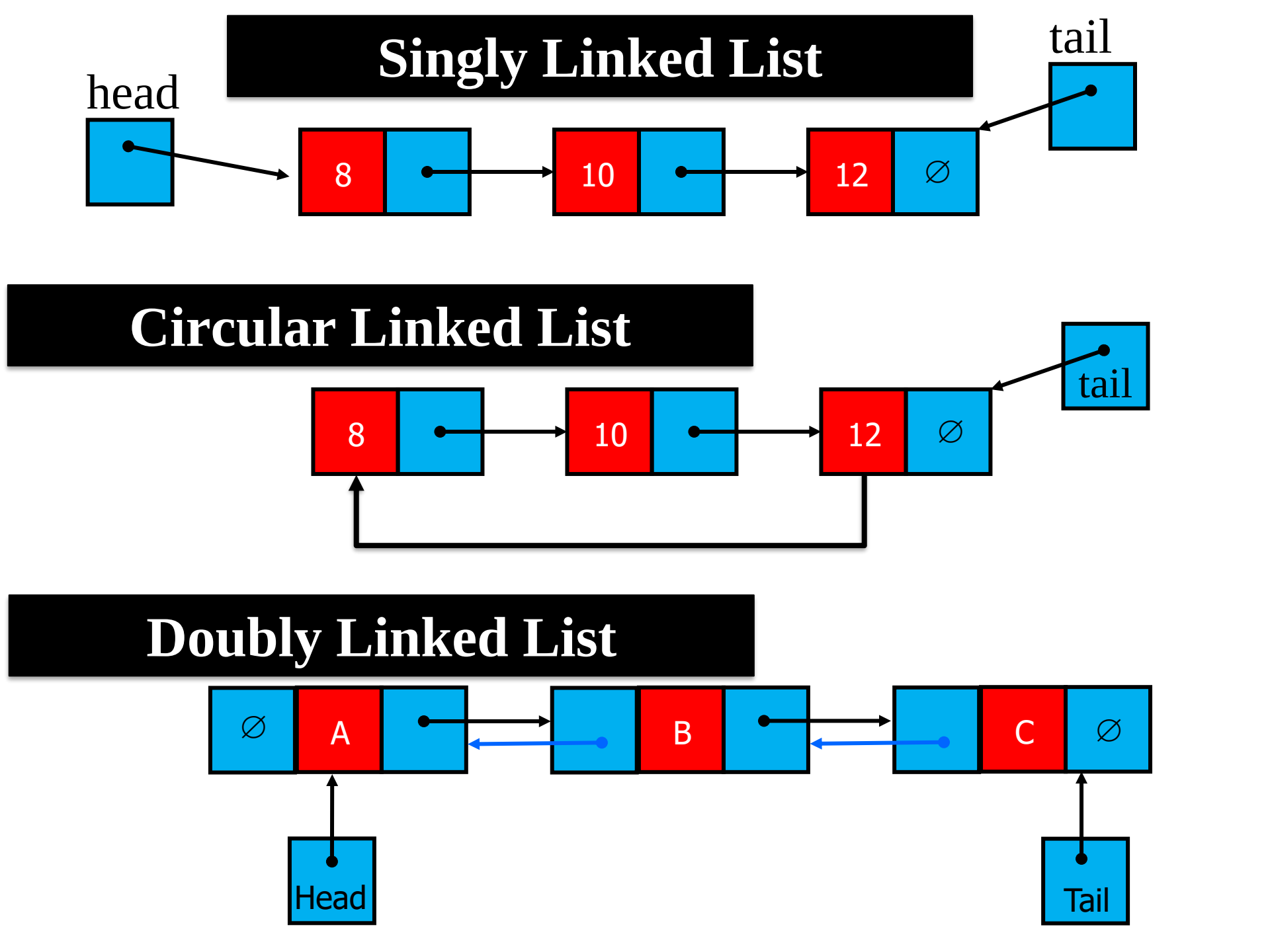
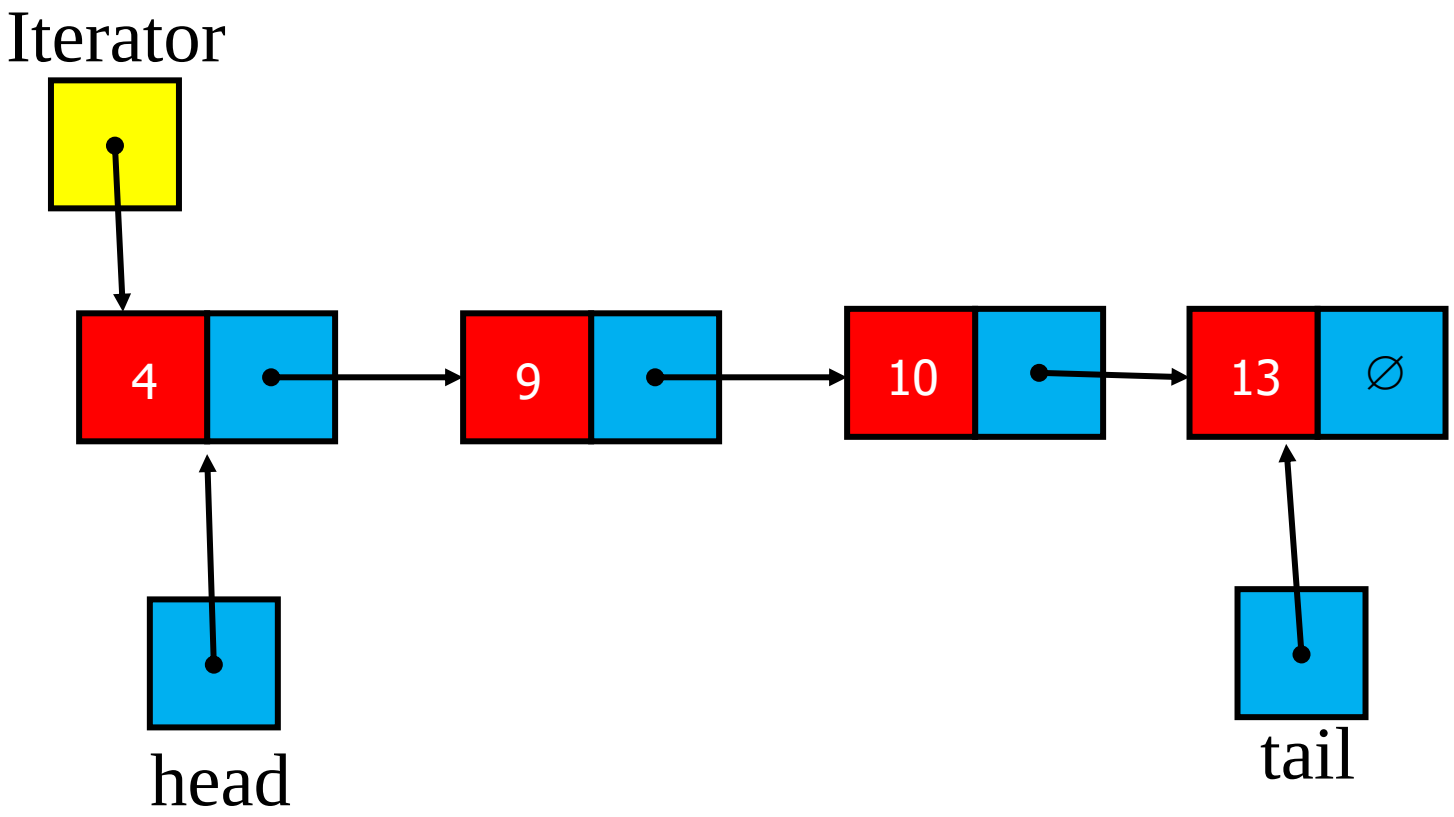
