

# Lists

# Today's Plan

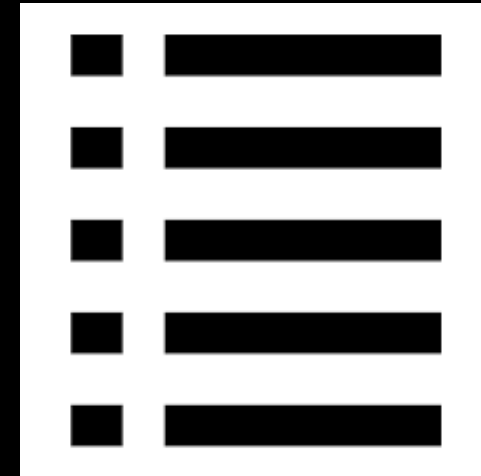


Lists

# List ADT

What makes a list?

E.g. Play**List**?

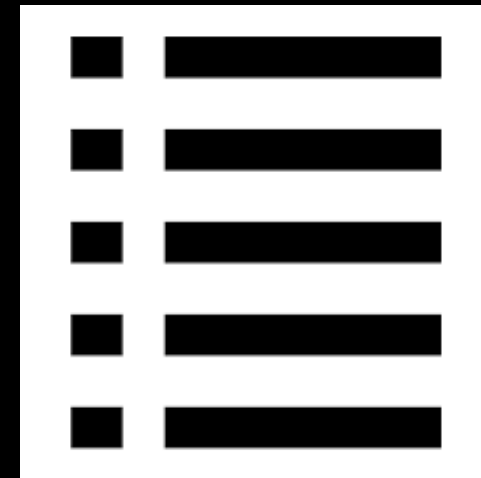


Duplicates allowed or not is not a defining factor

# List ADT

What makes a list?

E.g. Play**List**?



Duplicates allowed or not is not a defining factor

**ORDER!!!**

```

#ifndef LIST_H_
#define LIST_H_
template<typename ItemType> class List {
public:
    List(); // constructor
    List(const List<ItemType>& a_list); // copy constructor
    ~List(); // destructor
    bool isEmpty() const;
    size_t getLength() const;
    bool insert(size_t position, const ItemType& new_element);
        //retains list order, position is 0 to n-1, if position > n-1,
        //it inserts at end
    bool remove(size_t position); //retains list order
    ItemType getItem(size_t position) const;
    void clear();
private:
    //implementation details here
}; // end List
#include "List.cpp"
#endif // LIST_H_

```

Unsigned integer type. Guaranteed to store the max size of objects of any type.

# Implementation

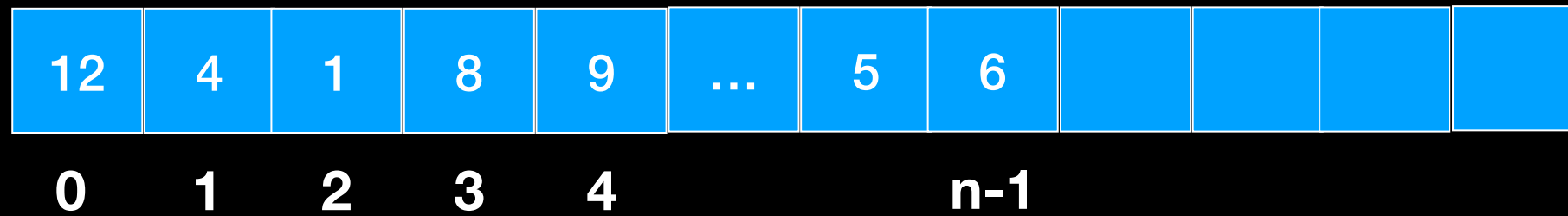


Must preserve order  
No swapping tricks

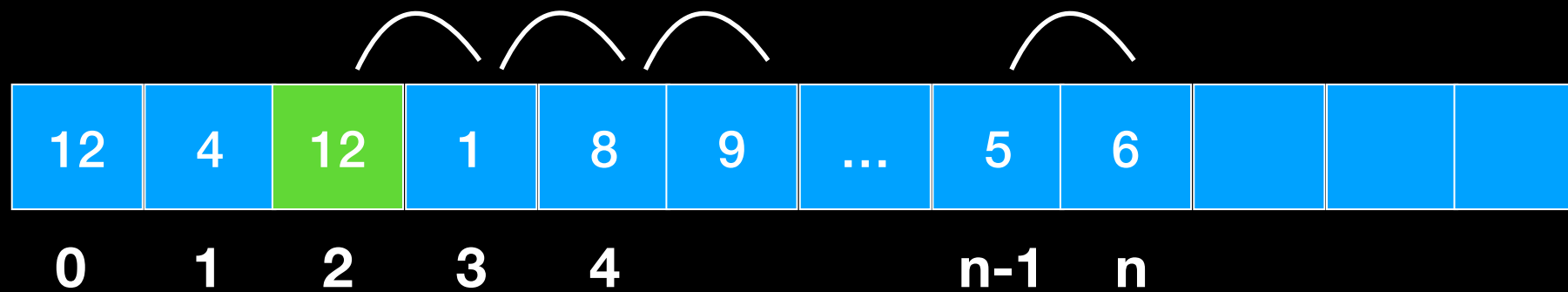
Array?

Linked Chain?

# Array

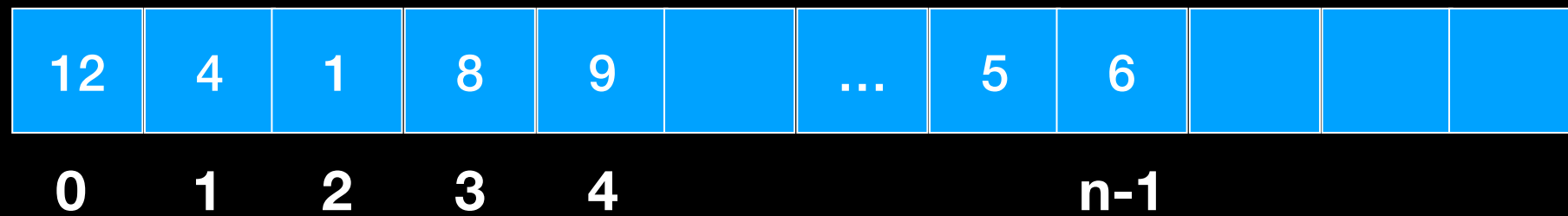


`insert(2, 12)`



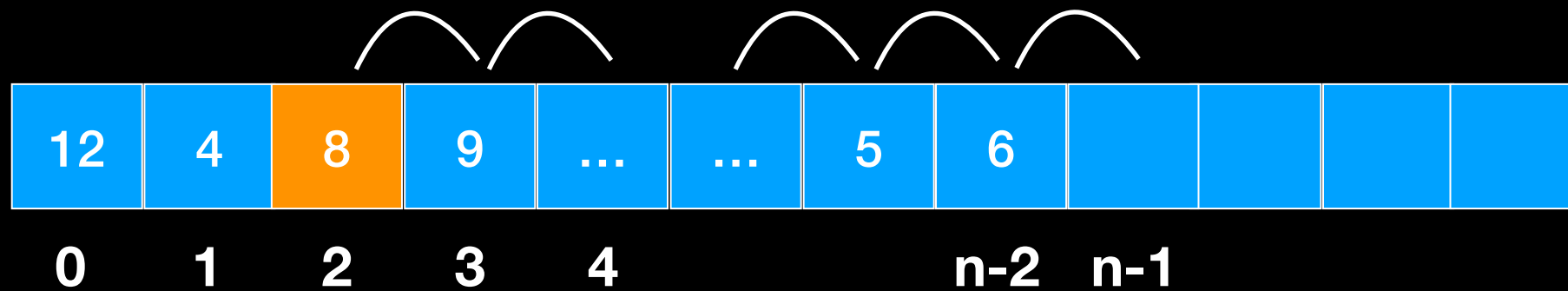
Must shift  $n$ -position elements

# Array



`remove(2)`

similarly



Must shift  $n - (\text{position} + 1)$  elements



# Array Analysis

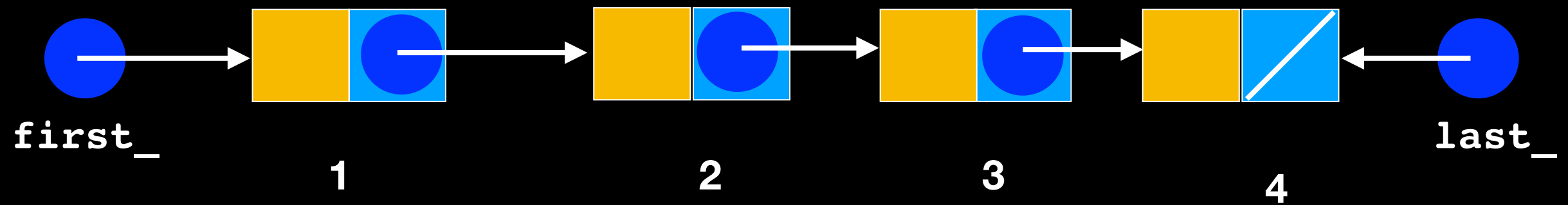
With Array both insert and remove are "Expensive"

Number of operations depends on size of List

Can we do better?

# What makes a list?

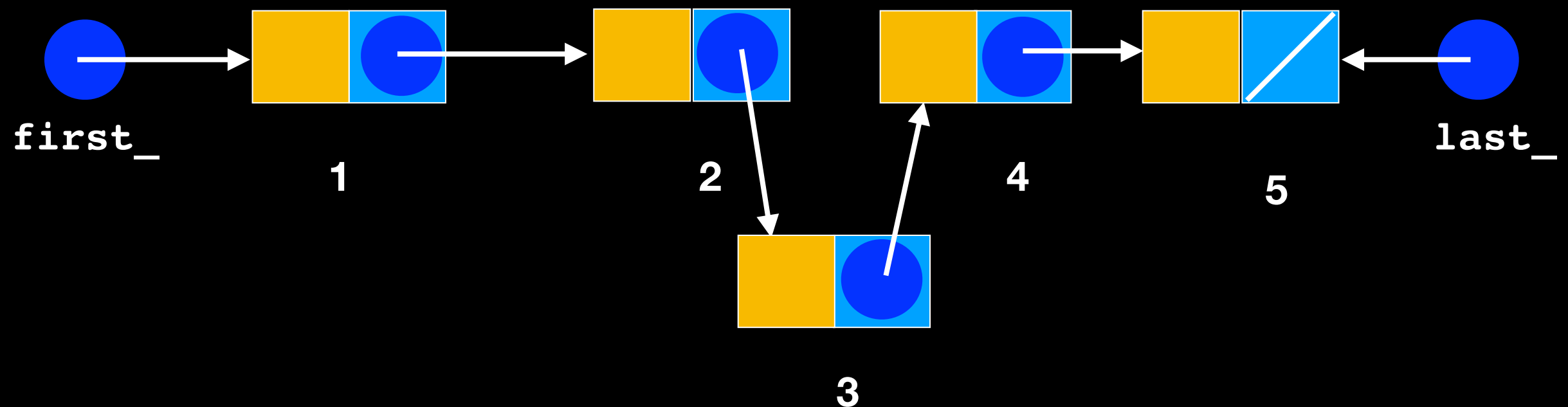
**Order** is implied



# What makes a list?

**Order** is implied

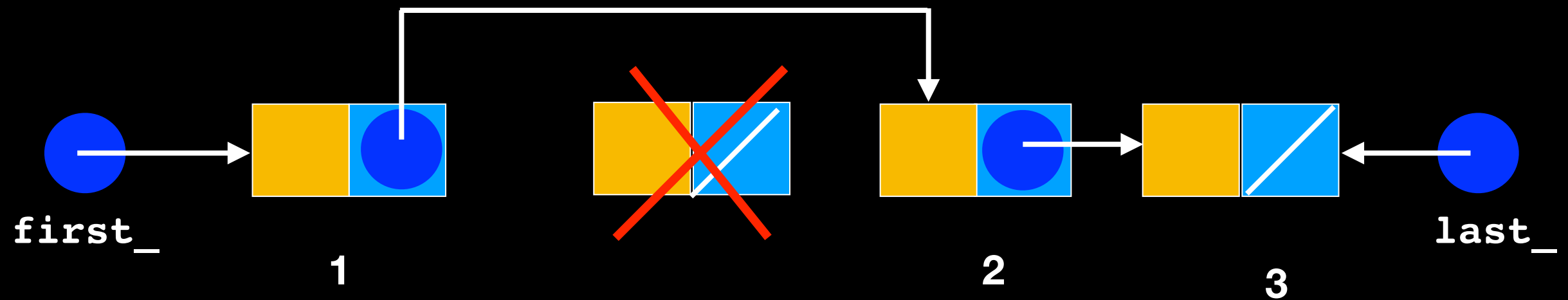
**Insertion** and removal from middle retains order



# What makes a list?

Order is implied

Insertion and **removal** from middle retains order



# What's the catch?

No random access

As opposed to arrays or vectors with direct indexing

“Expensive”: each insertion and removal must traverse `position+1` nodes

Here too, number of operations depends on size of List



Efficient, **does not depend on # of items**



Expensive, **depends on # of items**

	Arrays	Linked List
Random/direct access		
Retain order with Insert and remove At the <b>back</b>		
Retain order with insert and remove at <b>front</b>		
Retain order with insert and remove In the <b>middle</b>		



Efficient, **does not depend on # of items**



Expensive, **depends on # of items**

	Arrays	Linked List
Random/direct access		
Retain order with Insert and remove At the <b>back</b>		
Retain order with insert and remove at <b>front</b>		
Retain order with insert and remove In the <b>middle</b>		



Efficient, **does not depend on # of items**



Expensive, **depends on # of items**

	Arrays	Linked List
Random/direct access		
Retain order with Insert and remove At the <b>back</b>		
Retain order with insert and remove at <b>front</b>		
Retain order with insert and remove In the <b>middle</b>		





Efficient, **does not depend on # of items**



Expensive, **depends on # of items**

	Arrays	Linked List
Random/direct access		
Retain order with Insert and remove At the <b>back</b>		
Retain order with insert and remove at <b>front</b>		
Retain order with insert and remove In the <b>middle</b>		



Efficient, **does not depend on # of items**



Expensive, **depends on # of items**

No sifting but incurs cost of finding the node to remove (call to `getPointerTo`)



	Arrays	Linked List
Random/direct access		
Retain order with Insert and remove At the <b>back</b>		
Retain order with insert and remove at <b>front</b>		
Retain order with insert and remove In the <b>middle</b>		

# Singly-Linked List

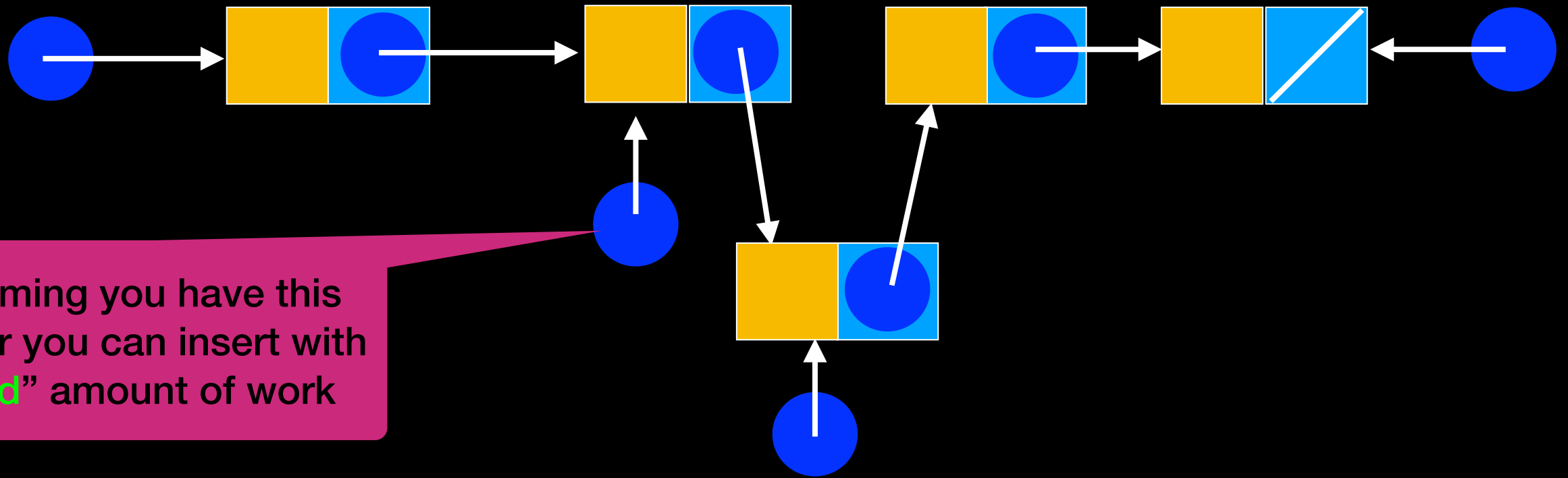
```

#ifndef LIST_H_
#define LIST_H_
template<typename ItemType> class List {
public:
    List(); //constructor
    List(const List<ItemType>& a_list); //copy constructor
    ~List(); //destructor
    bool isEmpty() const;
    size_t getLength() const;
    bool insert(size_t position, const ItemType&
                new_element); //retains list order, position is
                               //0 to n-1, if position > n-1 it inserts at end
    bool remove(size_t position); //retains list order
    ItemType getItem(size_t position) const;
    void clear();
private: //implementation details here
    Node<ItemType>* getPointerTo(size_t position) const;
}; // end List
#include "List.cpp"
#endif // LIST_H_

