Lists

Tiziana Ligorio
tligorio@hunter.cuny.edu

Today’s Plan

Lists

Announcements

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<class T>
class List
{

public:
    List(); // constructor
    List(const List<T>& a_list); // copy constructor
    ~List(); // destructor
    bool isEmpty() const;
    size_t getLength() const;

    //retains list order, position is 0 to n-1, if position > n-1 it inserts at end
    bool insert(size_t position, const T& new_element);
    bool remove(size_t position);//retains list order
    T getItem(size_t position) const;
    void clear();


private:
    //implementation details here


}; // end List

#include "List.cpp"
#endif // LIST_H_
```

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

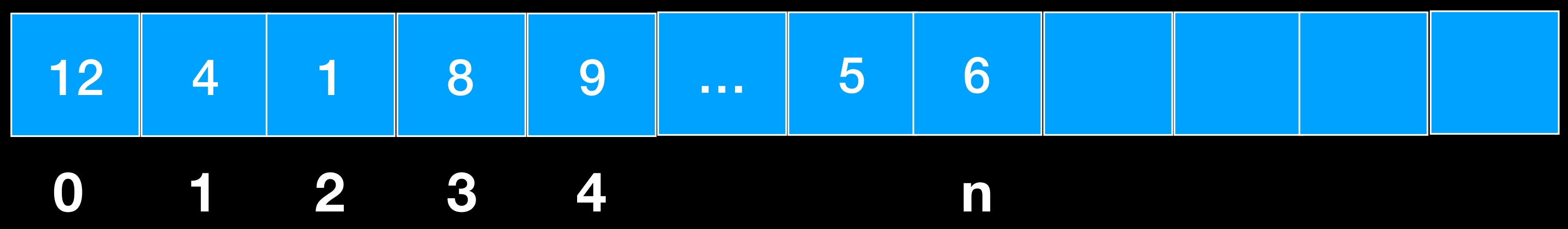
Implementation

Must preserve order
No swapping tricks

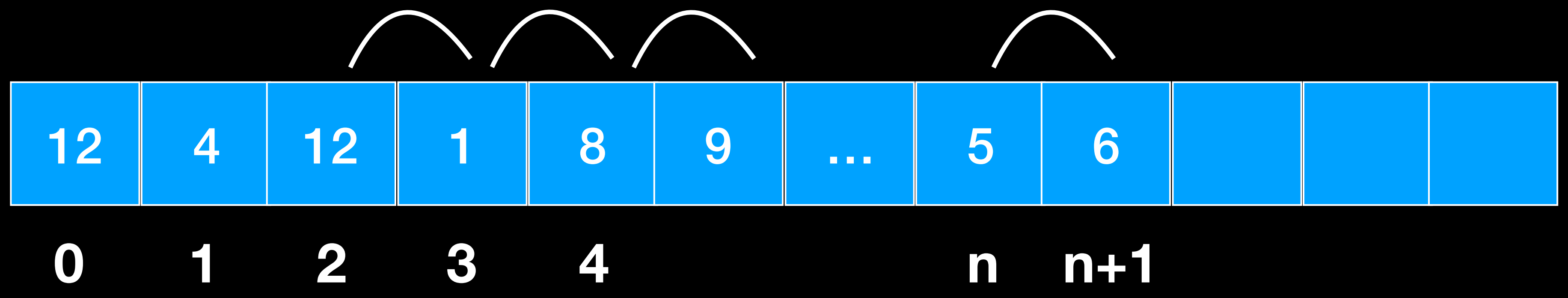
Array?

Linked Chain?