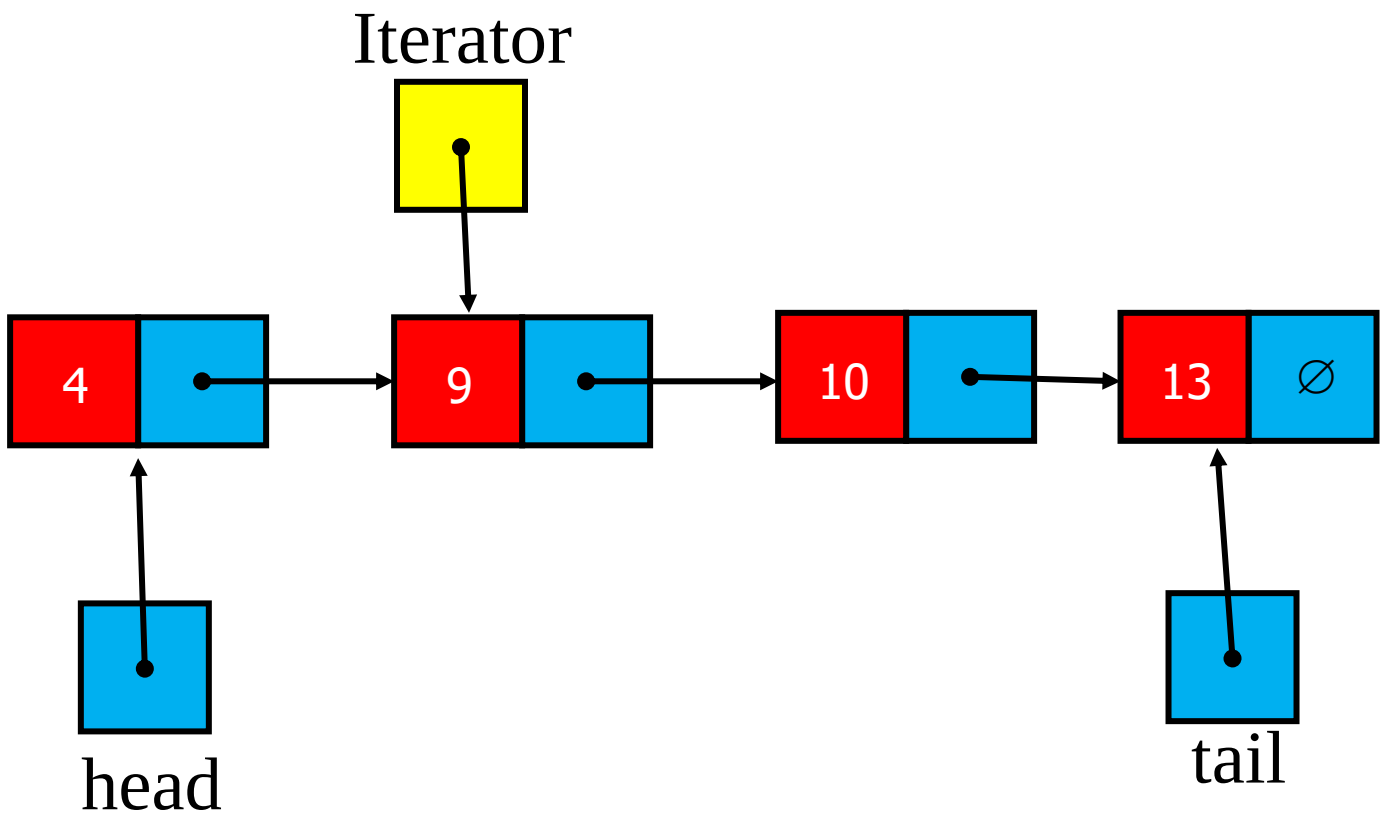
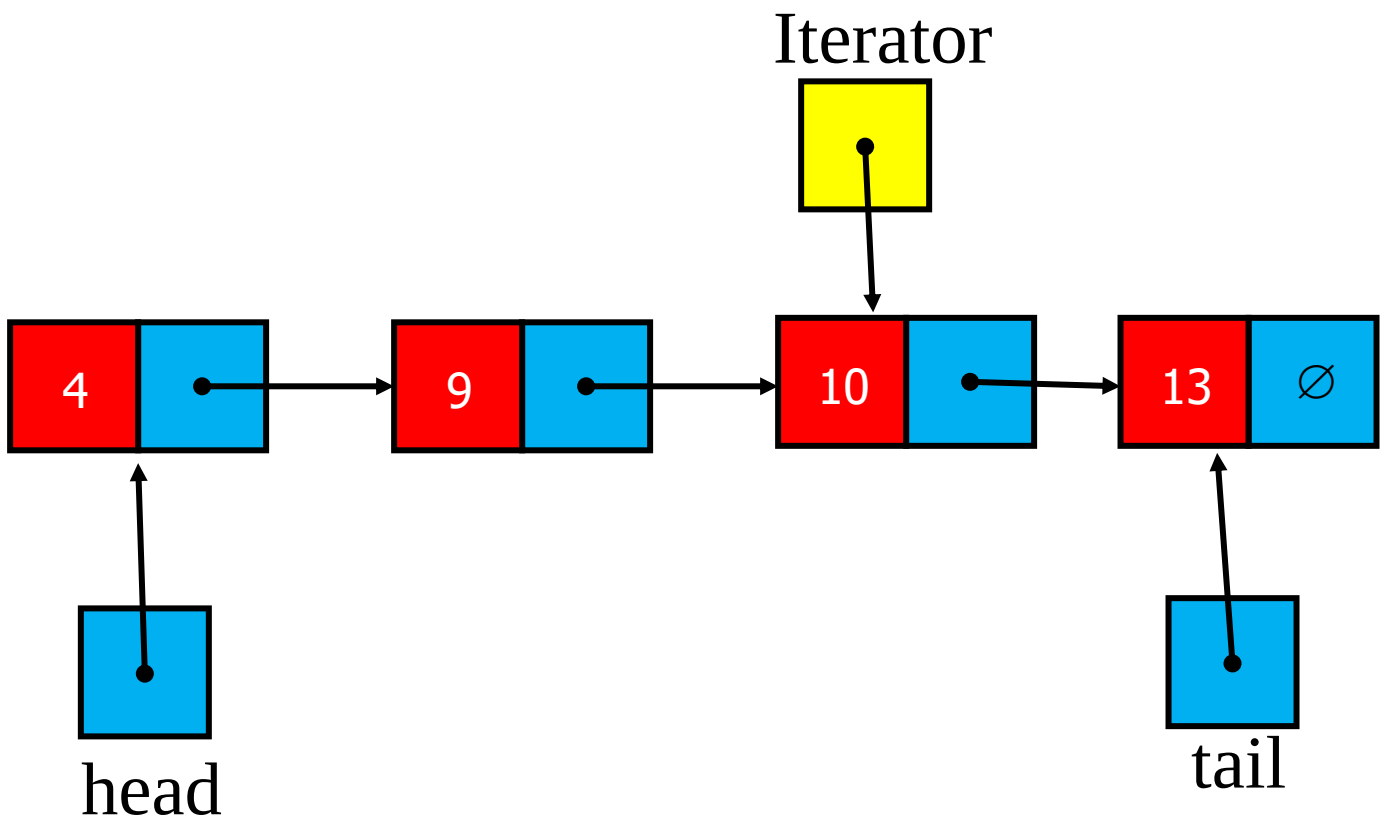