```

# INSERT

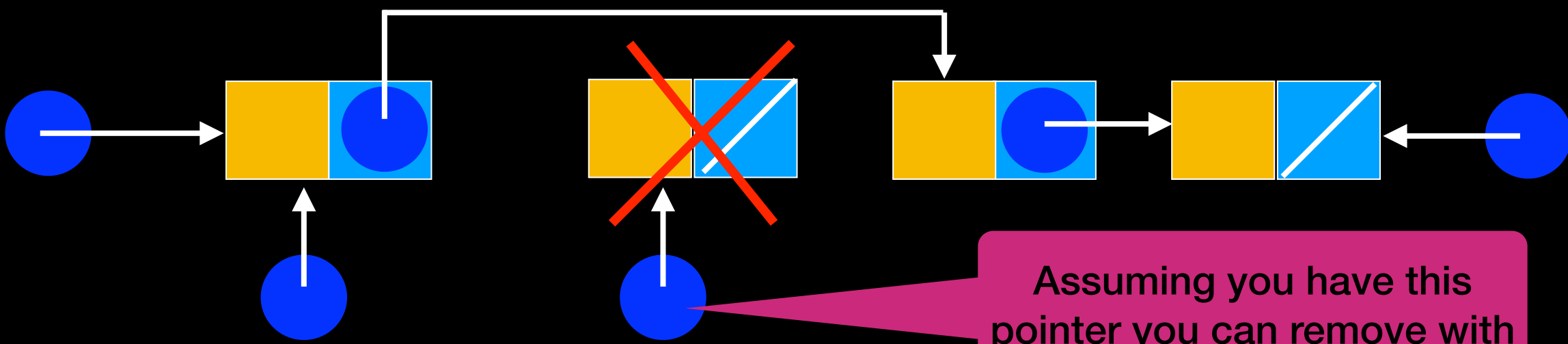
```
void insert(size_t position, ItemType new_element);
```



Assuming you have this pointer you can insert with "fixed" amount of work

# REMOVE

```
void remove(size_t position);
```



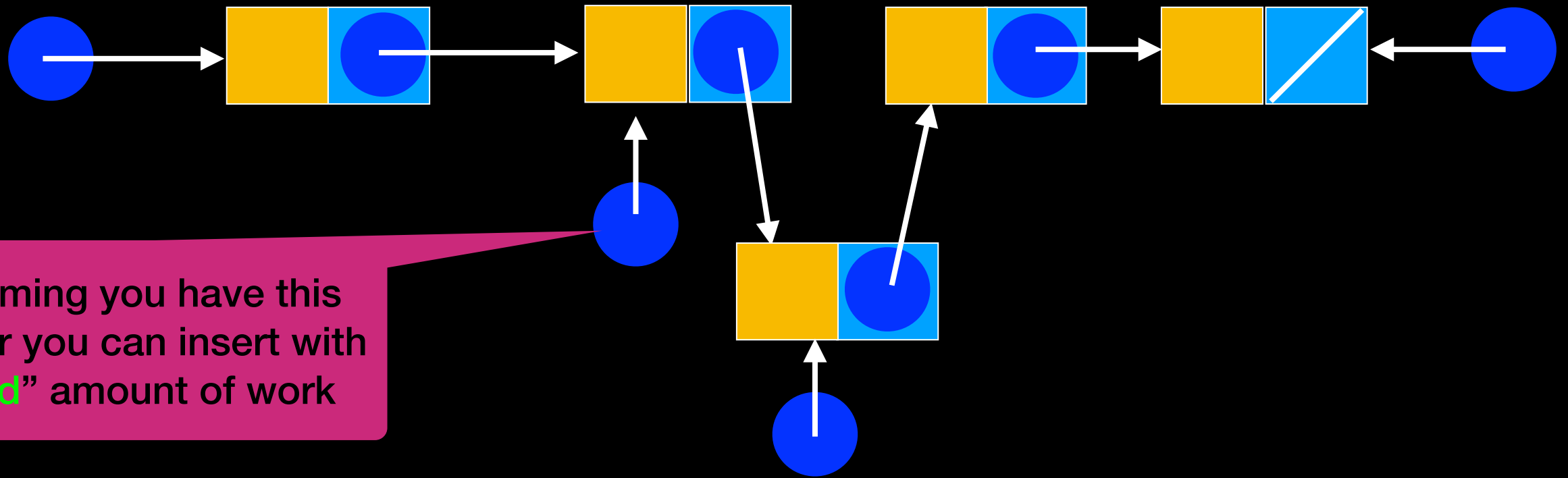
Assuming you have this pointer you can remove with "fixed" amount of work

# Caveat

Find the pointer to the node before inserting/  
removing —> **traversal: high cost - depends on  
number of elements in list**

# INSERT

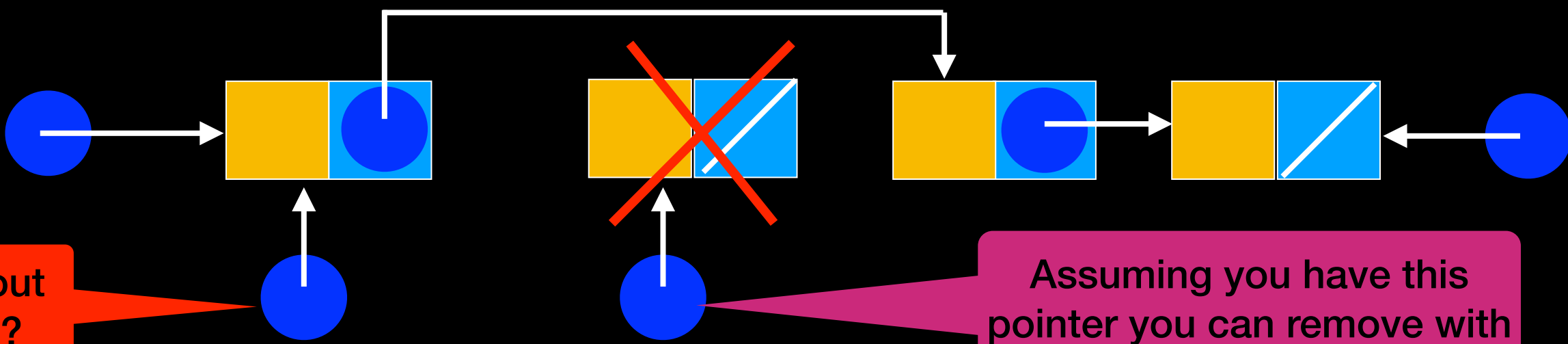
```
void insert(size_t position, ItemType new_element);
```



Assuming you have this pointer you can insert with “fixed” amount of work

# REMOVE

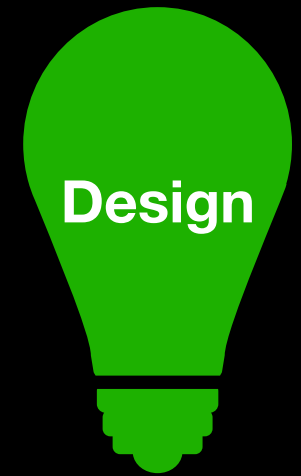
```
void remove(size_t position);
```



What about this one?

Assuming you have this pointer you can remove with “fixed” amount of work

# Lecture Activity

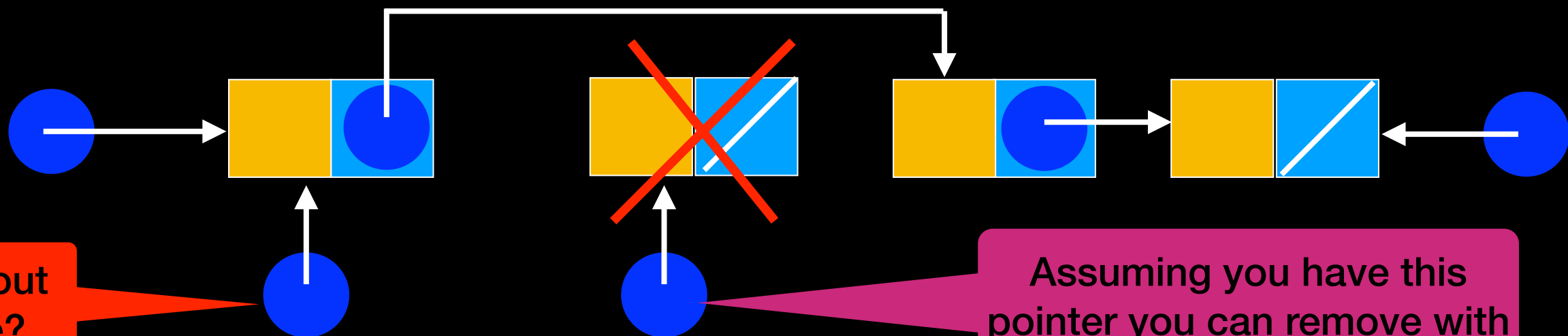


**Propose a solution to this problem:**

In English write a few sentences describing the changes you would make to the Linked-Chain implementation of the List ADT to remove from the middle

**REMOVE**

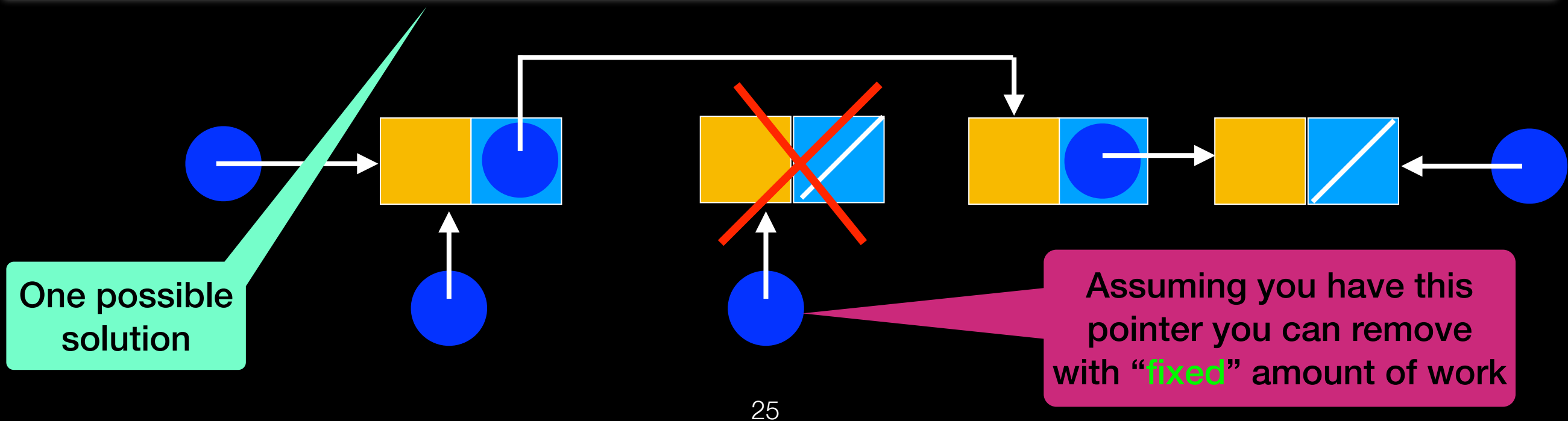
```
void remove(size_t position);
```





## REMOVE

```
void remove(size_t position);  
void getPointerTo(size_t position, Node<ItemType>*& pos_ptr, Node<ItemType>*& prev_ptr);
```



Another Solution?

# Doubly Linked List

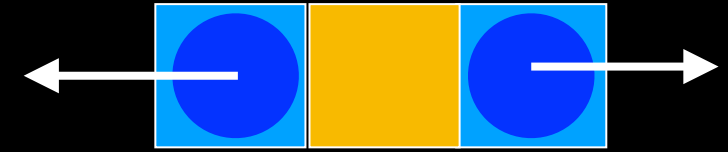
```

#ifndef NODE_H_
#define NODE_H_
template<class ItemType> class Node {

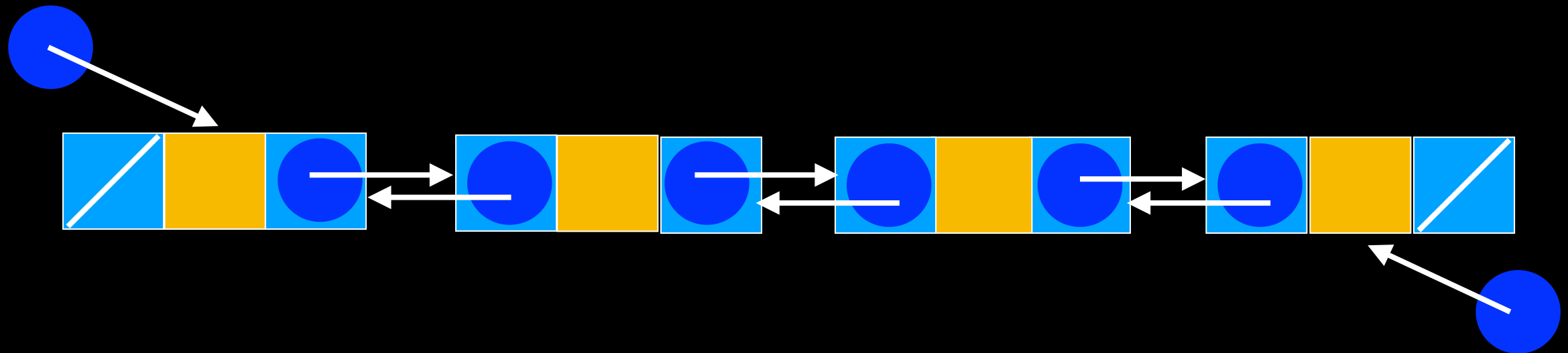
public:
    Node();
    Node(const ItemType& an_item);
    Node(const ItemType& an_item, Node<ItemType>* next_node_ptr);
    void setItem(const ItemType& an_item);
    void setNext(Node<ItemType>* next_node_ptr);
    void setPrevious(Node<ItemType>* prev_node_ptr);
    ItemType getItem() const;
    Node<ItemType>* getNext() const;
    Node<ItemType>* getPrevious() const;

private:
    ItemType item;           // A data item
    Node<ItemType>* next_;    // Pointer to next node
    Node<ItemType>* previous_; // Pointer to previous node
}; // end Node#include "Node.cpp"
#endif // NODE_H_