Array

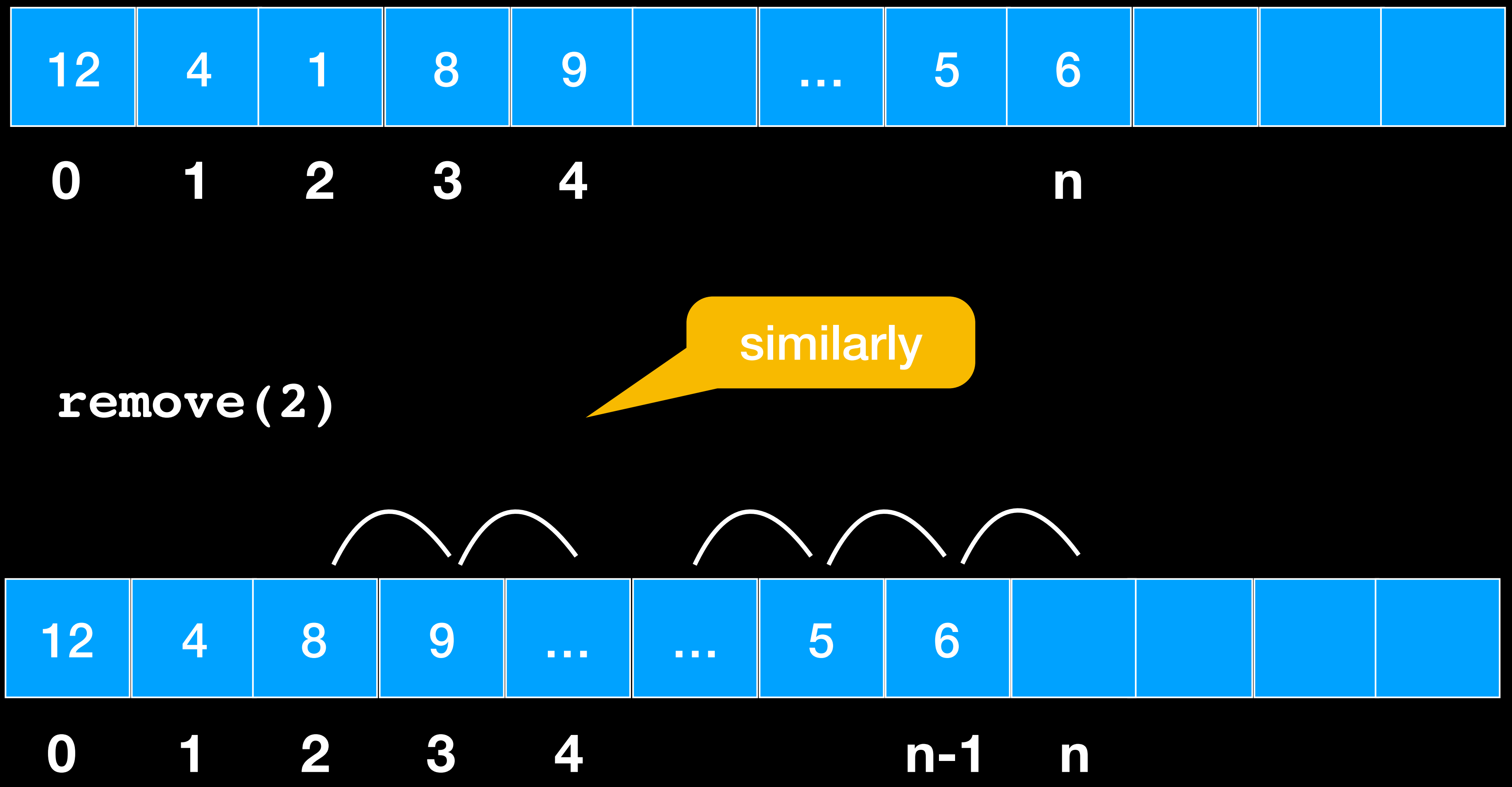


`insert(2, 12)`



Must shift `n-(position+1)` elements

Array



Must shift `n-position` elements

Array Analysis

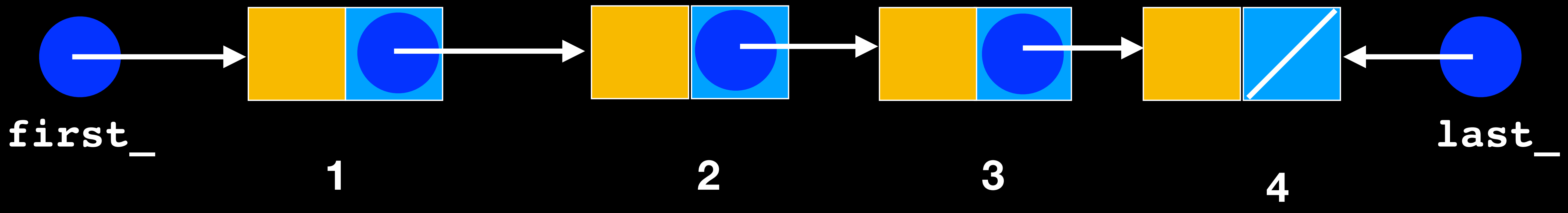
With Array both insert and remove are “Expensive”

Number of operations depends on size of List

Can we do better?

What makes a list?