Singly Linked List
head
tail
8
10
12
∅
Circular Linked List
tail
8
10
12
∅
Doubly Linked List
∅
A
B
C
∅
Head
Tail


Iterator
4
9
10
13
∅
head
tail

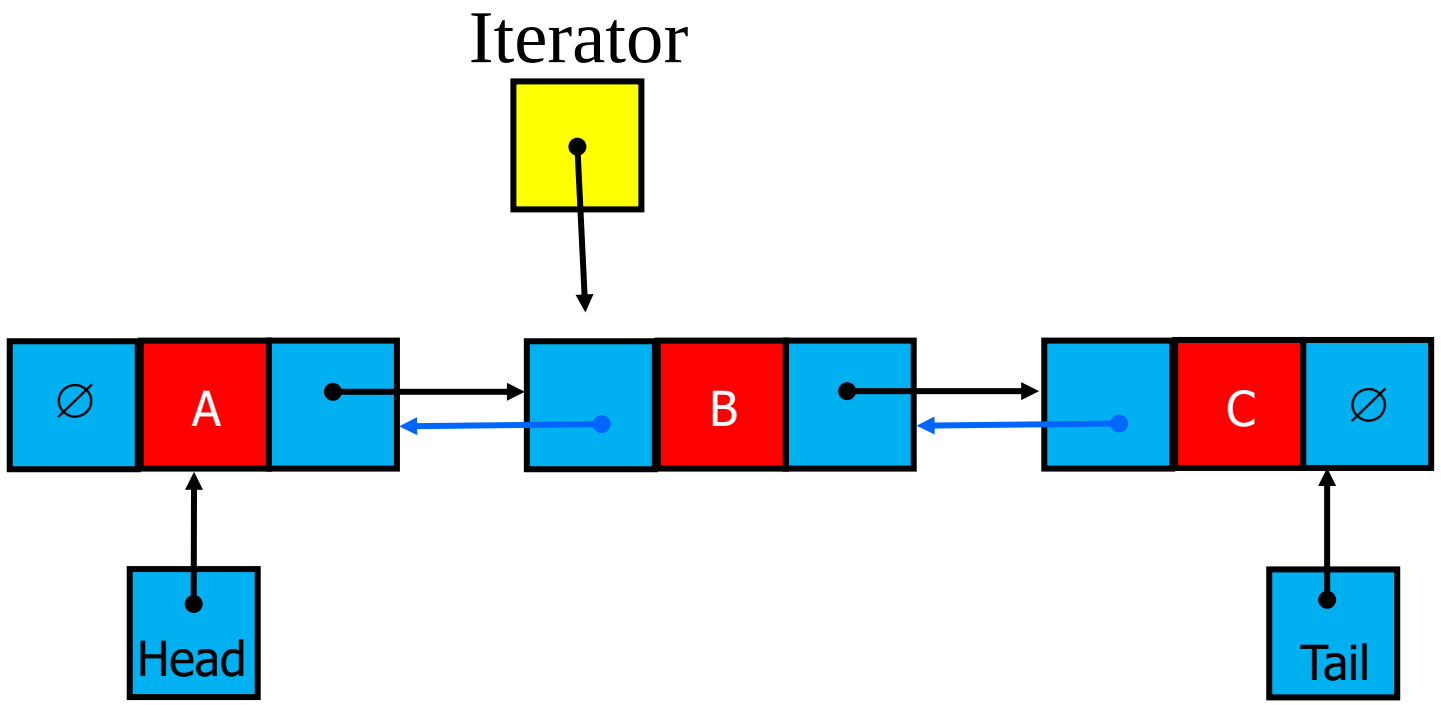

Iterator
4
9
10
13
∅
head
tail

Iterator++



Iterator++

Iterator--



Iterators

Data Structures and Algo Analysis in C++

Mark Allen Weiss

Chapter 3.3

Iterators

Why we need Iterators?

