = NULL;                       // let first points to NULL

    delete deleteNode;                      // delete the last node

    if(first)                               // if there is node in the list
        cur->link = first;                  // link the original second last node back to first node.
}
```

(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.



(a)



(b)

The following C++ code will be used with my self-created function deCircular() where it breaks the link from the last node to the head dummy node.

```cpp
template <class T>
void CircularList<T>::deCircular() {
    ListNode<T>* cur = head;
    if(head->link == head) {
        head->link = NULL;
        return;
    }

    while(cur->link != head) {
        cur = cur->link;
    }
    cur->link = NULL;
}
```

2-(a) (Circular Link with head node version)

```
template <class T>
int CircularList<T>::Size() {
    if(head->link == head)
        return 0;
    int count = 0;
    ListNode<T>* cur = head->link;
    do {
        count ++;
        cur = cur->link;
    }while (cur != head);

    return count;
}
```

2-(b) (Circular Link with head node version)

```
template <class T>
void CircularList<T>::InsertFront(const T& e) {
    if(head->link == head) {
        ListNode<T>* newNode = new ListNode<T>;
        newNode->data = e;
        newNode->link = head;
        head->link = newNode;
        return;
    }

    ListNode<T>* newNode = new ListNode<T>;
    newNode->link = head->link;
    newNode->data = e;
    head->link = newNode;
}
```

2-(c) (Circular Link with head node version)

```
template <class T>
void CircularList<T>::InsertBack(const T& e) {
    if(!head->link) {
        ListNode<T>* newNode = new ListNode<T>;
        newNode->data = e;
        newNode->link = head;
        head->link = newNode;
    }

    else {
        ListNode<T>* cur = head->link;
        while(cur->link != head) {
            cur = cur->link;
        }
        ListNode<T>* newNode = new ListNode<T>;
        newNode->data = e;
        newNode->link = head;
        cur->link = newNode;
    }
}
```

2-(d) (Circular Link with head node version)

```
template <class T>
void CircularList<T>::DeleteFront() {
    if(head->link == head)
        return;

    ListNode<T>* deleteNode = head->link;
    head->link = head->link->link;
    delete deleteNode;
}
```

2-(e) (Circular Link with head node version)

```
template <class T>
void CircularList<T>::DeleteBack() {
    if(head->link == head)
        return;
    ListNode<T>* cur = head->link;
    while(cur->link->link != head) {
        cur = cur->link;
    }

    ListNode<T>* deleteNode = cur->link;
    cur->link = NULL;
    deleteNode->link = NULL;

    delete deleteNode;

    if(head->link)
        cur->link = head;

    else
        head->link = head;
}
```

1-(b) (Circular Link with head node version)

```
template <class T>
void CircularList<T>::ReplaceNode(int k, T& newData) {
    ListNode<T>* cur = head->link;
    for(int i = 1; i < k; i++) {
        if(cur == head)
            cerr << "Given index is unttainable" << endl;
        else
            cur = cur->link;
    }
    cur->data = newData;
}
```

1-(c) (Circular Link with head node version)

```
template <class T>
void CircularList<T>::InsertionBeforeK(int k, T& newData) {
    ListNode<T>* cur = head->link;
    ListNode<T>* newNode = new ListNode<T>;
    newNode->data = newData;
    newNode->link = NULL;

    for(int i = 1; i < k - 1; i++) {
        if(cur == head)
            cerr << "Given index is unattainable";
        else
            cur = cur->link;
    }

    newNode->link = cur->link;
    cur->link = newNode;
}
```

1-(d) (Circular Link with head node version)

```
/**/
template <class T>
void CircularList<T>::deleteAllOddNode() {
    deCircular();
    if(head->link != head) {
        ListNode<T>* tempFirst = head->link;
        head->link = head->link->link;
        //tempFirst->link = NULL;
        delete tempFirst;

        if(head->link) {
            ListNode<T>* evenNode = head->link;
            ListNode<T>* oddNode = head->link->link;


            while (evenNode && oddNode )  // while the #even and #odd node exists
            {
                evenNode->link = oddNode->link;
                delete oddNode;

                evenNode = evenNode->link;

                if(evenNode)    // if #even node is not the end of the list
                    oddNode = evenNode->link;
            }

            ListNode<T>* cur = head;
            while(cur->link) {
                cur = cur->link;
            }
            cur->link = this->head;
        }
    }
}
```

1-(e) (Circular Link with head node version)

```
template <class T>
void CircularList<T>::divideMid(CircularList<T>* subList) {
    int count = this->Size();
    int mid = (count / 2 + count % 2);
    ListNode<T>* cur = head->link;

    if(count == 0 || count == 1)
        return;
    for(int i = 1; i < mid; i++) {
        cur = cur->link;
    }

    subList->head->link = cur->link;
    cur->link = this->head;

    ListNode<T>* subCur = subList->head;
    while(subCur->link != head) {
        subCur = subCur->link;
    }
    subCur->link = subList->head;
}
```

1-(f) (Circular Link with head node version)