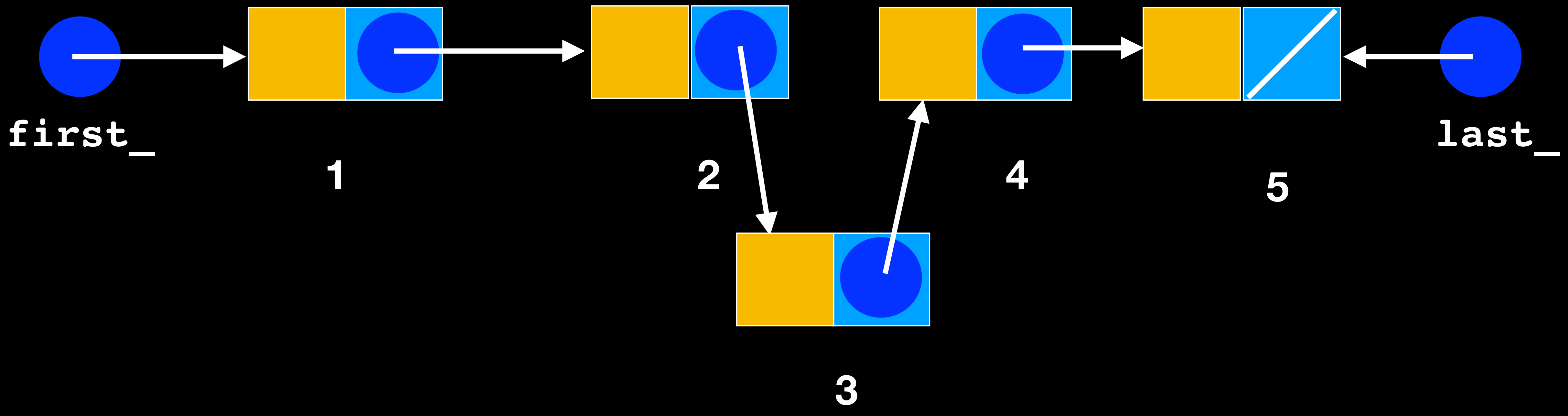
Order is implied



What makes a list?

Order is implied

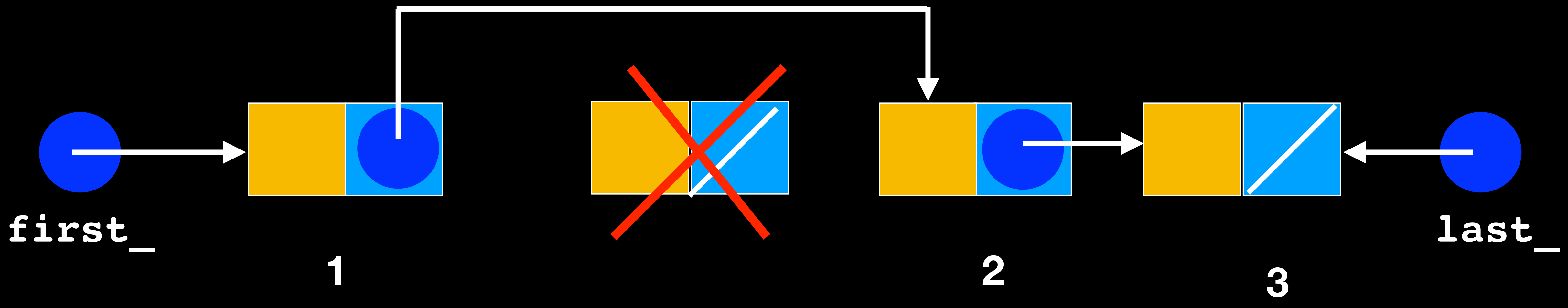
Insertion and removal from middle retains order



What makes a list?

Order is implied

Insertion and removal from middle retains order



What’s the catch?

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, umber 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 back		
Retain order with insert and remove at front		