Iterator++ | Iterator--

Let's assume you have created a DLList and shipped it as a DDL file (library) for others to use

- Just like iostream library or string library

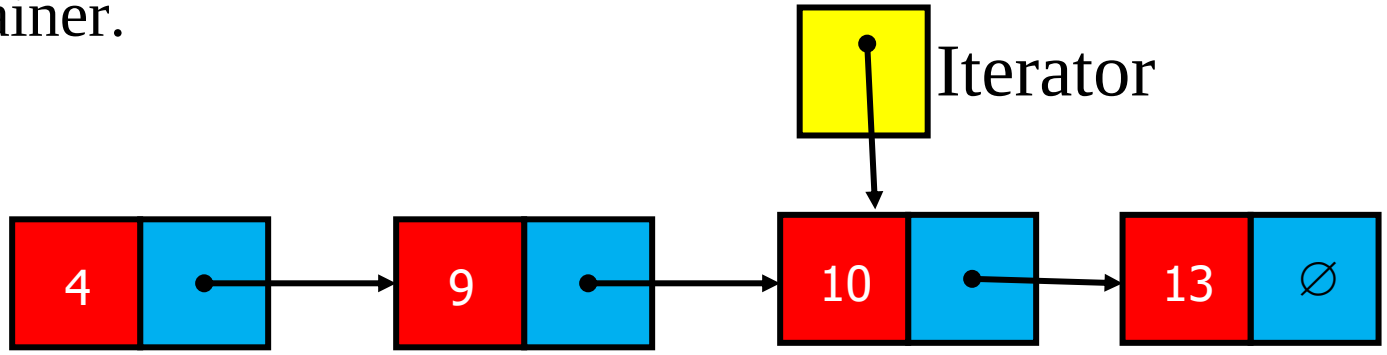
Now the user of this library wish to write new functionality

- find min, max or do some other work that require iterating the list or comparing some nodes of the list

But !!! the next and prev ptr in List class are private

Iterators

- An iterator allows a user to **process** every element of a container (List) while isolating the user from the **internal structure** of the container.



- This enable the List to store elements in any manner it wishes
  - while allowing the user to treat it as if it were a simple sequence or list.
- Iterators provide a consistent way to iterate on data structures of all kinds, and make the code more
  - ***readable,***
  - ***reusable, and***
  - ***less sensitive*** to a change

Nested Classes

```
class outer{
private:
    class inner{
        private:
            int x;
        public:
            void set_x(int a){x=a;}
    };
public:
        typedef inner nestedType;
};
```

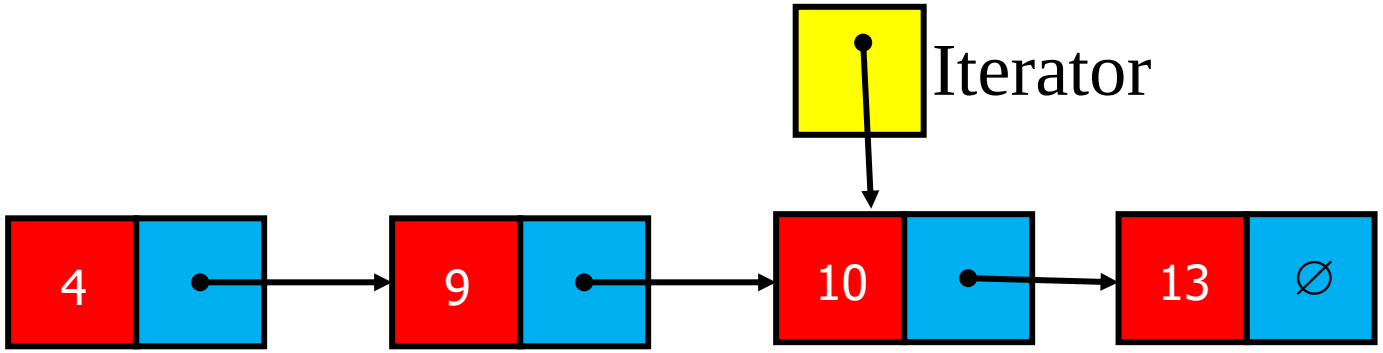
```
int main()
{
        outer o;
        //cannot do
        outer::inner nobj; //err
        //can do
        outer::nestedType yes;

}
```

- Generally we would like a **nested class to be private**
- We can declare an object of nested class in outer via the typedef
- No memory is allocated for object of type nestedType in outer class as no variable has been declared

ITERATORS

ITERATORS

- You can think of an iterator as an object
- The iterator object can traverse an ADT which has a collection of other objects
  - Like list, trees, etc.
- **EXAMPLE:** You can have an iterator which is a pointer to an item of the ADT
  - **Access:** Get data of an item in the ADT
  - **Traversal:** Modify itself to be able to point to next item