```



# Doubly Linked List



```

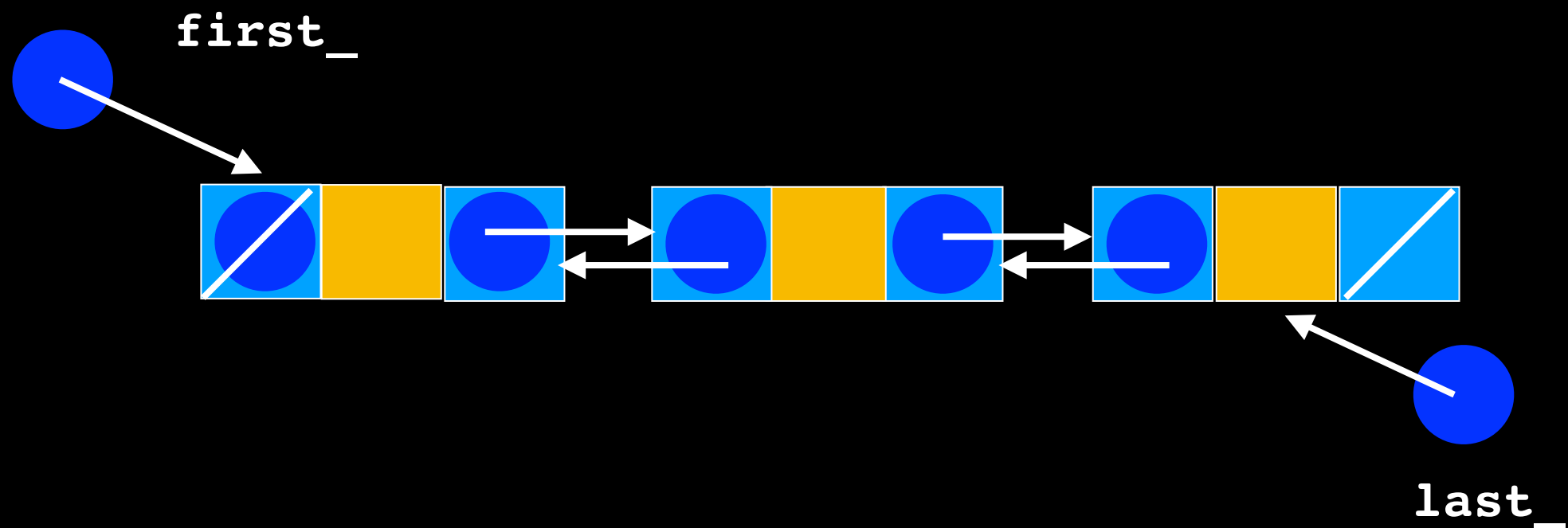
#ifndef LIST_H_
#define LIST_H_
template<typename ItemType> class List {
public:
    List(); // constructor
    List(const List<ItemType>& a_list); // copy constructor
    ~List(); // destructor
    bool isEmpty() const;
    size_t getLength() const;
    bool insert(size_t position, const ItemType& new_element);
    //retains list order, position is 0 to n-1, if position > n-1
    //it inserts at end
    bool remove(size_t position); //retains list order
    ItemType getItem(size_t position) const;
    void clear();

private:
    Node<ItemType>* first_; // Pointer to first node
    Node<ItemType>* last_; // Pointer to last node
    size_t item_count; // number of items in the list
    Node<ItemType>* getPointerTo(size_t position) const;
}; // end List
#include "List.cpp"
#endif // LIST_H_

```

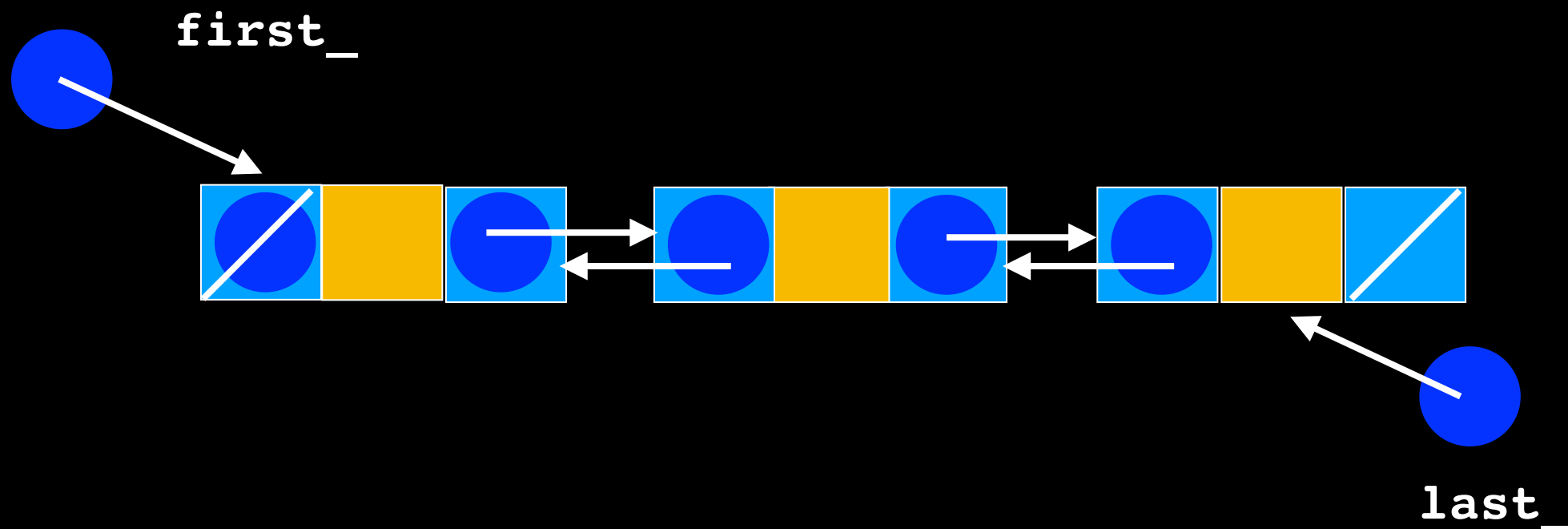
# List::insert

What are the different cases that should be considered?



# Lecture Activity

Write **Pseudocode** to insert a node at position 2 in a doubly-linked list (assume position follows classic indexing from 0 to `item_count - 1`)





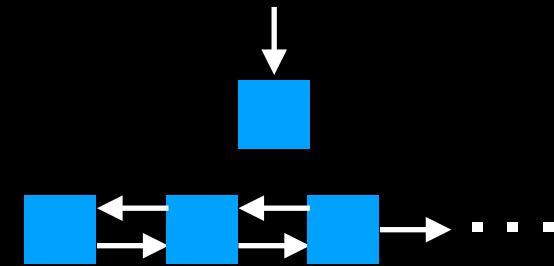
# Pseudocode

*Instantiate new node*

*Obtain pointer*

*Connect new node to chain*

*Reconnect the relevant nodes*

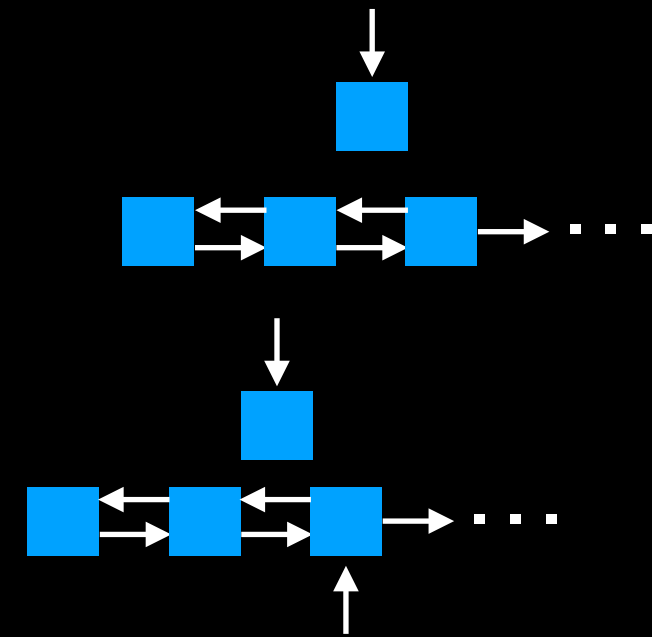


# Pseudocode

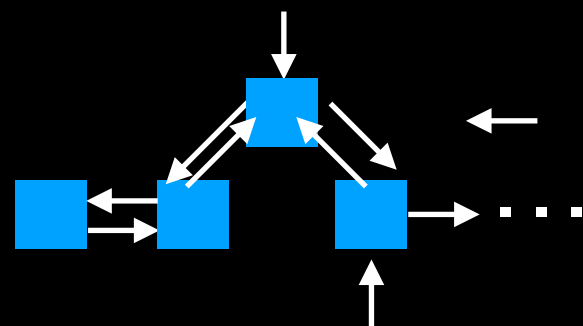
*Instantiate new node*

*Obtain pointer*

*Connect new node to chain*



*Reconnect the relevant nodes*



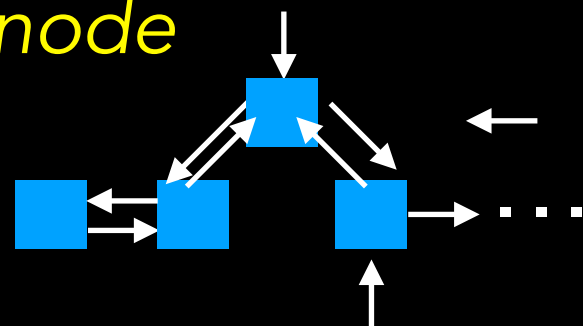
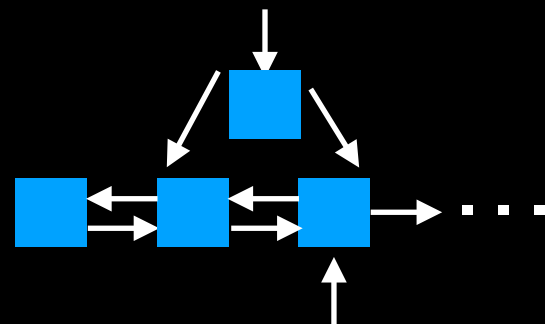
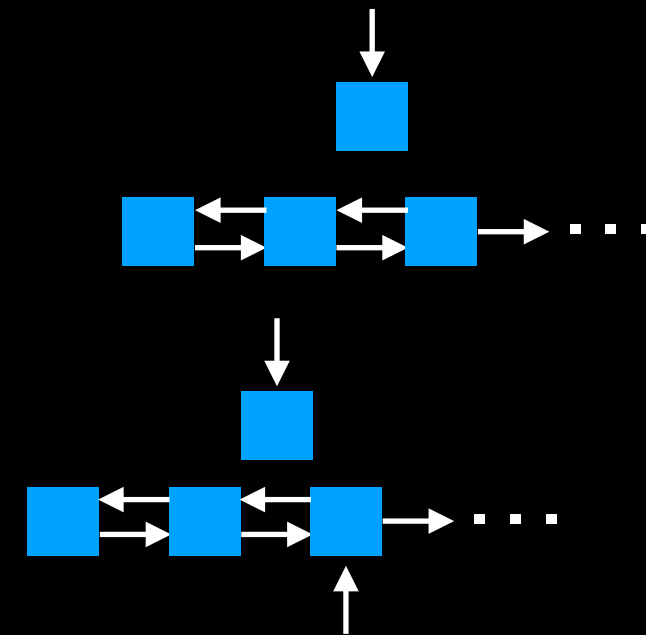
# Pseudocode

*Instantiate new node* to be inserted and set its value

*Obtain pointer* to node currently at position 2

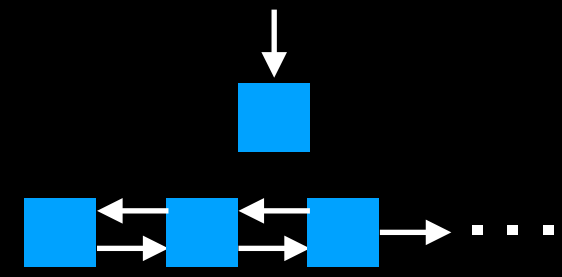
*Connect new node to chain* by pointing *its next pointer* to the node currently at *position* and *its previous pointer* to the node at *position-1*

*Reconnect the relevant nodes* in the chain by pointing *position-1->previous->next* to the *new node* and *position->previous* to the *new node*



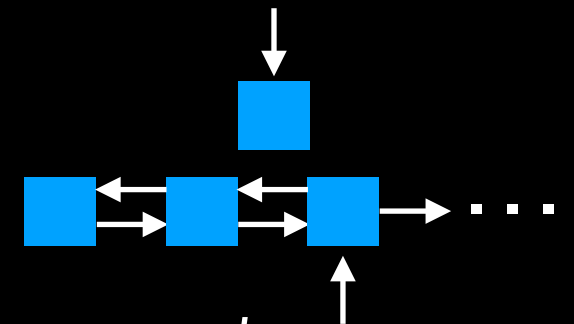
**Order Matters!**

# More Pseudocodey

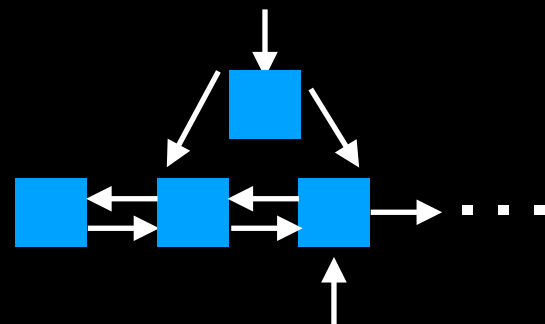


*Instantiate new node*  $new\_ptr = new\ Node()$  and  $new\_ptr \rightarrow setItem()$

*Obtain pointer*  $position\_ptr = getPointerTo(2)$



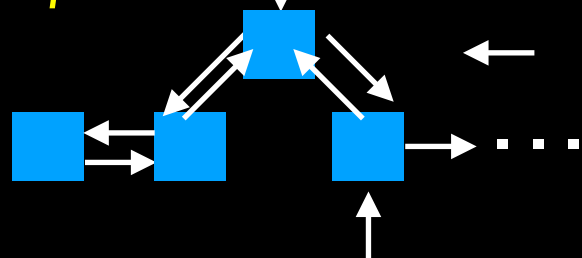
*Connect new node to chain*  $new\_ptr \rightarrow next = position\_ptr$  and  
 $new\_ptr \rightarrow previous = temp \rightarrow previous$



*Reconnect the relevant nodes*

$position\_ptr \rightarrow previous \rightarrow next = new\_ptr$  and

$position \rightarrow previous = new\_ptr$



# List::insert

```
template<typename ItemType>
bool List<ItemType>::insert(size_t position, const ItemType& new_element) {
//Create a new node containing the new entry and get a pointer to position
Node<ItemType>* new_node_ptr = new Node<ItemType>(new_element);
Node<ItemType>* pos_ptr = getPointerTo(position);
// Attach new node to chain

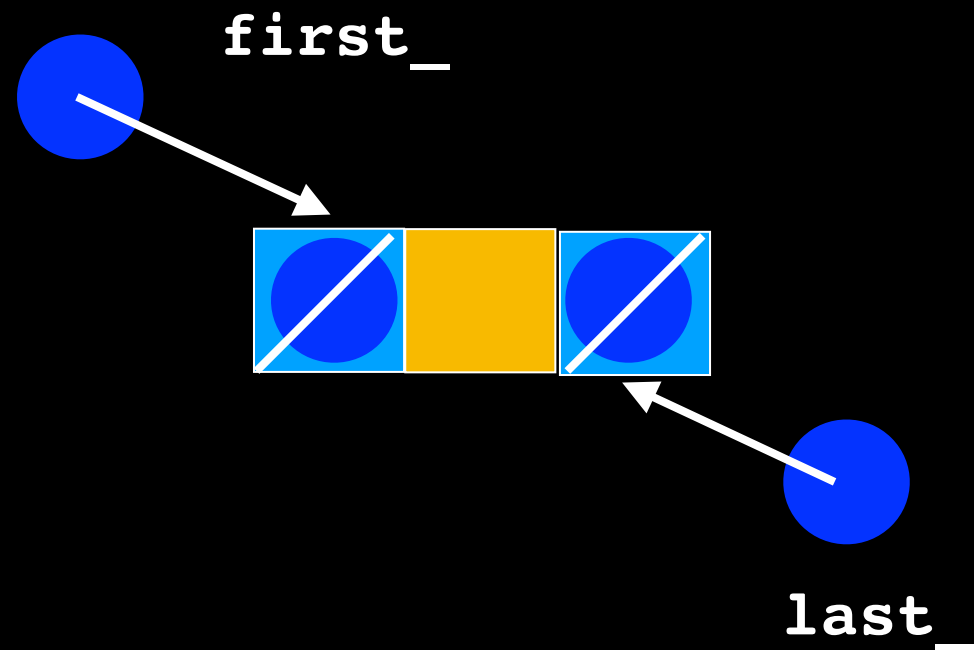
else if (pos_ptr == first_) {
    // Insert new node at head of chain
    new_node_ptr->setNext(first_);
    new_node_ptr->setPrevious(nullptr);
    first_>setPrevious(new_node_ptr);
    first_ = new_node_ptr;
}
else if (pos_ptr == nullptr) {
    //insert at end of list
    new_node_ptr->setNext(nullptr);
    new_node_ptr->setPrevious(last_);
    last_>setNext(new_node_ptr);
    last_ = new_node_ptr;
}
else {
    // Insert new node before node to which position points
    new_node_ptr->setNext(pos_ptr);
    new_node_ptr->setPrevious(pos_ptr->getPrevious());
    pos_ptr->getPrevious()->setNext(new_node_ptr);
    pos_ptr->setPrevious(new_node_ptr);
} //end if

item_count++; // Increase count of entries
return true;
} //end insert
```

```
if (first_ == nullptr)
{
    // Insert first node
    new_node_ptr->setNext(nullptr);
    new_node_ptr->setPrevious(nullptr);
    first_ = new_node_ptr;
    last_ = new_node_ptr;
}
```