Retain order with insert and remove In the middle		

✓ 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 back		
Retain order with insert and remove at front		
Retain order with insert and remove In the middle		

✓ 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 back	✓	✓
Retain order with insert and remove at front		
Retain order with insert and remove In the middle		

✓ 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 back	✓	✓
Retain order with insert and remove at front	✗	✓
Retain order with insert and remove In the middle		

✓ Efficient, does not depend on # of items

✗ Expensive, depends on # of items

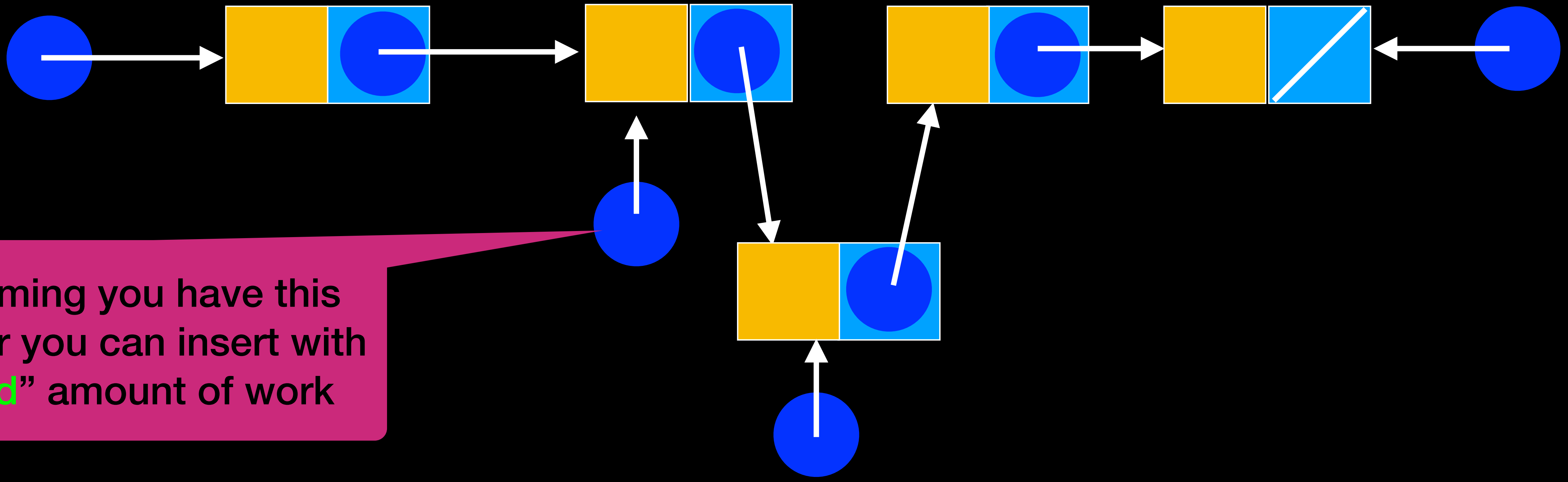
No sifting but incurs cost of finding the node to remove (call to `getPointerTo`)

	Arrays	Size Linked List
Random/direct access	✓	✗
Retain order with Insert and remove At the back	✓	✓
Retain order with insert and remove at front	✗	✓
Retain order with insert and remove In the middle	✗	✗