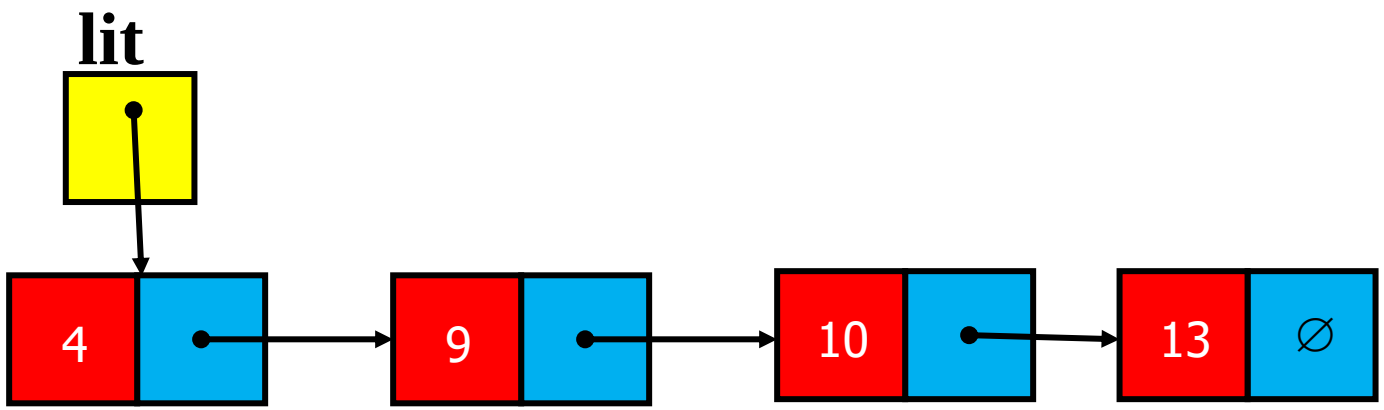


List Iterator

```
template<class T>
class List {
private:
        struct Node; // forward declaration
        Node * head, *tail;

    class ListIterator {
        public:
            ListIterator(Node * ptr = NULL) { iptr = ptr; };
            ListIterator & operator++() {//prefix ++
                if (iptr) iptr = iptr->next;
                return (*this);
            }
        private:
                Node * iptr;
    };

public:
    typedef ListIterator Iterator;
    Iterator begin() {Iterator I(head); return I; }
    Iterator end() { Iterator I(tail); return I; }

    List() { head = tail = 0; };
    ~List();
    Node * find(const T & val);
};
```

The caller

```
void main() {

    cout << "SL list Iterator" << endl;
    List<int> L;
    for (int i = 0; i < 5; i++)
        L.addToStart(i);

    List<int>::Iterator lit;

    for (lit = L.begin(); lit != nullptr; lit++)
        cout << *lit << endl;
}
```



List Iterator- *with forward class declaration*

```
template<class T>
class List {
     private:
     Node * head, *tail;

     class ListIterator;

     public:

          typedef ListIterator Iterator;
          void insert(Iterator it, const T & val);
          Iterator begin() { return head;};
          Iterator end() {return tail; };

          List() { head = tail = 0; };
          ~List();
          bool isEmpty() {return head == NULL;};
          Node * find(const T & val);
          void addToStart(const T & val);
          bool deleteFromTail();
          void print();


};
```

List Iterator- forward declaration

```
template<class T>
class List<T>::ListIterator {
    public:

        ListIterator(Node* t = NULL) { iptr = t; };

        ListIterator & operator++(int) {
        //dummy int param give info that it is postfix ++
            ListIterator old = *this;
            ++(*this)
            return old;
        }

    private:
        Node * iptr;
};
```

List Iterator- forward declaration

```
template<class T>
class List<T>::ListIterator {
    public:

        ListIterator(Node* t = NULL) { iptr = t; };

        ListIterator & operator++(int) {
        //dummy int param give info that it is postfix ++
            ListIterator old = *this;
            ++(*this)
            return old;
        }

        ListIterator & operator++() {//prefix ++
            if (iptr) iptr = iptr->next;
            return (*this);
        }

    private:
        Node * iptr;
};
```

List Iterator- forward declaration

```
template<class T>
class List<T>::ListIterator {
    public:

        ListIterator(Node* t = NULL) { iptr = t; };

        T & operator*(){
                return iptr->data;
        }

        bool operator==(const ListIterator & l) const
        {
                return iptr == l.iptr;
        }

        bool operator!=(const ListIterator & l) const {
                return !(iptr == l.iptr);
        }

    private:
        Node * iptr;
};
```

List Iterator- forward declaration

```
template<class T>
class List<T>::ListIterator {
    public:
        friend class List;
        ListIterator(Node* t = NULL) { iptr = t; };
        ListIterator & operator++(int) {//dummy int param give info that it is postfix ++
            ListIterator old = *this;
            ++(*this)
            return old;
        }
        ListIterator & operator++() {//prefix ++
            if (iptr) iptr = iptr->next;
            return (*this);
        }
        bool operator==(const ListIterator & l) const {
                return iptr == l.iptr;
        }
        T & operator*() { return iptr->data; }
        bool operator!=(const ListIterator & l) const { return !(iptr ==
        l.iptr); }
    private:
        Node * iptr;
};
```

The caller

```
void main() {

    cout << "SL list Iterator" << endl;
    List<int> L;
    for (int i = 0; i < 5; i++)
        L.addToStart(i);

    List<int>::Iterator lit=L.begin(), it2=L.begin();
    it2++;
    *lit = *it2
    // this will copy the data of the object iterator it2 points at to the object
    pointed at by lit
    //we can do this because  T & operator*() { return iptr->data; }

    for (lit = L.begin2(); lit != nullptr ;lit++)
        cout << *lit << endl;


}
```