Always return true

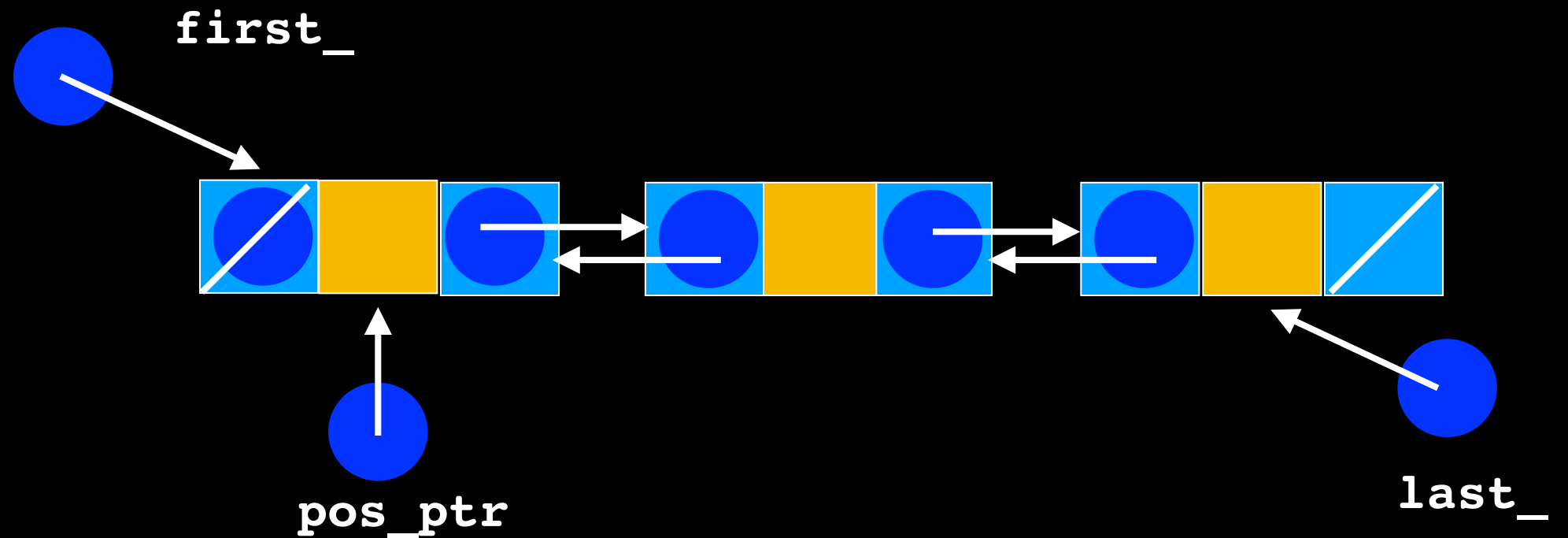
```
if (first_ == nullptr)
{
    // Insert first node
    new_node_ptr->setNext(nullptr);
    new_node_ptr->setPrevious(nullptr);
    first_ = new_node_ptr;
    last_ = new_node_ptr;
}
```



```

else if (pos_ptr == first_)
{
    // Insert new node at beginning of chain
    new_node_ptr->setNext(first_);
    new_node_ptr->setPrevious(nullptr);
    first_>setPrevious(new_node_ptr);
    first_ = new_node_ptr;
}

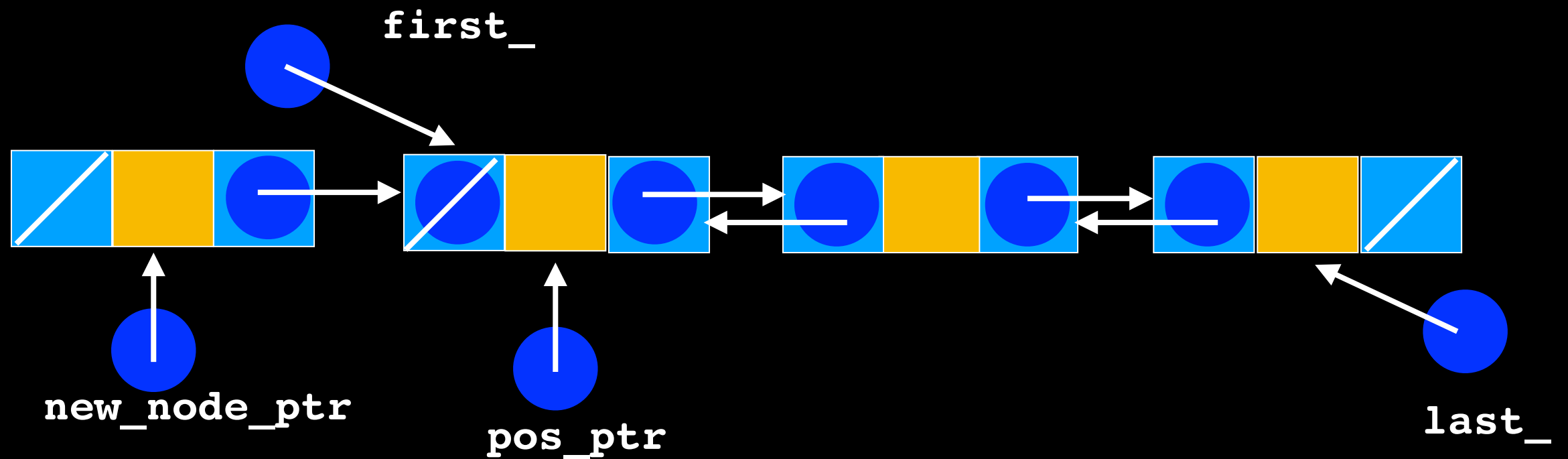
```



```

else if (pos_ptr == first_)
{
    // Insert new node at beginning of chain
    new_node_ptr->setNext(first_);
    new_node_ptr->setPrevious(nullptr);
    first_>setPrevious(new_node_ptr);
    first_ = new_node_ptr;
}

```

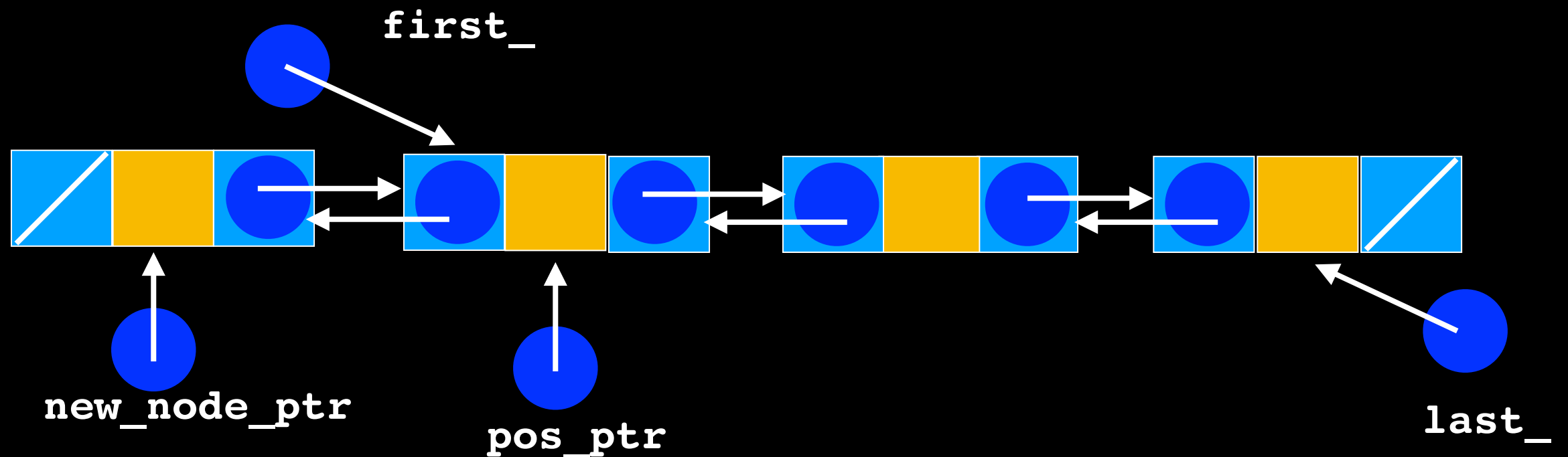




```

else if (pos_ptr == first_)
{
    // Insert new node at beginning of chain
    new_node_ptr->setNext(first_);
    new_node_ptr->setPrevious(nullptr);
    first_ ->setPrevious(new_node_ptr);
    first_ = new_node_ptr;
}

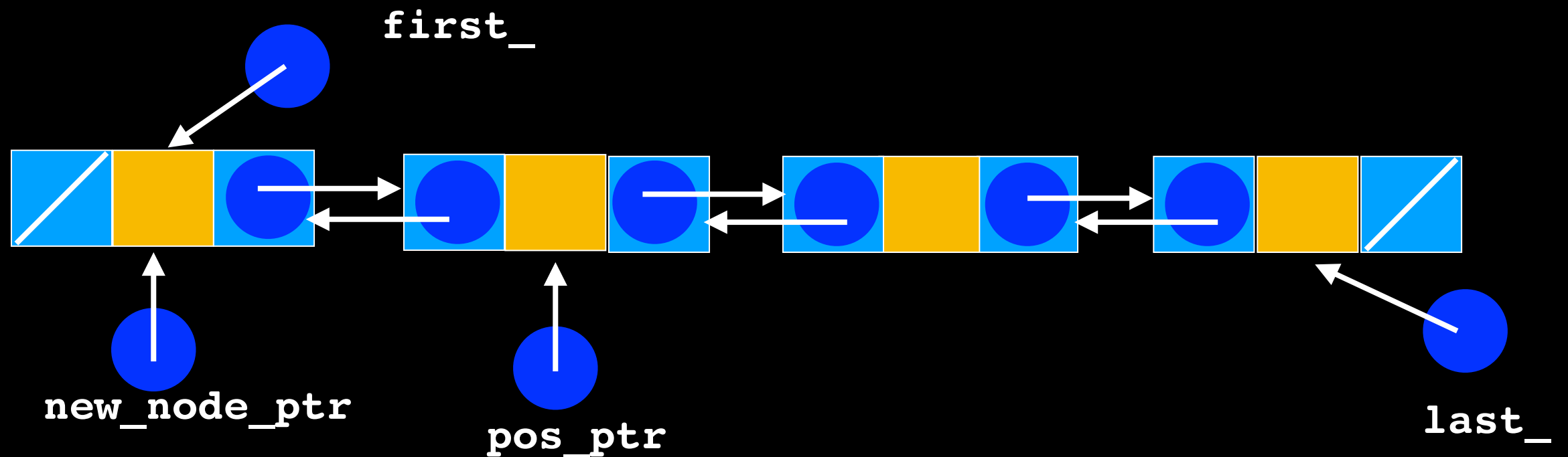
```



```

else if (pos_ptr == first_)
{
    // Insert new node at beginning of chain
    new_node_ptr->setNext(first_);
    new_node_ptr->setPrevious(nullptr);
    first_>setPrevious(new_node_ptr);
    first_ = new_node_ptr;
}

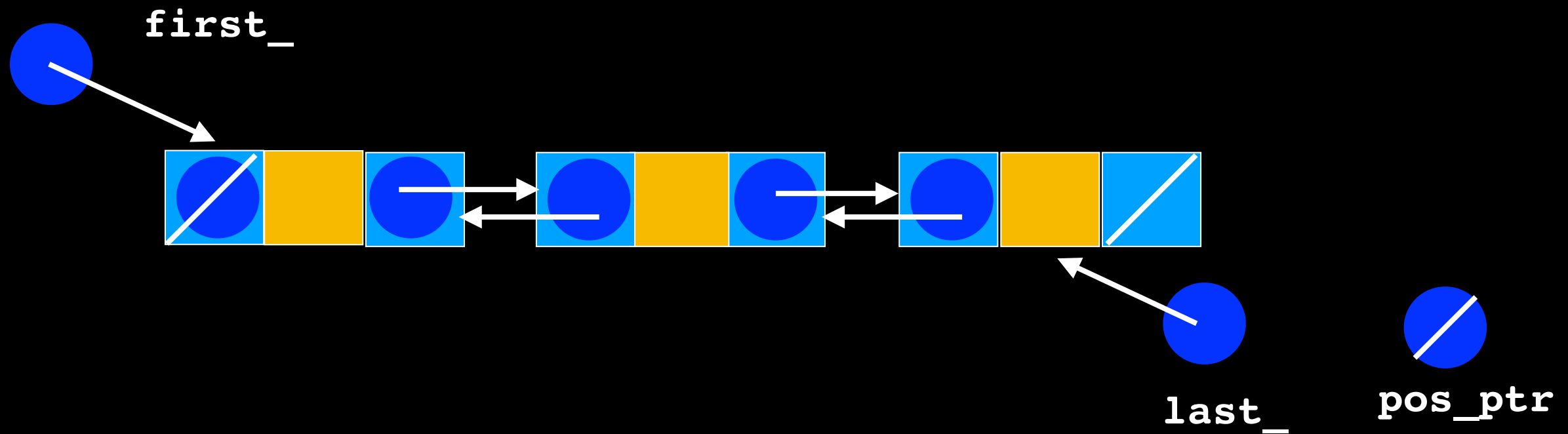
```