Singly-Linked List

INSERT

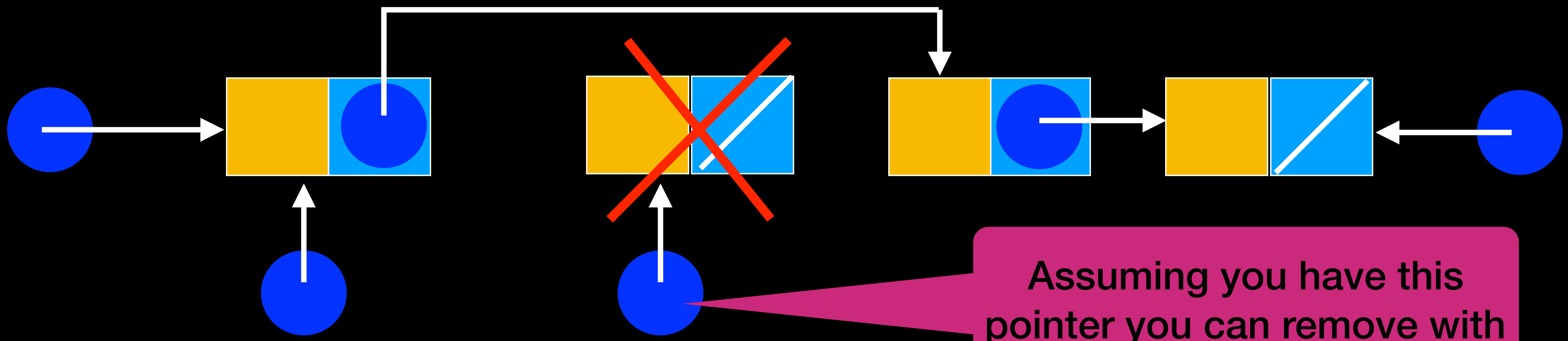
```
void insert(size_t position, T new_element);
```



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

REMOVE

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



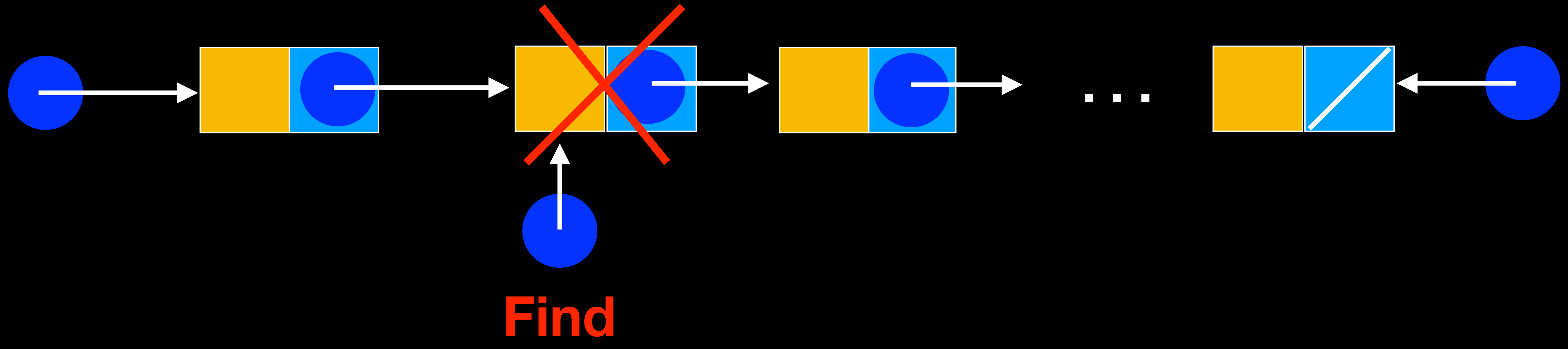
Assuming you have this pointer you can remove with "fixed" amount of work

Caveat

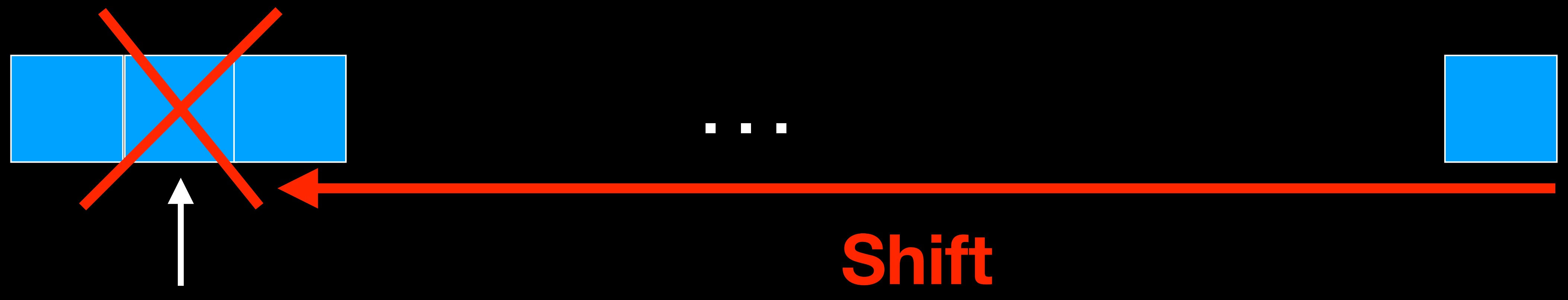
Find the pointer to the node before inserting/removing —> traversal: high cost - *depends on number of elements in list*