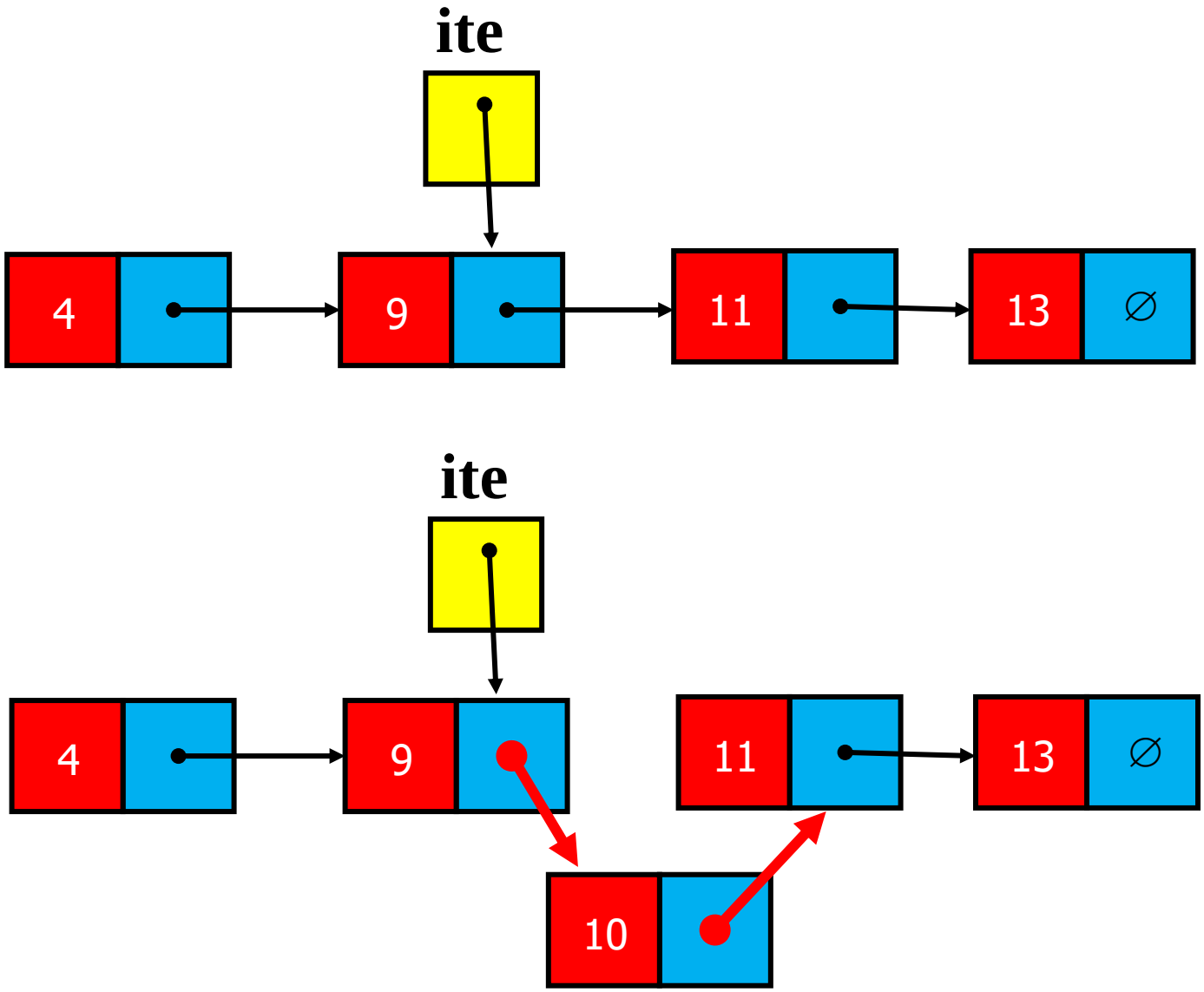
What does this loop Do ?

```
for (it = L.begin();it != L.end();){
        if(*it%2 ==0)
                it = L.delete(it);
        else
                ++it;
}
```

What does this function DO?

```
template<class T>
void Mystery(List<T> L1, List<T> L2, List<T> & L3 ) {

    List<int>::Iterator L1i = L1.begin(), L2i = L2.begin();
    while (L1i != nullptr && L2i != nullptr) {
        if (*L1i == *L2i) {
            L3.addToTail(*L1i);
            L2i++;
            L1i++;
        }
        else if (*L1i < *L2i)
                L2i++;
        else
                L1i++;
    }
}
```

insert



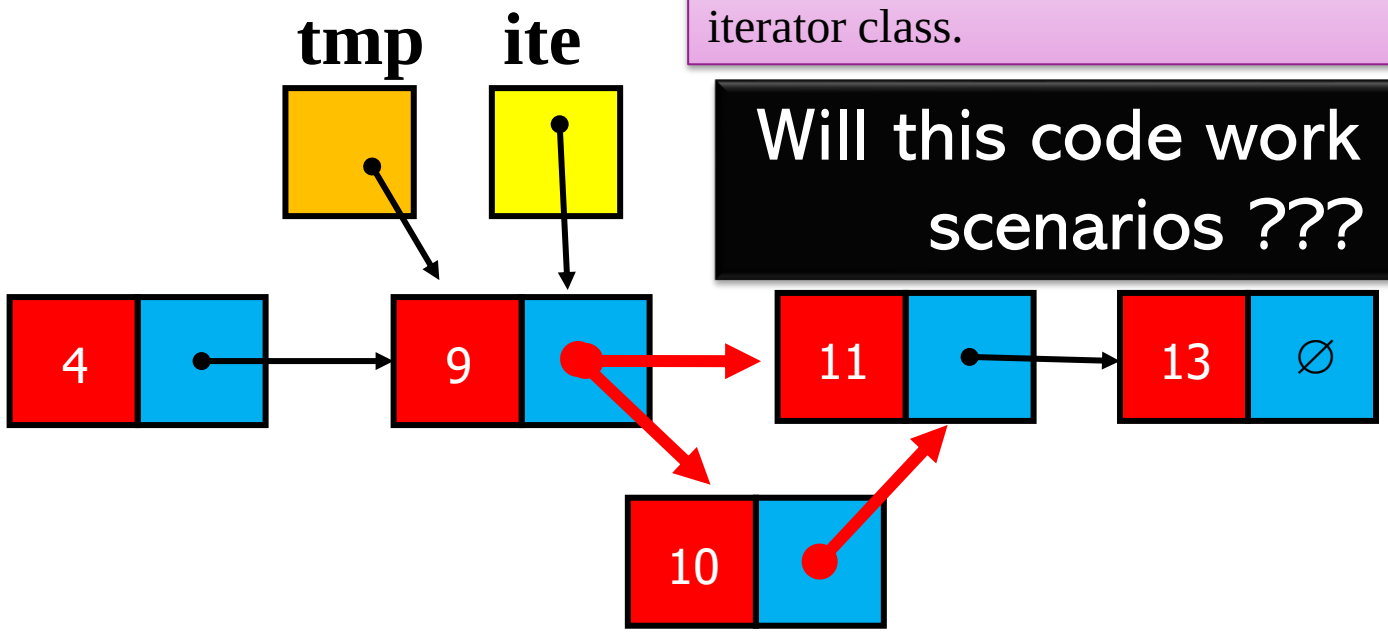
Insert

```
template<class T>
void List<T>::insert(const Iterator ite, const T & val) {

    Node * tmp = ite.iptr;

    Node * nptr = new Node(val, tmp->next);
    tmp->next = nptr;

}
```

Can we access the private member of nested class ListIterator in List ?