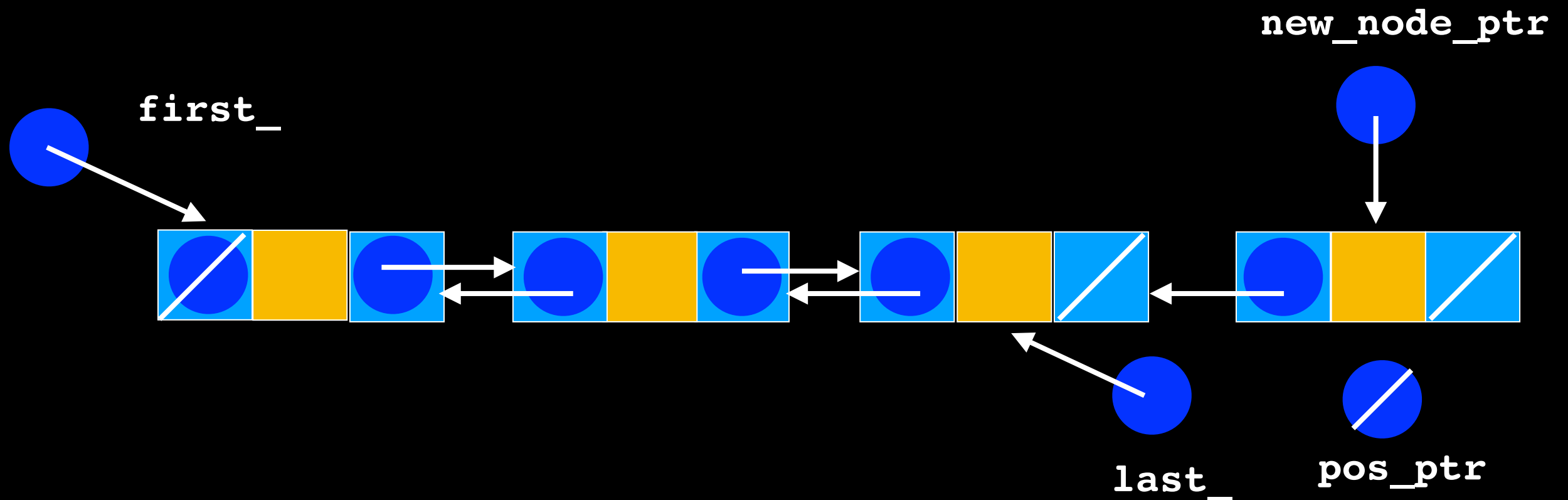
```

else if (pos_ptr == nullptr)
{
    //insert at end of list
    new_node_ptr->setNext(nullptr);
    new_node_ptr->setPrevious(last_);
    last_->setNext(new_node_ptr);
    last_ = new_node_ptr;
}

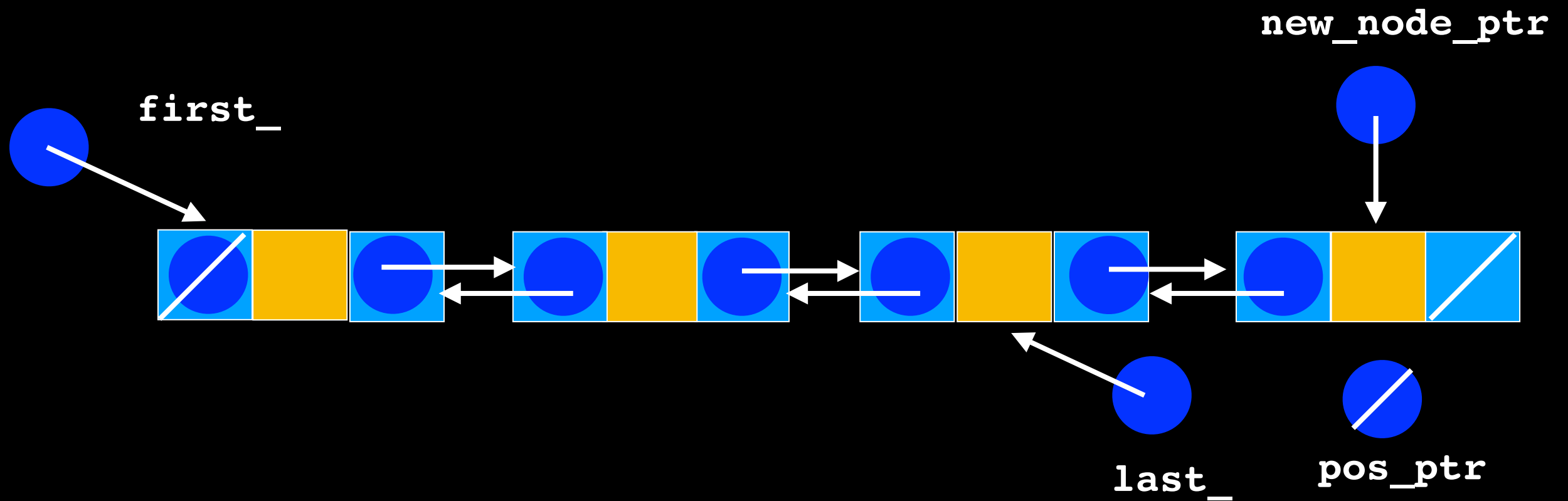
```



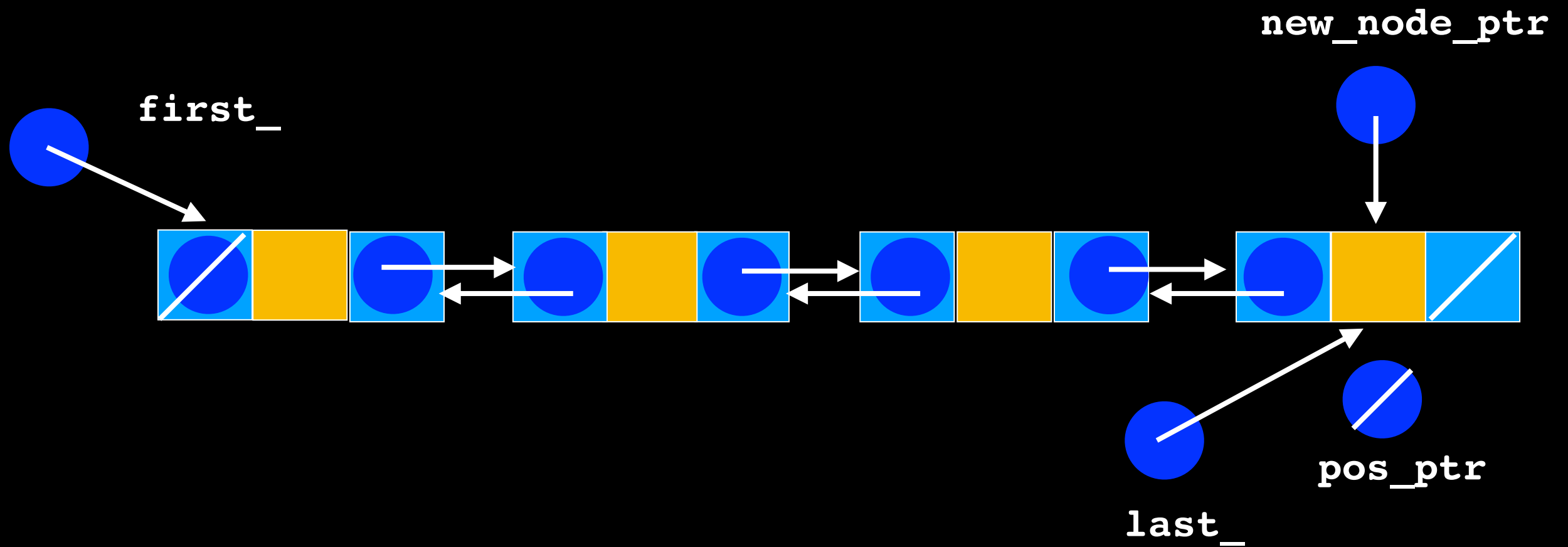
```
else if (pos_ptr == nullptr)
{
    //insert at end of list
    new_node_ptr->setNext(nullptr);
    new_node_ptr->setPrevious(last_);
    last_->setNext(new_node_ptr);
    last_ = new_node_ptr;
}
```



```
else if (pos_ptr == nullptr)
{
    //insert at end of list
    new_node_ptr->setNext(nullptr);
    new_node_ptr->setPrevious(last_);
    last_->setNext(new_node_ptr);
    last_ = new_node_ptr;
}
```



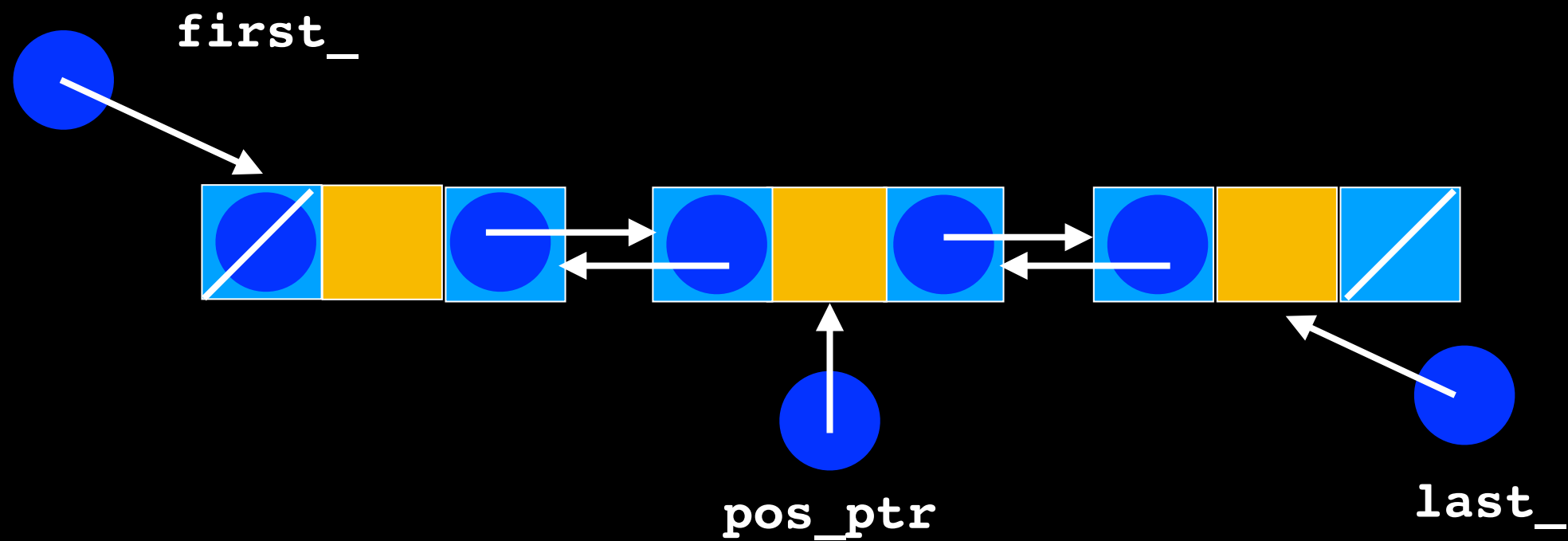
```
else if (pos_ptr == nullptr)
{
    //insert at end of list
    new_node_ptr->setNext(nullptr);
    new_node_ptr->setPrevious(last_);
    last_->setNext(new_node_ptr);
    last_ = new_node_ptr;
}
```



```

else
{
    // Insert new node before node to which position points
    new_node_ptr->setNext(pos_ptr);
    new_node_ptr->setPrevious(pos_ptr->getPrevious());
    pos_ptr->getPrevious()->setNext(new_node_ptr);
    pos_ptr->setPrevious(new_node_ptr);
}
// end if

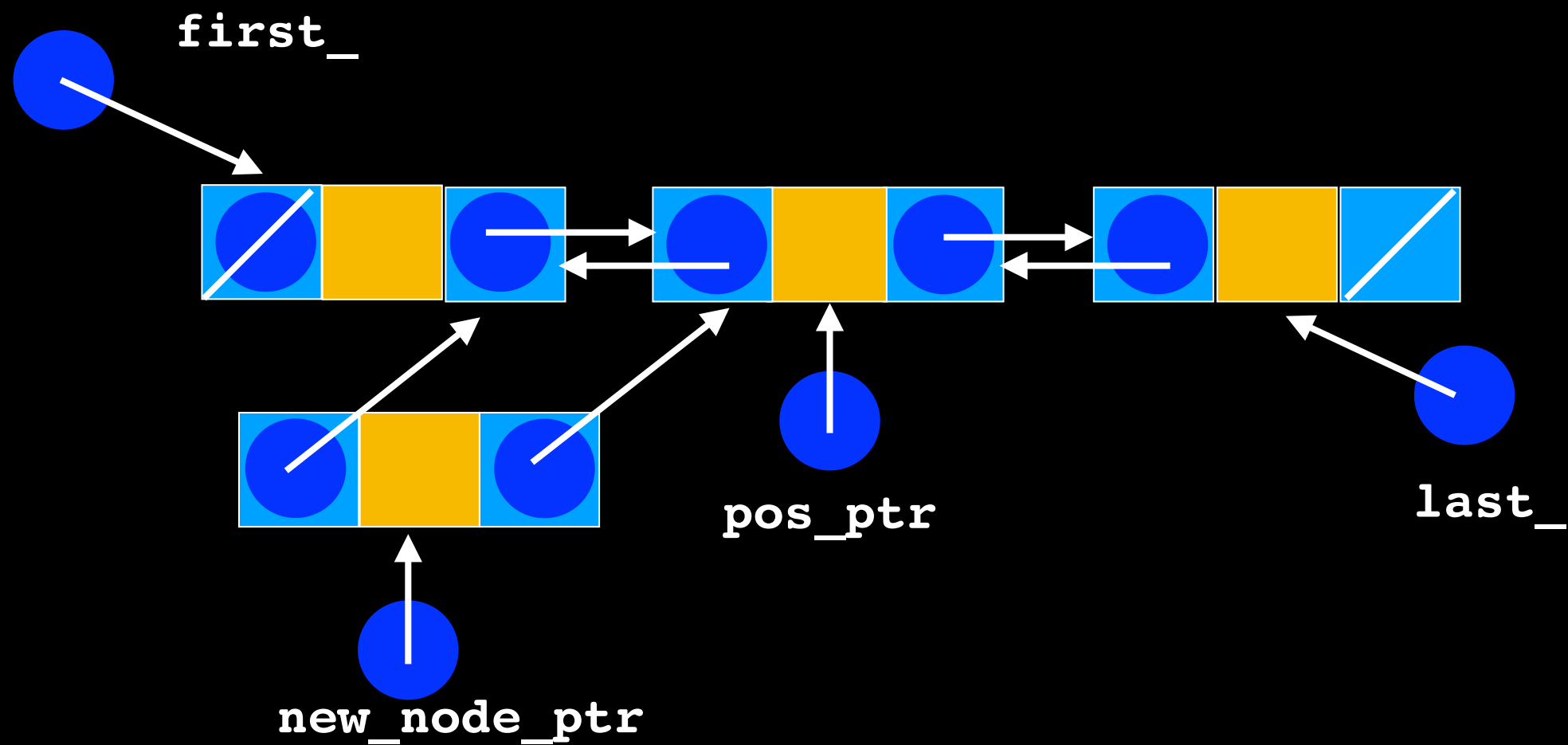
```



```

else
{
    // Insert new node before node to which position points
    new_node_ptr->setNext(pos_ptr);
    new_node_ptr->setPrevious(pos_ptr->getPrevious());
    pos_ptr->getPrevious()->setNext(new_node_ptr);
    pos_ptr->setPrevious(new_node_ptr);
}
// end if