If operations (insertion/deletions) occur on nodes that are close to each other operation cost can stay low - in an ordered list this is more likely

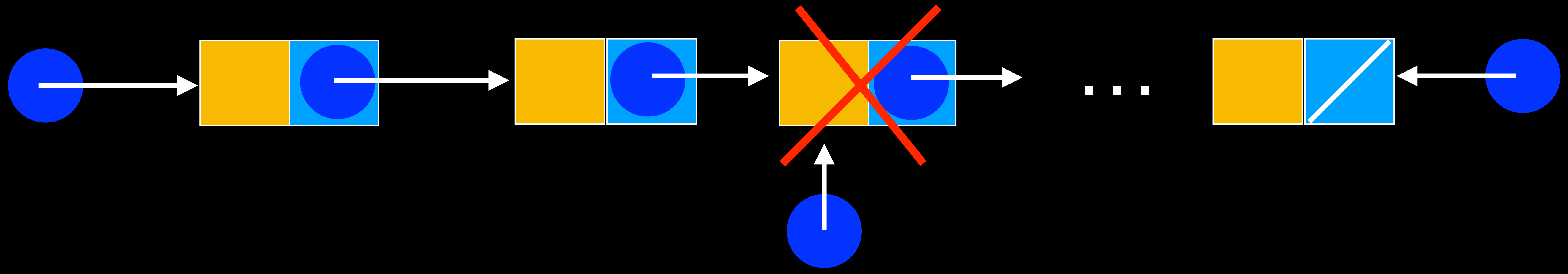
Linked List



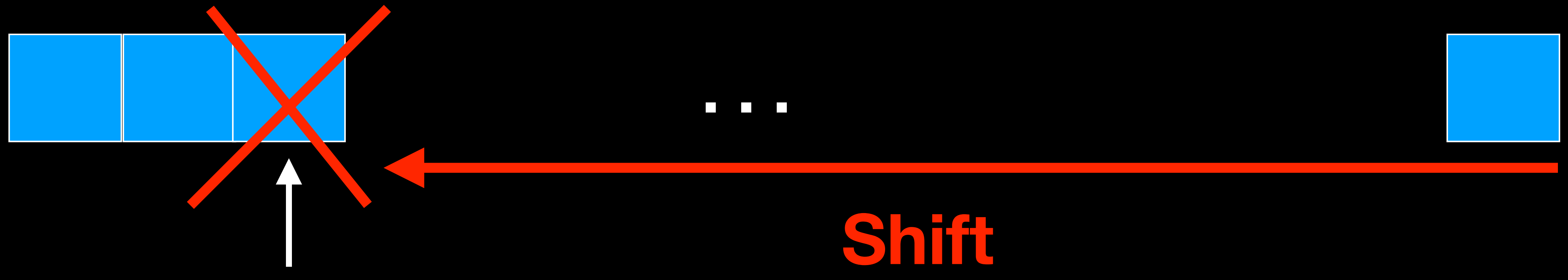
Array



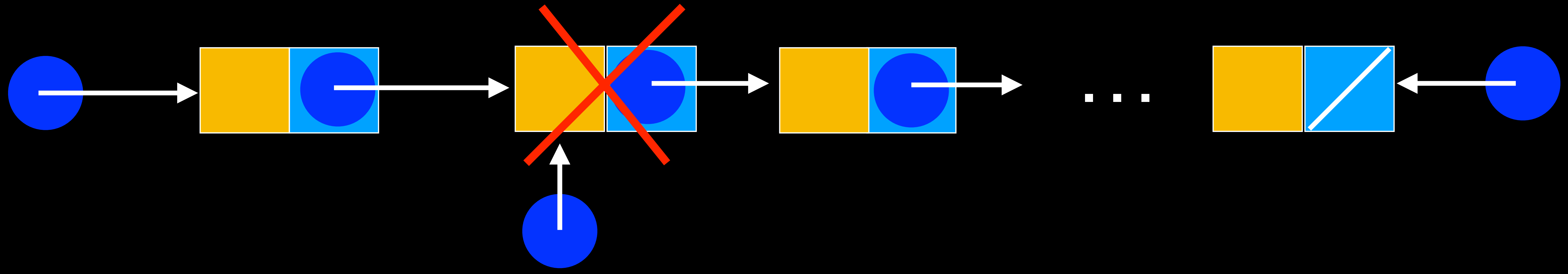
Linked List



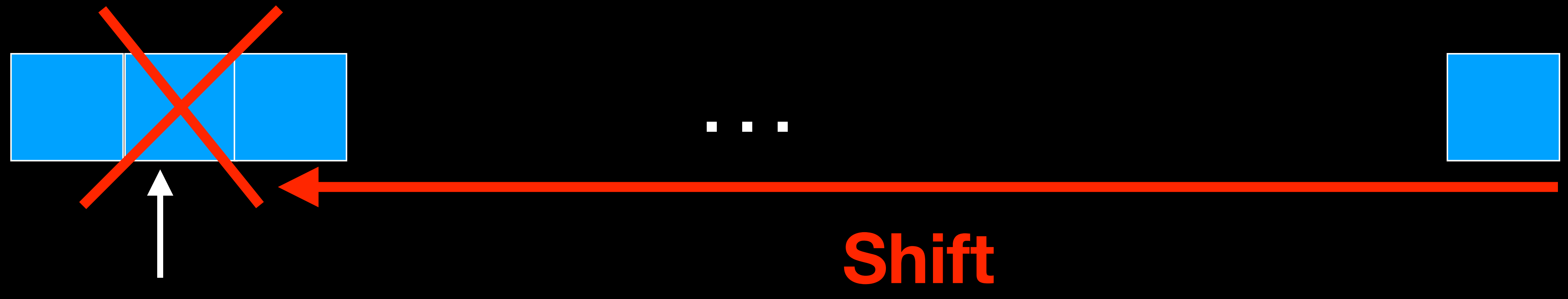
Array



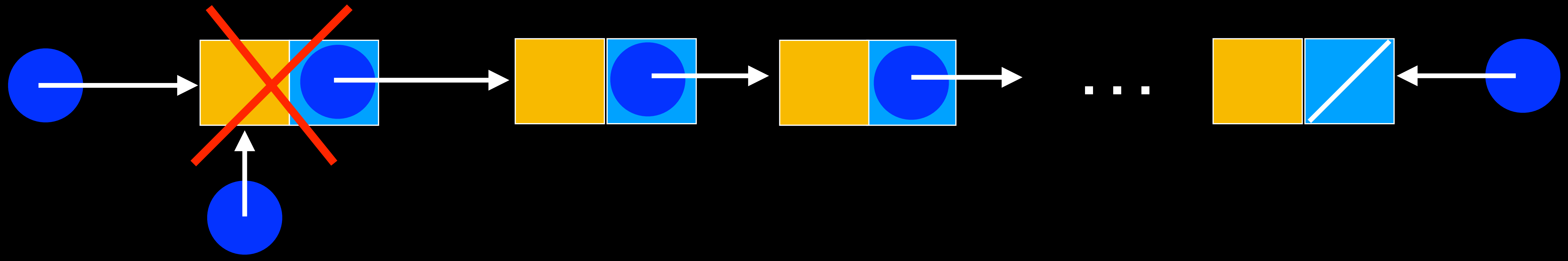
Linked List