Make List class friend of ListIterator class
So only List class can use the functions of iterator class.

Will this code work in all scenarios ???



Insert

```
template<class T>
void List<T>::insert(const Iterator ite, const T & val) {

    Node * tmp = ite.iptr;

    if (head != NULL && tmp != NULL) {// not empty
        if (head == tail) {// only one element in current list
                if (head == tmp)
                        head->next = tail = new Node(val);
        }
        else {
            Node * nptr = new Node(val, tmp->next);
            tmp->next = nptr;
            if (tmp == tail)//last element
            tail = nptr;

        }
    }

}
```

Make List class friend of iterator class

So only List class can use the functions of iterator class.

Or

Make getter to access iptr – Note with getter any function can access the elements of the list

TO-DO Iterators

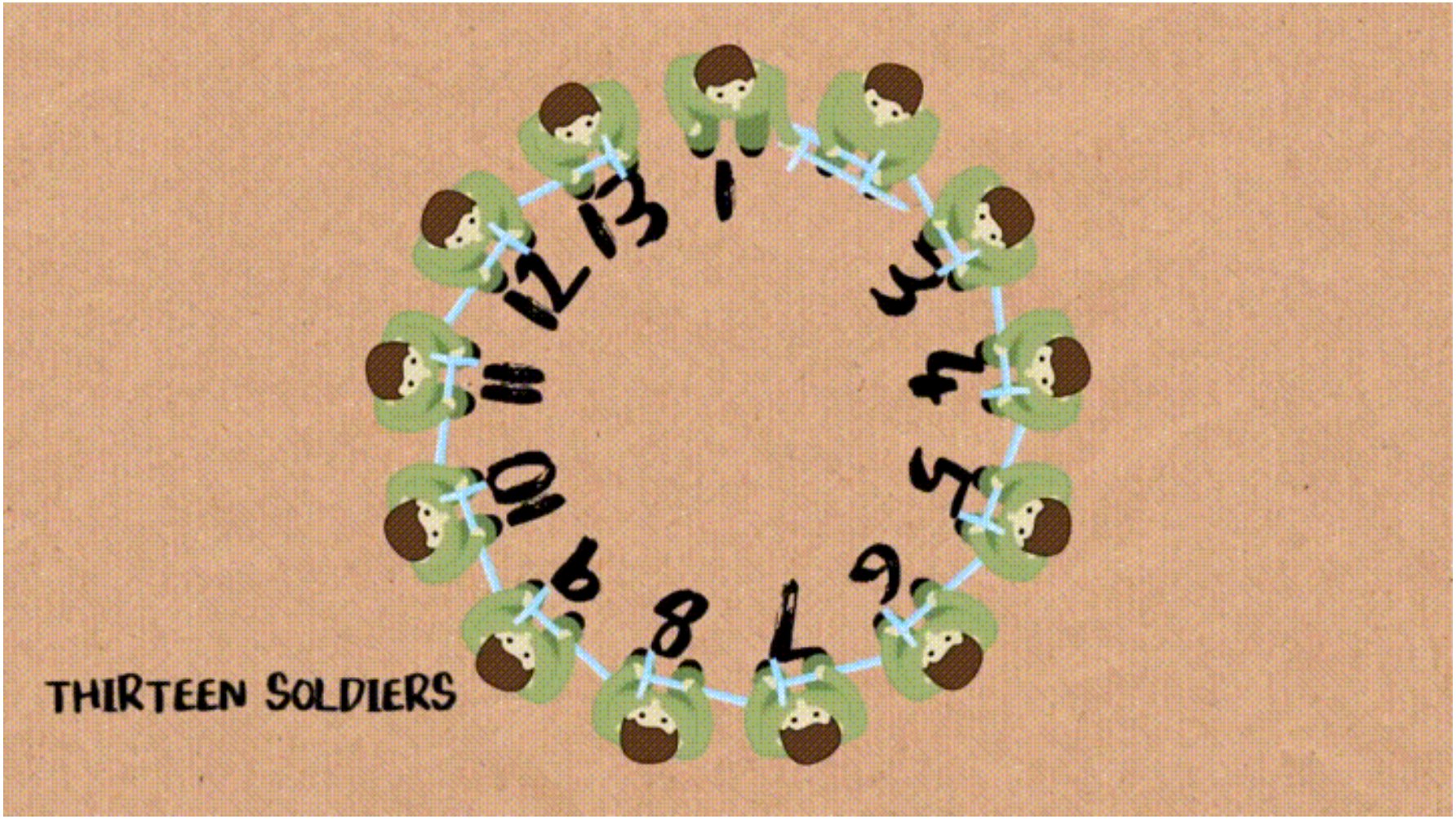
- random shuffle(*p*,*q*):
  - Rearrange the elements in the range from *p* to *q* in random order.
- reverse(*p*,*q*):
  - Reverse the elements in the range from *p* to *q*.
- find(*p*,*q*,*e*):
  - Return an iterator to the first element in the range
- min element(*p*,*q*):
  - Return an iterator to the minimum element in the range from *p* to *q*.
- max element(*p*,*q*):
  - Return an iterator to the maximum element in the range from *p* to *q*.
- for each(*p*,*q*, *f*):
  - Apply the function *f* the elements in the range from *p* to *q*.