```

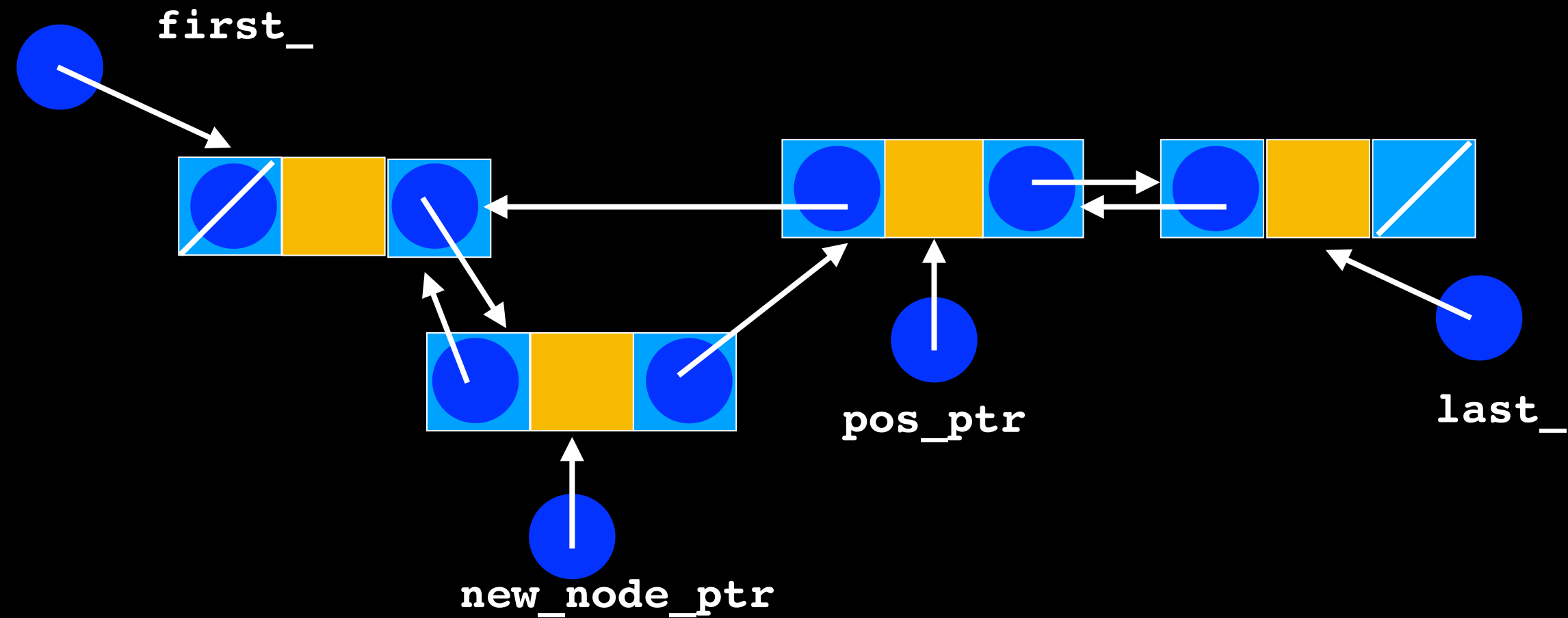




```

else
{
    // Insert new node before node to which position points
    new_node_ptr->setNext(pos_ptr);
    new_node_ptr->setPrevious(pos_ptr->getPrevious());
    pos_ptr->getPrevious()->setNext(new_node_ptr);
    pos_ptr->setPrevious(new_node_ptr);
} // end if

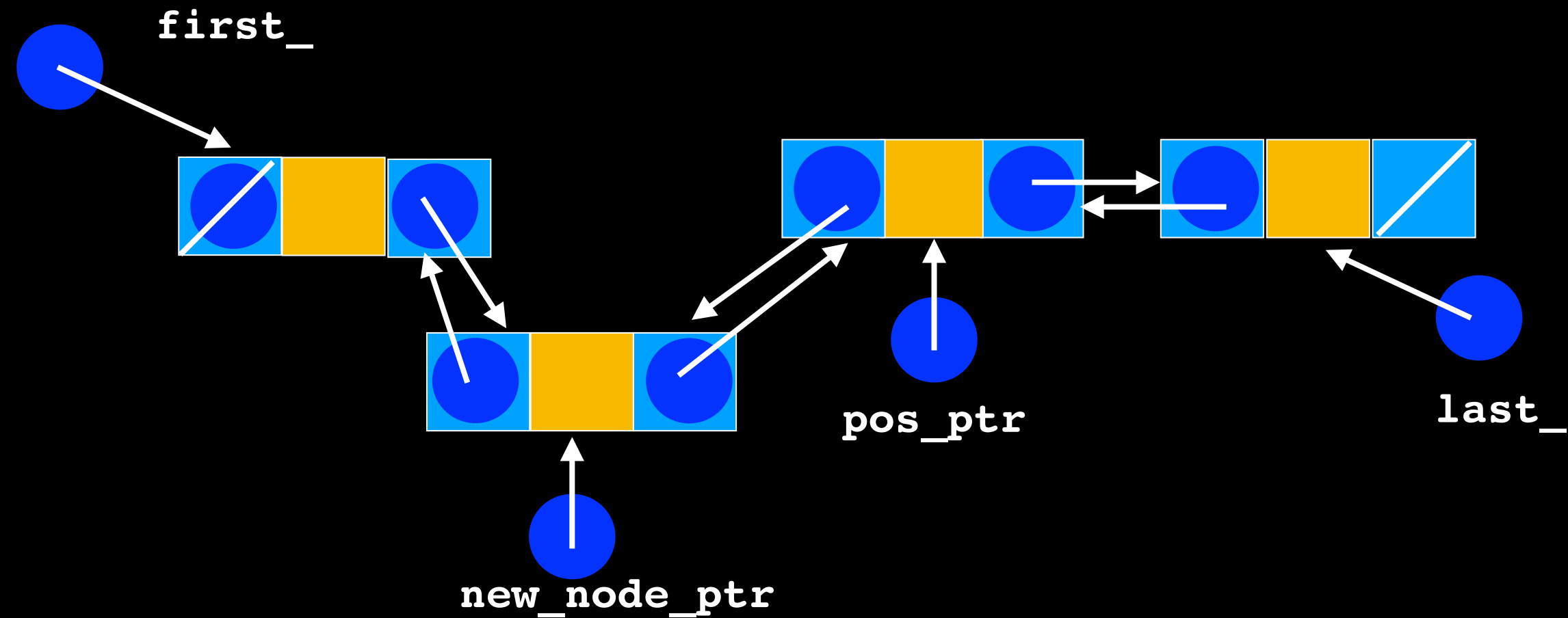
```



```

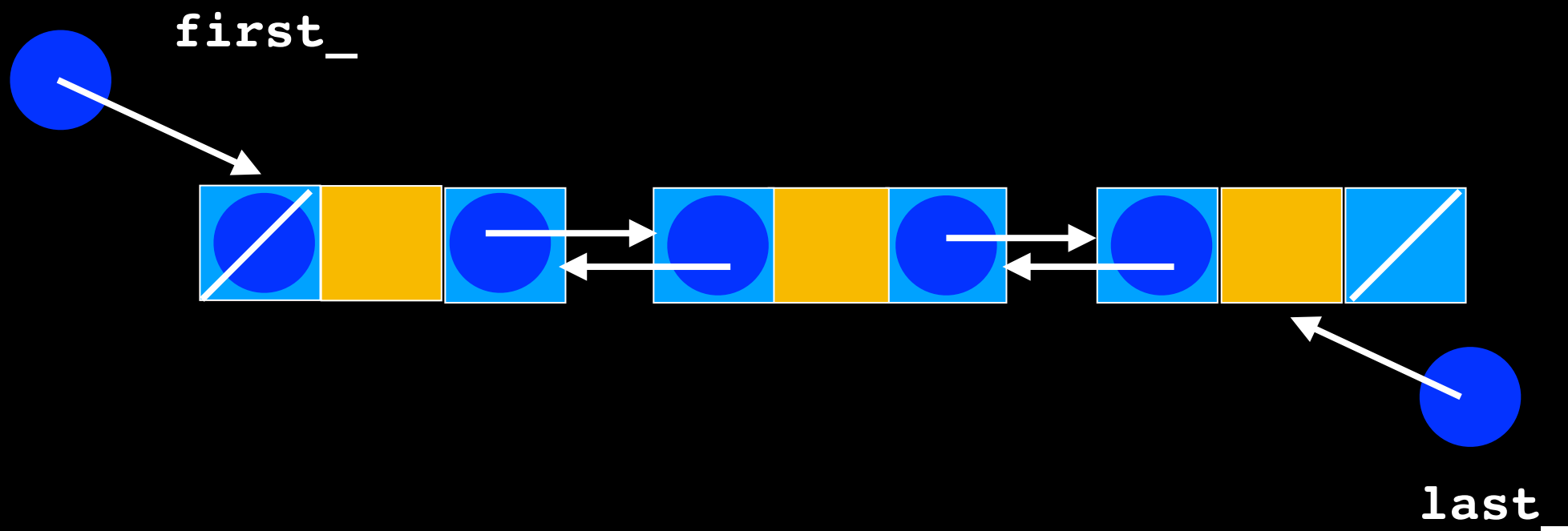
else
{
    // Insert new node before node to which position points
    new_node_ptr->setNext(pos_ptr);
    new_node_ptr->setPrevious(pos_ptr->getPrevious());
    pos_ptr->getPrevious()->setNext(new_node_ptr);
    pos_ptr->setPrevious(new_node_ptr);
}
// end if

```



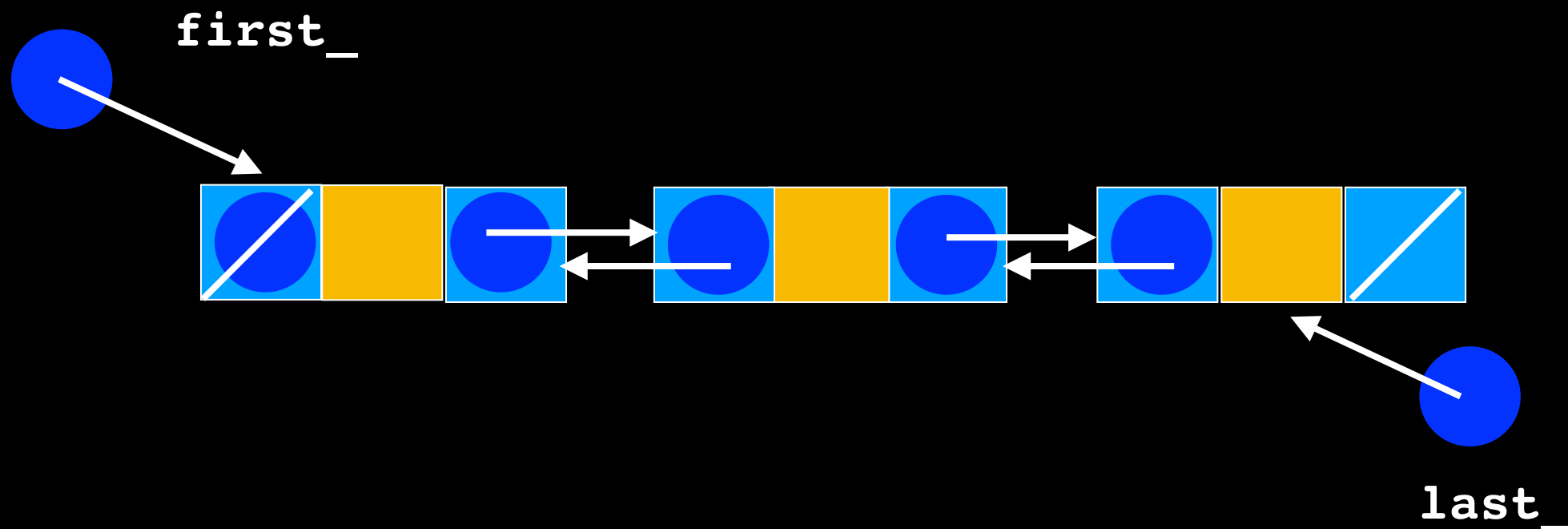
# List::remove

What are the different cases that should be considered?



# Lecture Activity

Write **Pseudocode** to remove the node at position 1 in a doubly-linked list (assume position follows classic indexing from 0 to  $\text{item\_count} - 1$ , and there is a node at position 2)



# List::Remove

```
template<typename ItemType> bool List<T>::remove(size_t position) {
    // get pointer to position
    Node<ItemType>* pos_ptr = getPointerTo(position);
    if (pos_ptr == nullptr) // no node at position
        return false;
    else {
        // Remove node from chain
        else if (pos_ptr == last_) {
            //remove last_ node
            last_ = pos_ptr->getPrevious();
            last_ ->setNext(nullptr);
            // Return node to the system
            pos_ptr->setPrevious(nullptr);
            delete pos_ptr;
            pos_ptr = nullptr;
        }
        else {
            //Remove from the middle
            pos_ptr->getPrevious()->setNext(pos_ptr->getNext());
            pos_ptr->getNext()->setPrevious(pos_ptr->getPrevious());
            // Return node to the system
            pos_ptr->setNext(nullptr);
            pos_ptr->setPrevious(nullptr);
            delete pos_ptr;
            pos_ptr = nullptr;
        }
        item_count--;
        return true;
    }
} //end remove
```

```
if (pos_ptr == first_)
{
    // Remove first node
    first_ = pos_ptr->getNext();
    first_->setPrevious(nullptr);

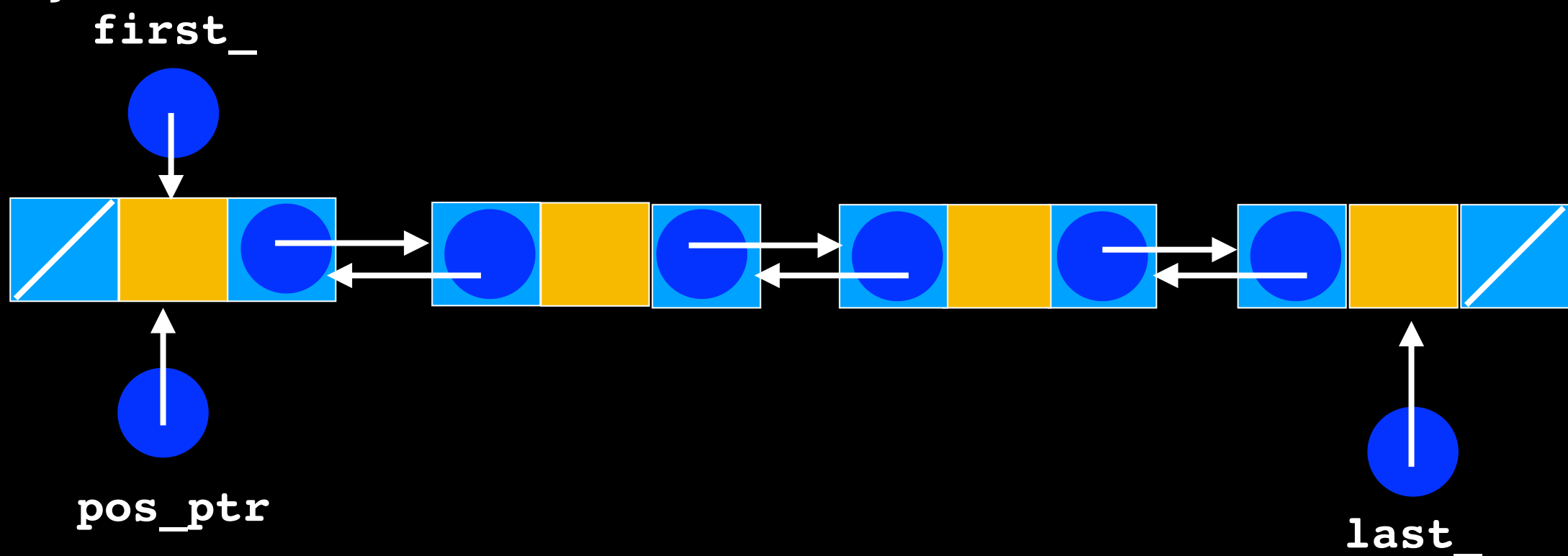
    // Return node to the system
    pos_ptr->setNext(nullptr);
    delete pos_ptr;
    pos_ptr = nullptr;
}
```

```

// Remove node from chain
if (pos_ptr == first_)
{
    // Remove first node
    first_ = pos_ptr->getNext();
    first_->setPrevious(nullptr);

    // Return node to the system
    pos_ptr->setNext(nullptr);
    delete pos_ptr;
    pos_ptr = nullptr;
}

```

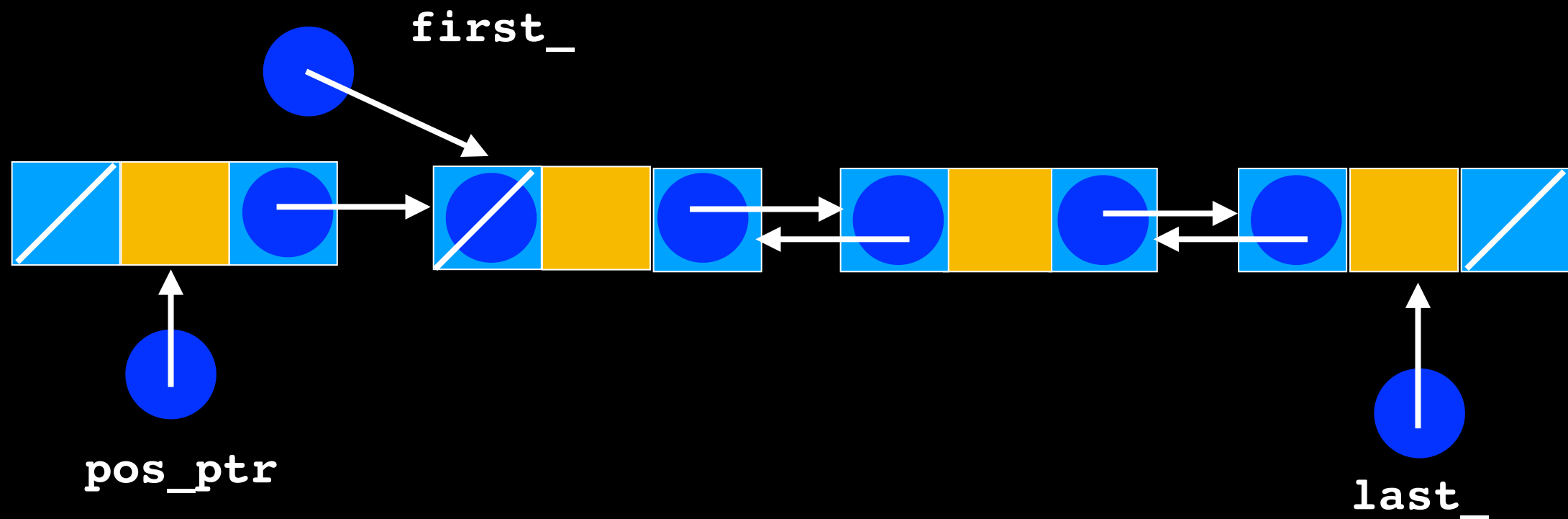


```

// Remove node from chain
if (pos_ptr == first_)
{
    // Remove first node
    first_ = pos_ptr->getNext();
    first_->setPrevious(nullptr);

    // Return node to the system
    pos_ptr->setNext(nullptr);
    delete pos_ptr;
    pos_ptr = nullptr;
}