```cpp
template <class T>
CircularList<T>* CircularList<T>::deconcatenate(int k) {
    if(head->link == head)
        return NULL;
    ListNode<T>* cur = this->head->link;
    CircularList<T>* newCircularList = new CircularList<T>;
    int count = 0;
    while(cur->link != head) {
        if(count == k - 1) {
            newCircularList->head->link = cur->link;
            cur->link = this->head;

            ListNode<T>* sublistCur = newCircularList->head;
            while(sublistCur->link != this->head) {
                sublistCur = sublistCur->link;
            }

            sublistCur->link = newCircularList->head;
            return newCircularList;
        }

        else {
            cur = cur->link;
            count++;
        }

    }
    cerr << "Given index is unattainable.";
    return NULL;
}
```

1-(g) (Circular Link with head node version)

```cpp
template <class T>
CircularList<T>* CircularList<T>::merge(CircularList<T>* CL2) {
    CircularList<T>* CL3 = new CircularList<T>;
    if(this->head->link == this->head && CL2->head->link == CL2->head)
        return CL3;
    if(this->head->link == this->head) {
        CL3->head->link = CL2->head->link;
        CL2->head->link = CL2->head;
        return CL3;
    }

    if(CL2->head->link == CL2->head) {
        CL3->head->link == this->head->link;
        this->head->link = this->head;
        return CL3;
    }

    deCircular();
    CL2->deCircular();

    ListNode<T>* N1 = this->head->link->link;
    ListNode<T>* N2 = CL2->head->link;
    ListNode<T>* cur = NULL;
    CL3->head->link = this->head->link;
    cur = CL3->head->link;

    while(N1 != NULL && N2 != NULL) {
        cur->link = N2;
        cur = cur->link;
        N2 = N2->link;

        cur->link = N1;
        cur = cur->link;
        N1 = N1->link;
    }
```

```cpp
    if(N1) {
        cur->link = N1;
        while (cur->link)
        {
            cur = cur->link;
        }
        cur->link = CL3->head;
    }

    if(N2) {
        cur->link = N2;
        while (cur->link)
        {
            cur = cur->link;
        }
        cur->link = CL3->head;
    }

    this->head->link = this->head;
    CL2->head->link = CL2->head;
    return CL3;


}
#endif
```

3. (15%) The class List<T> is shown below,

```
template <class T> class List;
template <class T>
class Node{
friend class List<T>;
private:    T data;
            Node* link;
};
template <class T>

class List{
public:
    List(){first = 0;}
    void InsertBack(const T& e);
    void Concatenate(List<T>& b);
    void Reverse();
    class Iterator{
    ….
    };
    Iterator Begin();
    Iterator End();
private:
    Node* first;
};
```

Prerequisite: create InsertFront class method

```
/*InsertFront*/
template <class T>
void List<T>::InsertFront(T newData) {
    Node<T>* newNode = new Node<T>;
    newNode->data = newData;
    newNode->link = first;
    first = newNode;
}
```

(a) Implement (pseudo code or C++) the stack data structure as a derived class of the class List<T>.

Class of stack derived from List:

```
template<class T>
class Stack:public List<T> {
public:
    Stack():List<T>(){}
    ~Stack(){}
    void Pop();
    void Push(T newdata);
    bool isEmpty();
    T Top();
private:
};
```

Top Class method: Returns the top element of the stack

isEmpty Class method: Returns Boolean value of whether the stack is empty or not

```
template <class T>
T Stack<T>::Top() {
    return this->Front();
}
```

```
template <class T>
bool Stack<T>::isEmpty() {
    return this->first == NULL;
}
```

Push Class method: Push the element into the stack.

Pop Class method: pop the top element of the stack if not empty.

```
template <class T>
void Stack<T>::Push(T newData) {
    this->InsertFront(newData);
}
```

```
template <class T>
void Stack<T>::Pop() {
    this->DeleteFront();
}
```

(b) Implement (pseudo code or C++) the queue data structure as a derived class of the class List<T>.

Class of queue derived from List:

```
template<class T>
class Queue:public List<T> {
public:
    Queue():List<T>(){}
    ~Queue(){}
    void Pop();
    void Push(T);
    bool isEmpty();
private:
};
```

is_empty Class method: Returns Boolean value of whether the queue is empty or not.

```
template <class T>
bool Queue<T>::is_empty() {
    return this->first == NULL;
}
```

Push Class method: Push the element into the queue.

```
template <class T>
void Queue<T>::Push(T newData) {
    this->InsertBack(newData);
}
```

Pop Class method: pop the front element of the stack if not empty.

```
template <class T>
void Queue<T>::Pop() {
    this->DeleteFront();
}
```

(c) Let $x_1, x_2, \ldots, x_n$ be the elements of a List<int> object. Each x_i is an integer. Formulate an algorithm (pseudo code OK, C++ code not necessary) to compute the expression $\sum_{i=1}^{n-5}(x_i \times x_{i+5})$

```
/**/
template<class T>
T& List<T>::mult_with_special_rule() {
    Iterator xi = Begin();           // start from the first element
    T sum = 0;
    for (; xi + 5 <= End(); xi++)    //ranging from the first to the
        sum += (*xi) * (*(xi + 5)); // dereferencing the element and sum the product
    return sum;
}
```