Defining Functions outside the class

```
template<class T>
class List<T>::ListIterator {
    public:
        ListIterator(Node* t = NULL) { iptr = t; };
        ListIterator & operator++();
        bool & operator== (const ListIterator & l) const;
    private:
        Node * iptr;
};
```

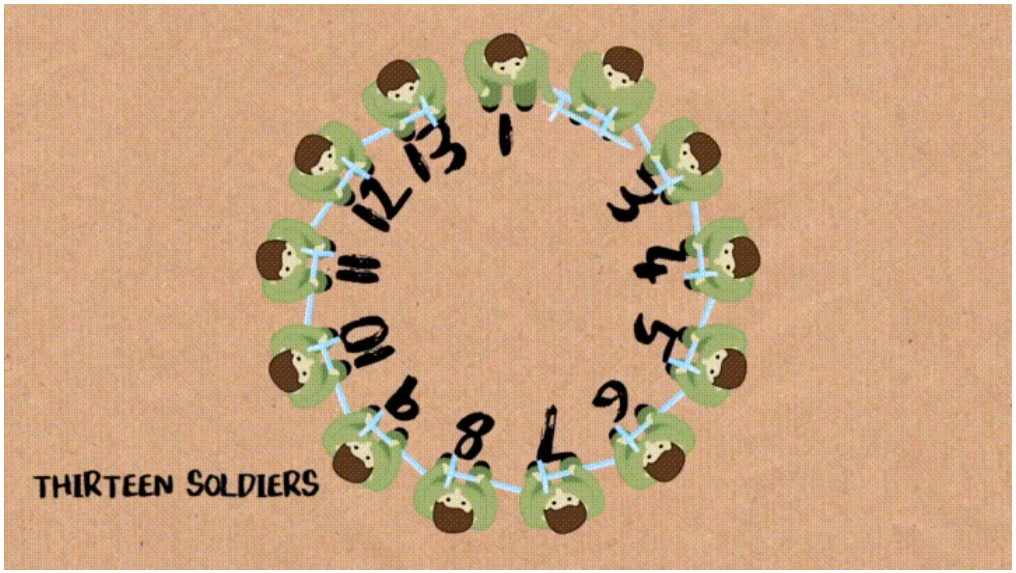
```
template<class T>
bool List<T>::ListIterator::operator==(const ListIterator & l) const
{
        return iptr == l.iptr;
}
template<class T>
typename List<T>::ListIterator & List<T>::ListIterator::operator++(){
    if (iptr) iptr = iptr->next;
    return (*this);
}
```

Example of Circular Link list with Iterator



Josephus Circle Problem

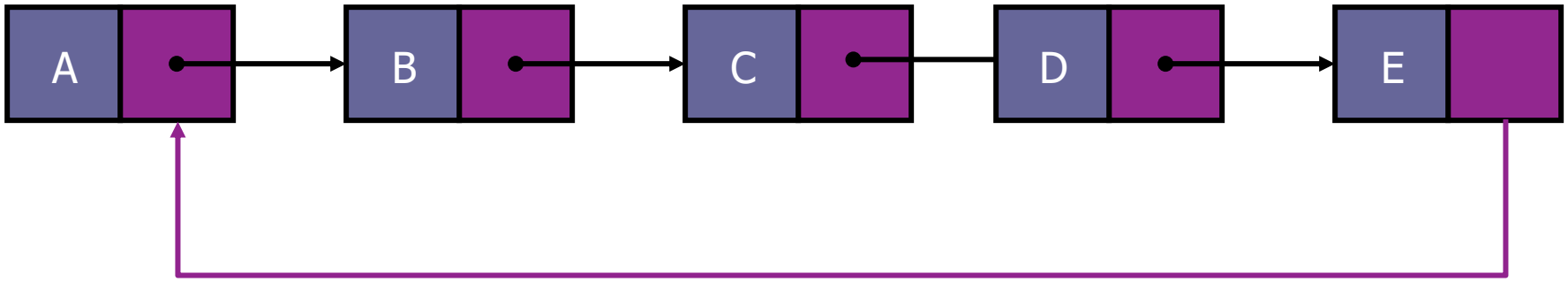
- There are n people standing in a circle waiting to be executed.



- The counting out begins at some point in the circle and proceeds around the circle in a fixed direction.
- In each step, a certain number of people are skipped, and the next person is executed.

Josephus Circle Problem

- The elimination proceeds around the circle and circle becomes smaller and smaller as the executed people are removed
- Till only the last person remains, who is given freedom.



Given the total number of persons n and a number m which indicates that m-1 persons are skipped and m-th person is killed in circle

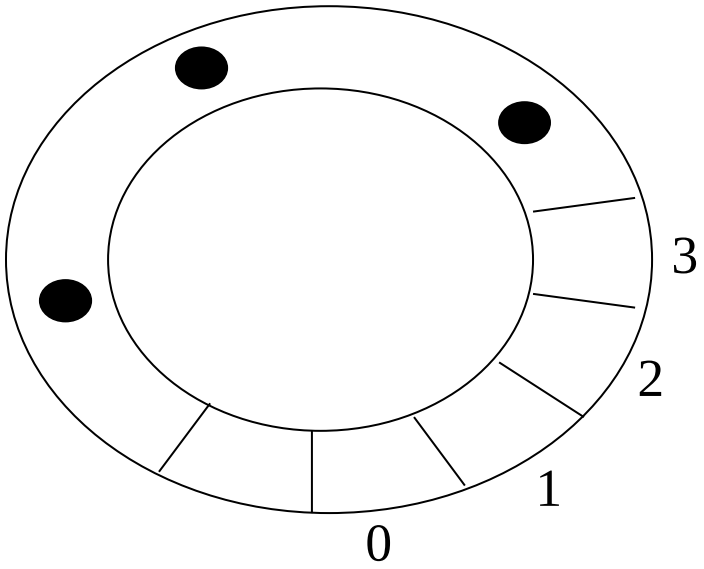
The task is to choose the place in the initial circle so that you are the last one remaining and so survive.

Josephus Problem– With Circular link list and circular Iterator

- Assume a circular linked list exists of size N.
- Function to find the only person left

```
void JosephusPosition(int m, CLList CL){
    CLList<int>::Iterator ite=CL.begin(); //circular link list iterator
    int count = 1;
    while (CL.size() != 1){/* while only one node is left */
    // Find m-th node
        count = 1;
        while (count < m-1){
            ite++
            count++;
      }
    /* Remove the m-th node, ite points to one prev node so Clink can remove
    it efficiently */
        CL.RemoveNext(ite);
    }
    cout<<"Last person is "<< *ite;
}
```

Josephus Problem with circular array



CAN we use circular array to solve Josephus Problem ?

Will it be more efficient or less than **CIRULAR LIST?**