```

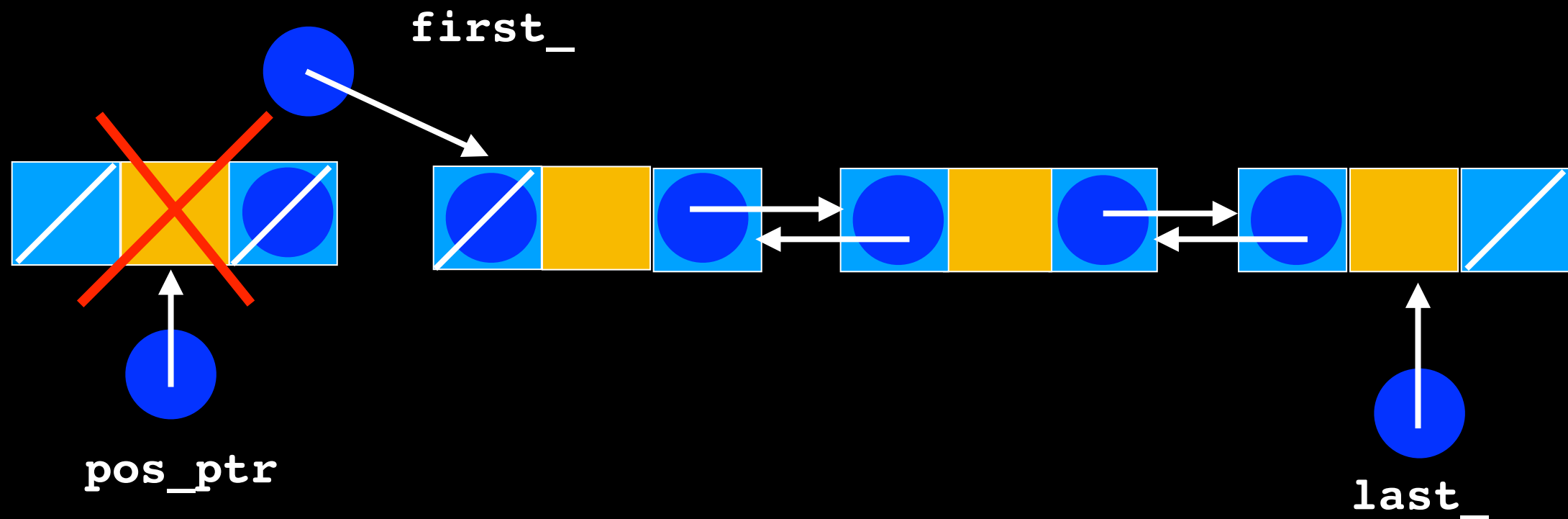


```

// Remove node from chain
if (pos_ptr == first_)
{
    // Remove first node
    first_ = pos_ptr->getNext();
    first_->setPrevious(nullptr);

    // Return node to the system
    pos_ptr->setNext(nullptr);
    delete pos_ptr;
    pos_ptr = nullptr;
}

```



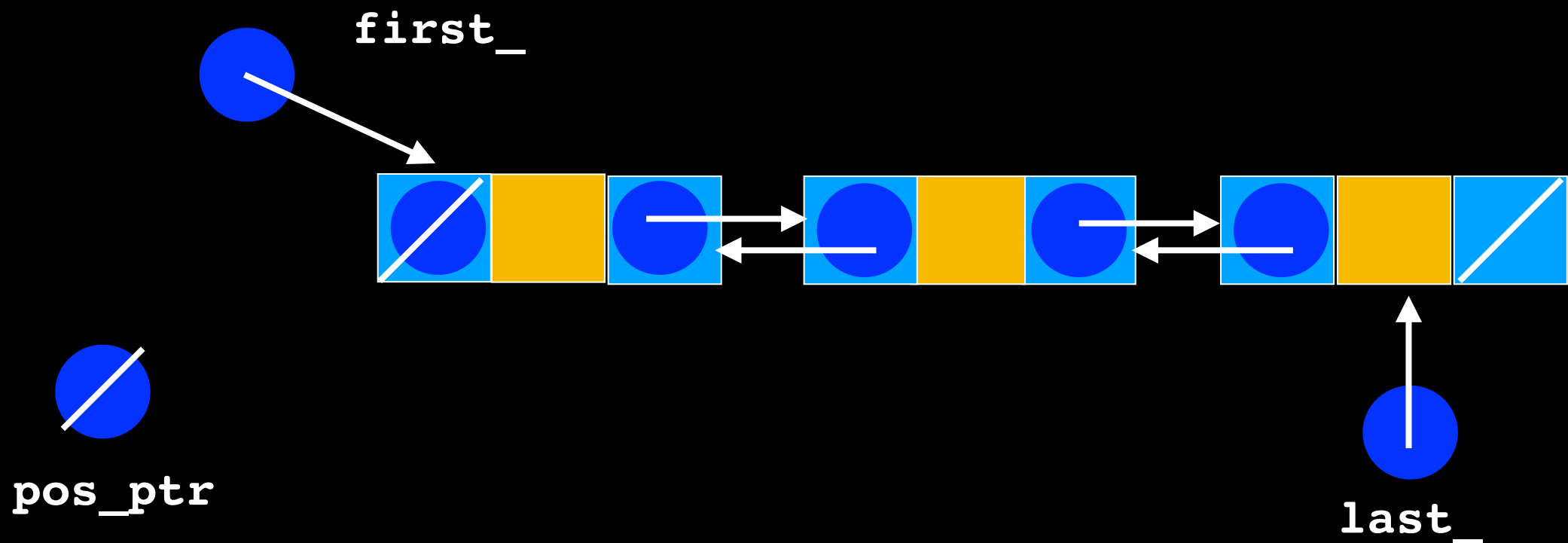


```

// Remove node from chain
if (pos_ptr == first_)
{
    // Remove first node
    first_ = pos_ptr->getNext();
    first_->setPrevious(nullptr);

    // Return node to the system
    pos_ptr->setNext(nullptr);
    delete pos_ptr;
    pos_ptr = nullptr;
}

```

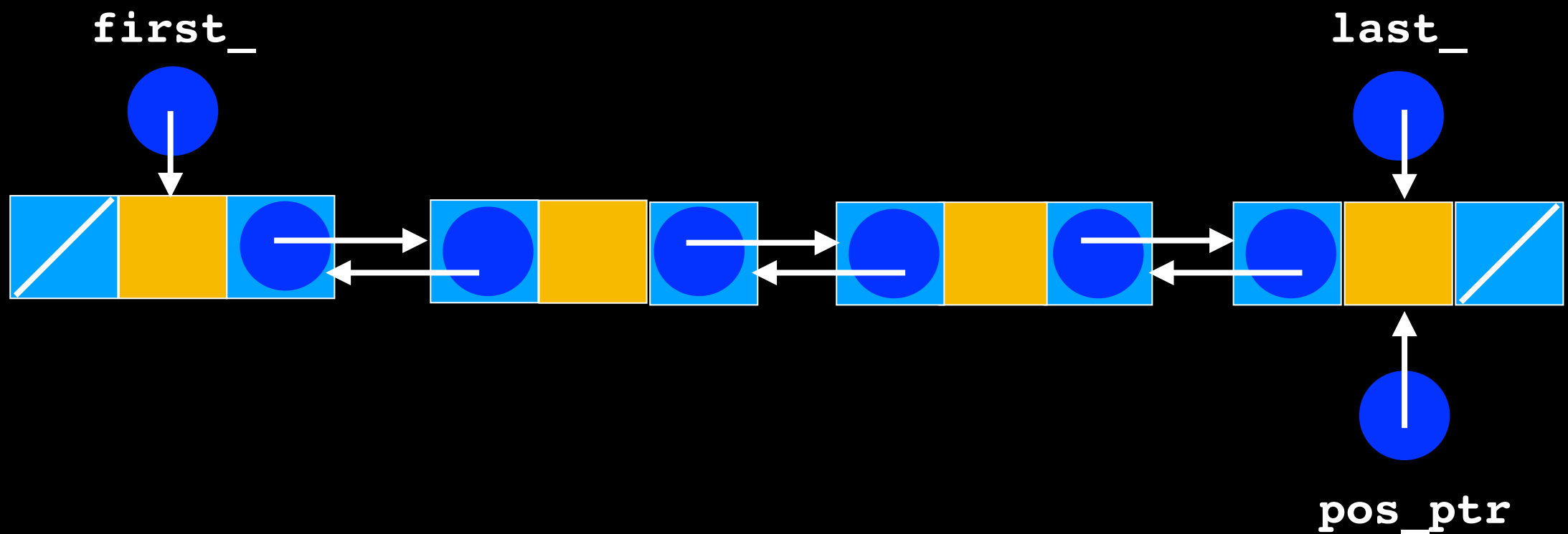


```

else if (pos_ptr == last_)
{
    //remove last_ node
    last_ = pos_ptr->getPrevious();
    last_ ->setNext(nullptr);

    // Return node to the system
    pos_ptr->setPrevious(nullptr);
    delete pos_ptr;
    pos_ptr = nullptr;
}

```

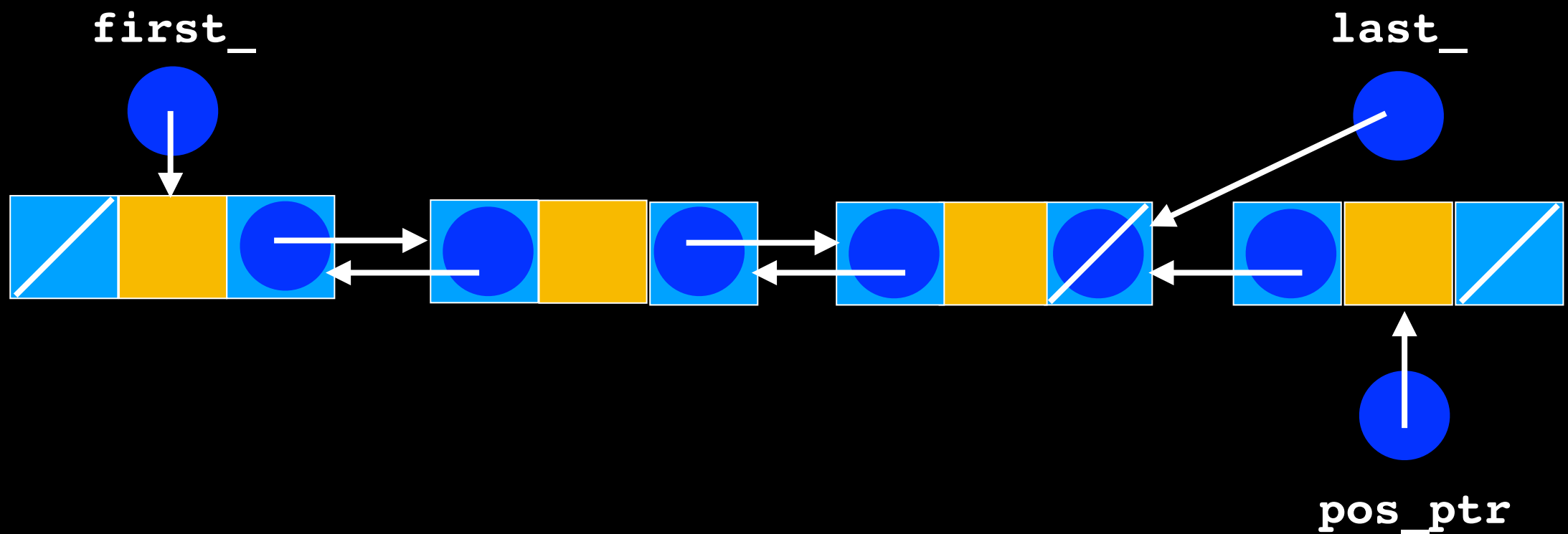


```

else if (pos_ptr == last_)
{
    //remove last_ node
    last_ = pos_ptr->getPrevious();
    last_ ->setNext(nullptr);

    // Return node to the system
    pos_ptr->setPrevious(nullptr);
    delete pos_ptr;
    pos_ptr = nullptr;
}

```

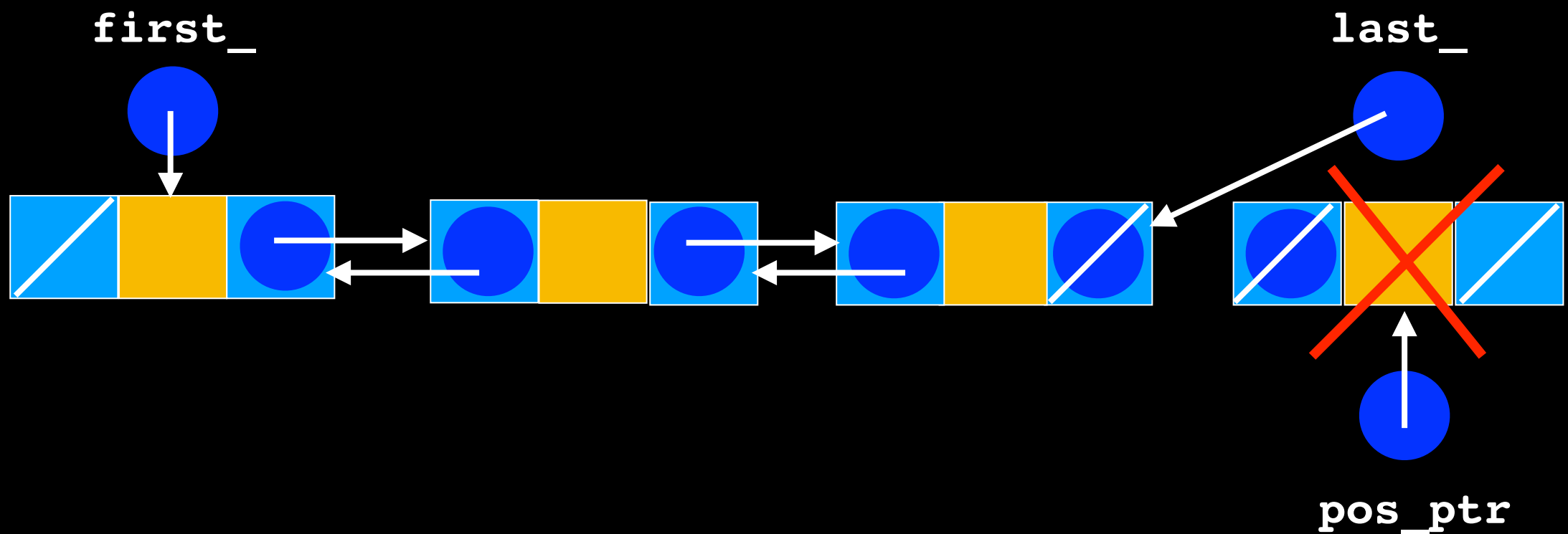


```

else if (pos_ptr == last_)
{
    //remove last_ node
    last_ = pos_ptr->getPrevious();
    last_ ->setNext(nullptr);

    // Return node to the system
    pos_ptr->setPrevious(nullptr);
    delete pos_ptr;
    pos_ptr = nullptr;
}

```

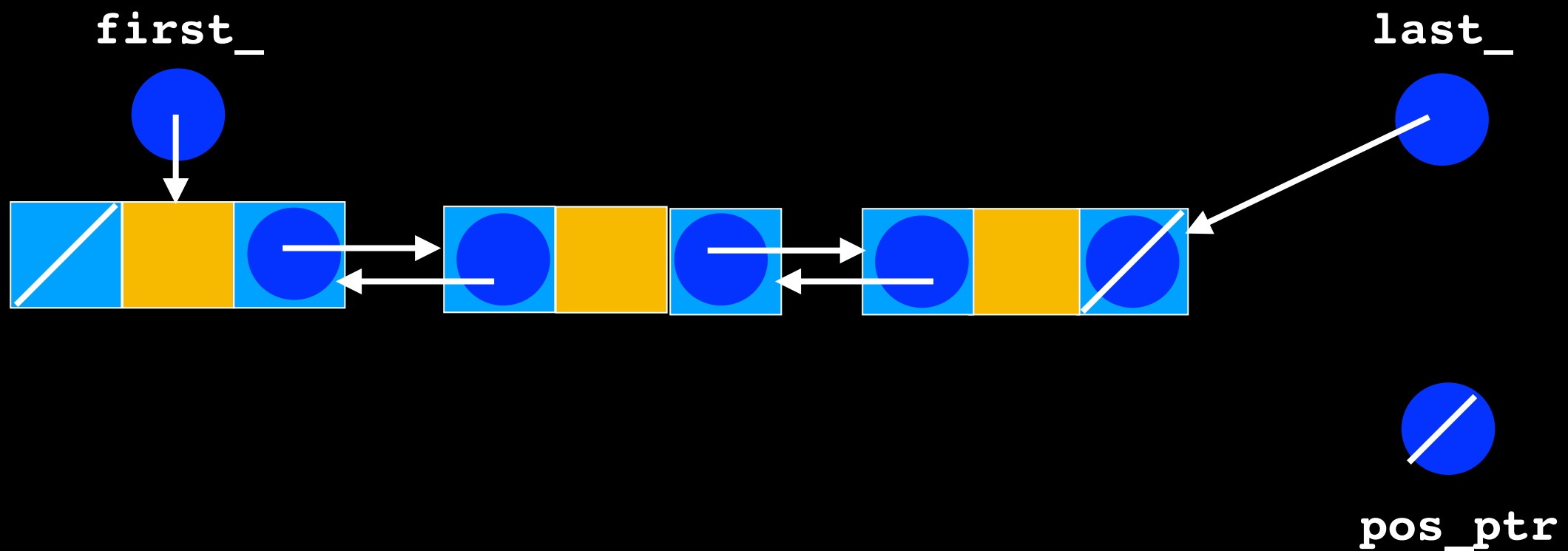


```

else if (pos_ptr == last_)
{
    //remove last_ node
    last_ = pos_ptr->getPrevious();
    last_ ->setNext(nullptr);

    // Return node to the system
    pos_ptr->setPrevious(nullptr);
    delete pos_ptr;
    pos_ptr = nullptr;
}

```

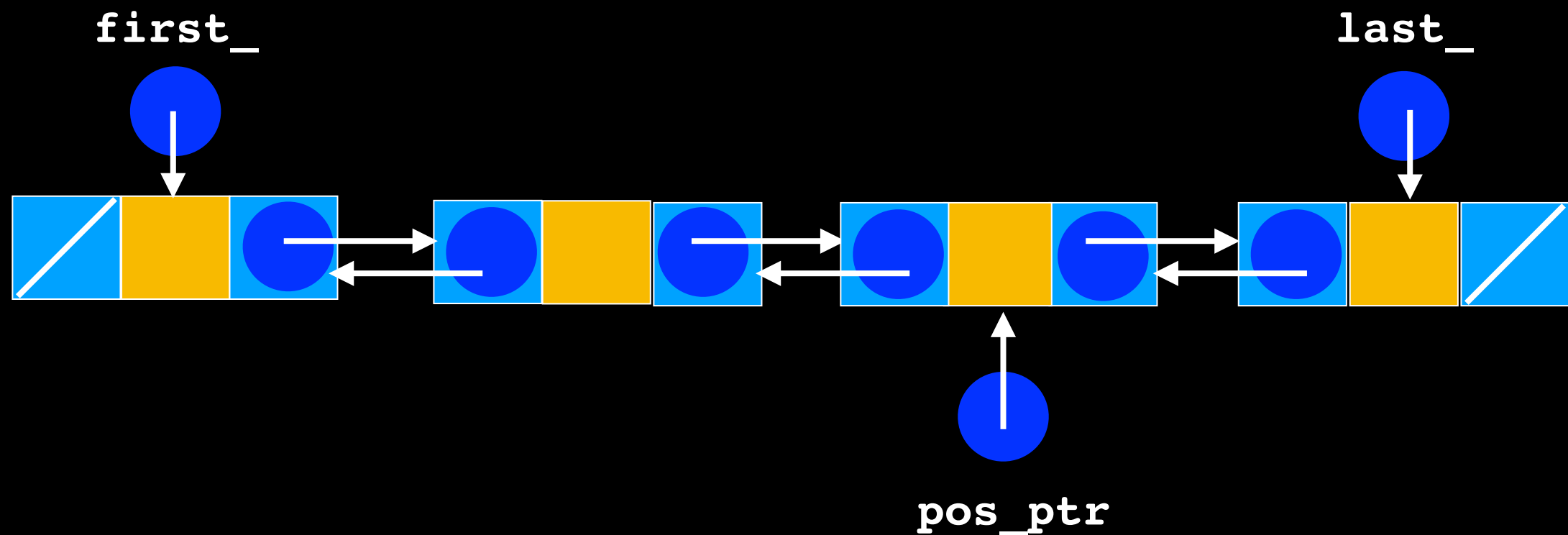


```

else if (pos_ptr != nullptr)
{
    //Remove from the middle
    pos_ptr->getPrevious()->setNext(pos_ptr->getNext());
    pos_ptr->getNext()->setPrevious(pos_ptr->getPrevious());

    // Return node to the system
    pos_ptr->setNext(nullptr);
    pos_ptr->setPrevious(nullptr);
    delete pos_ptr;
    pos_ptr = nullptr;
} // end if

```

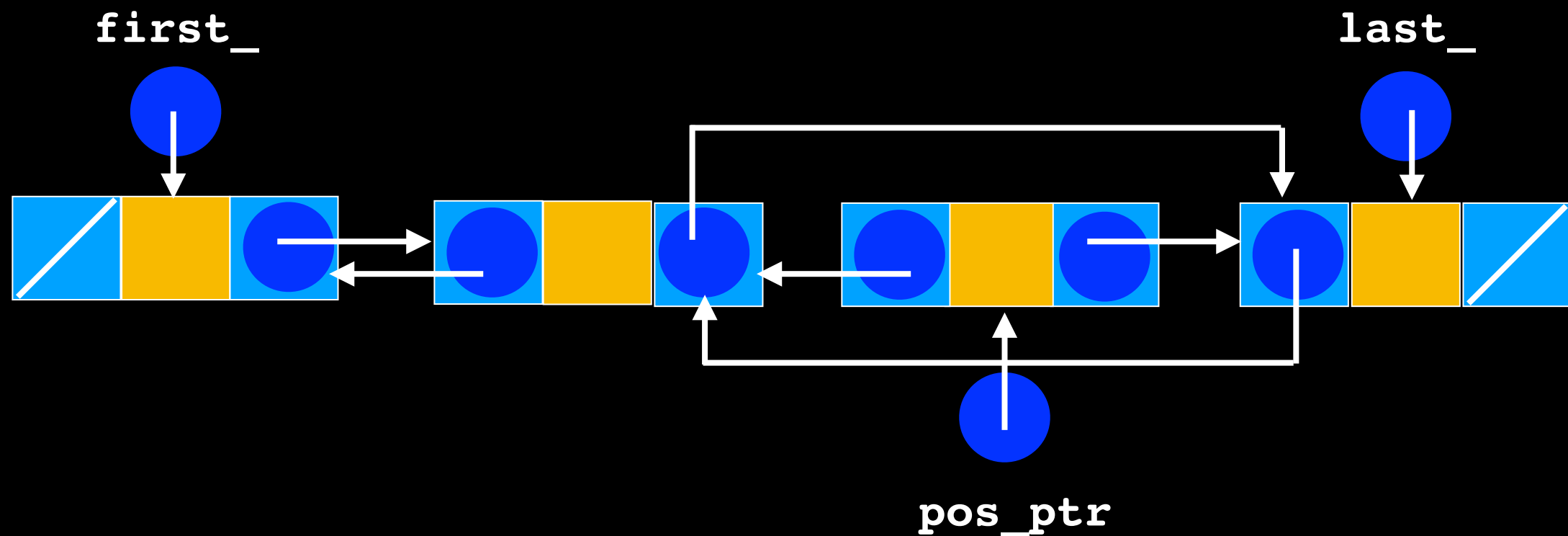


```

else if (pos_ptr != nullptr)
{
    //Remove from the middle
    pos_ptr->getPrevious()->setNext(pos_ptr->getNext());
    pos_ptr->getNext()->setPrevious(pos_ptr->getPrevious());

    // Return node to the system
    pos_ptr->setNext(nullptr);
    pos_ptr->setPrevious(nullptr);
    delete pos_ptr;
    pos_ptr = nullptr;
} // end if

```

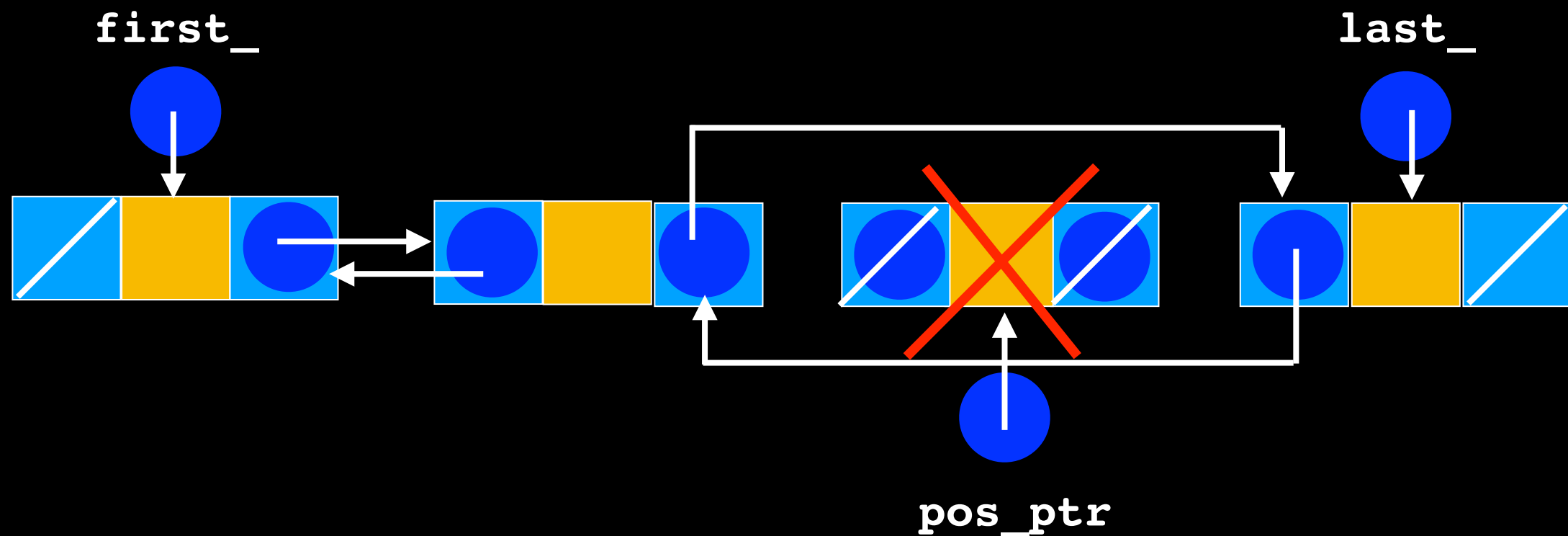


```

else if (pos_ptr != nullptr)
{
    //Remove from the middle
    pos_ptr->getPrevious()->setNext(pos_ptr->getNext());
    pos_ptr->getNext()->setPrevious(pos_ptr->getPrevious());

    // Return node to the system
    pos_ptr->setNext(nullptr);
    pos_ptr->setPrevious(nullptr);
    delete pos_ptr;
    pos_ptr = nullptr;
}
// end if

```



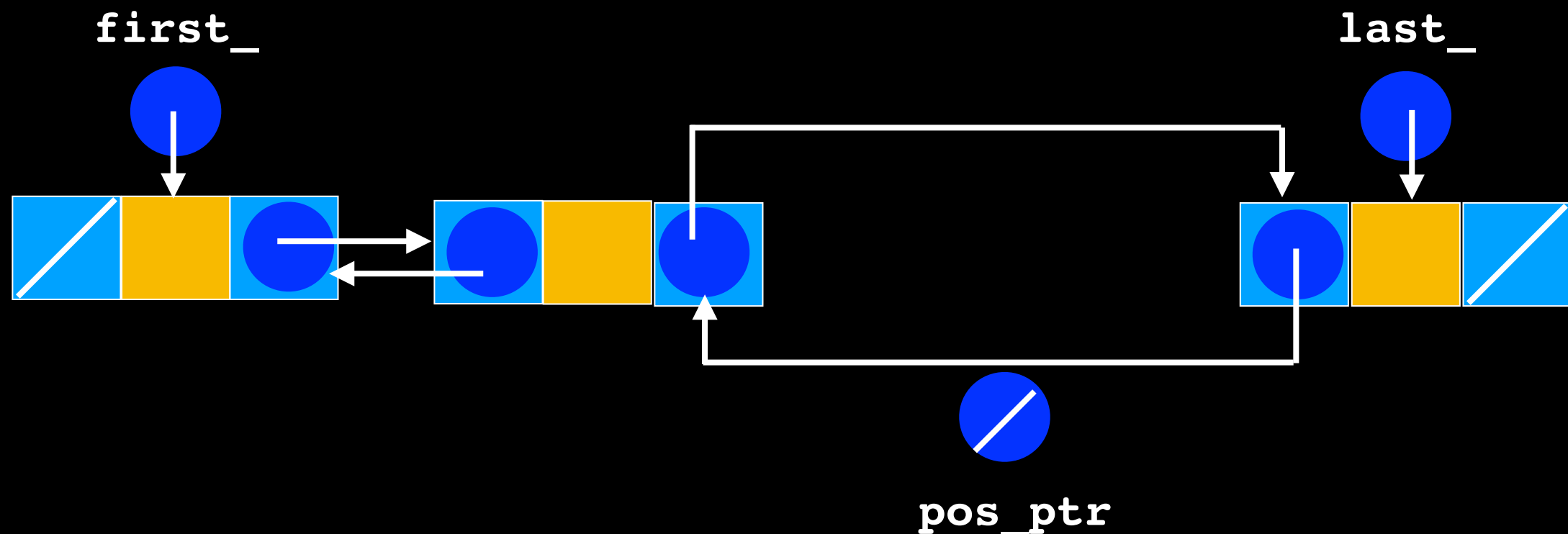


```

else if (pos_ptr != nullptr)
{
    //Remove from the middle
    pos_ptr->getPrevious()->setNext(pos_ptr->getNext());
    pos_ptr->getNext()->setPrevious(pos_ptr->getPrevious());

    // Return node to the system
    pos_ptr->setNext(nullptr);
    pos_ptr->setPrevious(nullptr);
    delete pos_ptr;
    pos_ptr = nullptr;
}
// end if

```



# List::getPointerTo

```
template<typename ItemType>
Node<ItemType>* List<ItemType>::getPointerTo(size_t position) const
{
    Node<ItemType>* find_ptr = nullptr;
    // return nullptr if there is no node at position

    if(position < item_count)
    { //there is a node at position
        find_ptr = first_;
        for(size_t i = 0; i < position; ++i)
        {
            find_ptr = find_ptr->getNext();
        }
        //find_ptr points to the node at position
    }

    return find_ptr;
} //end getPointerTo
```

# List::getItem

```
template<typename ItemType>
ItemType List<ItemType>::getItem(size_t position) const
{
    Node<ItemType>* pos_ptr = getPointerTo(position);
    if(pos_ptr != nullptr)
        return pos_ptr->getItem();
    else
        ???
}
```

# List::getItem

```
template<typename ItemType>
ItemType List<ItemType>::getItem(size_t position) const
{
    Node<ItemType>* pos_ptr = getPointerTo(position);
    if(pos_ptr != nullptr)
        return pos_ptr->getItem();
    else
        ???
}
```

**Problem:** return type is  
ItemType  
There is no “default” or null  
value to indicate  
uninitialized object