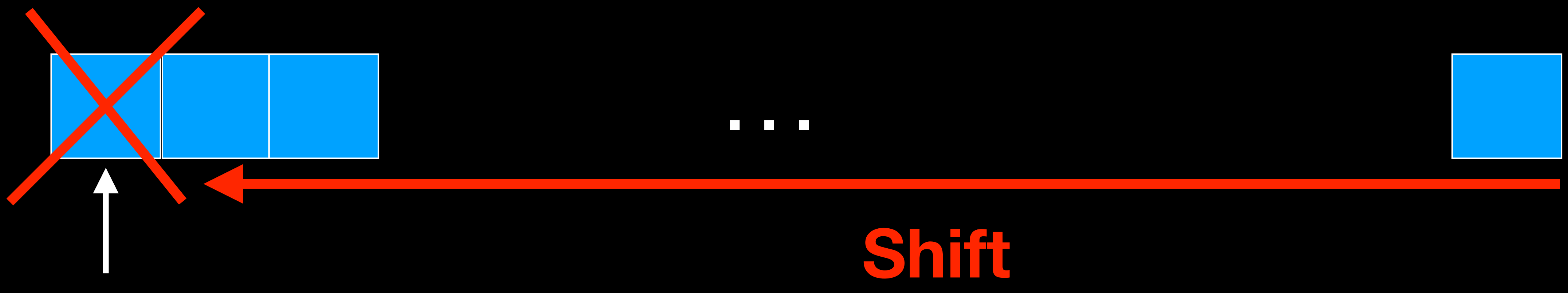
Array



Linked List

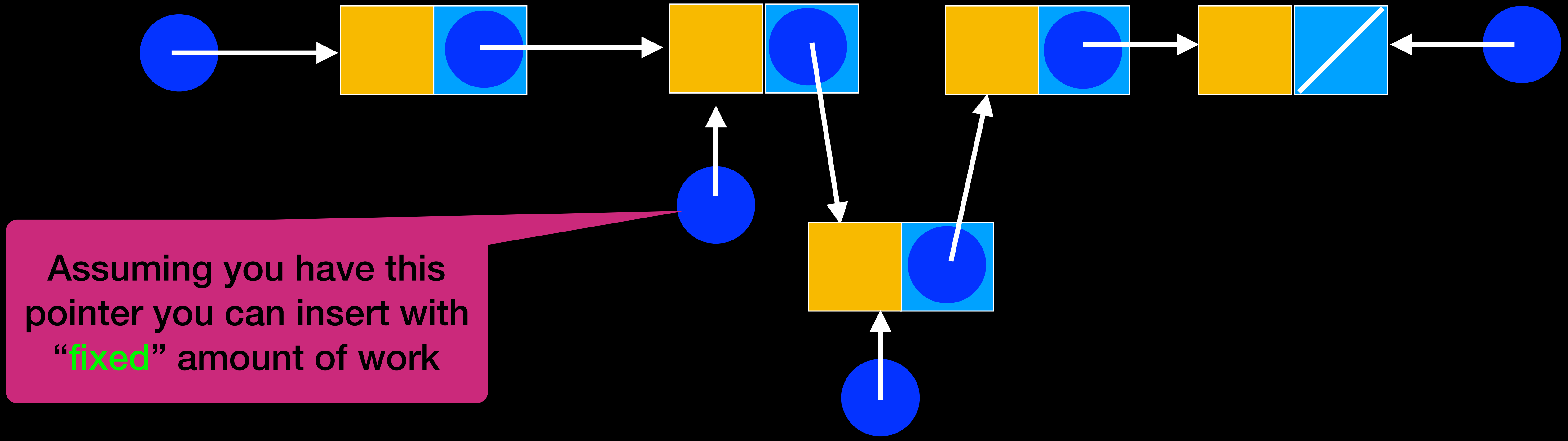


Array



INSERT

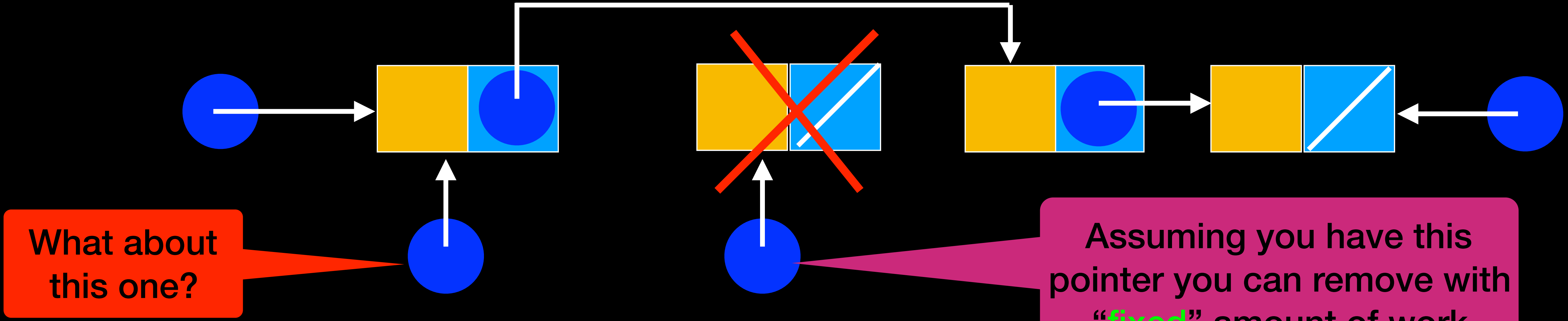
```
void insert(size_t position, T new_element);
```



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

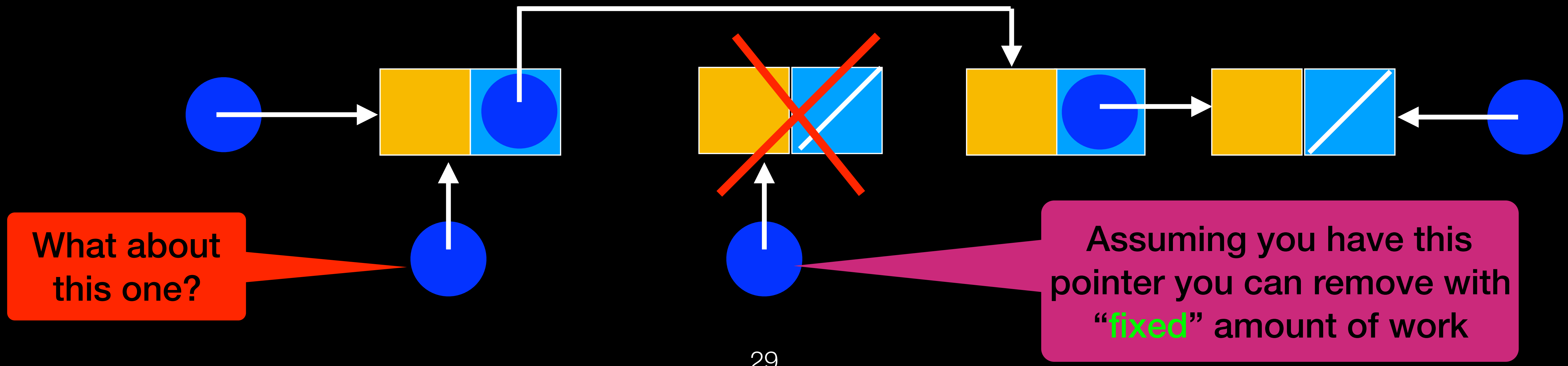
Lecture Activity

Design

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

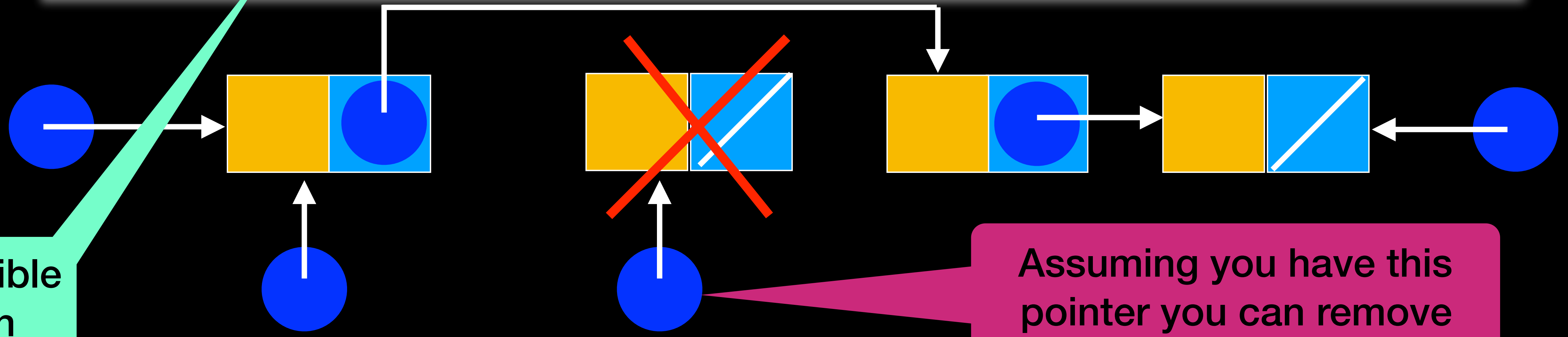
```
void remove(size_t position);
```


What about this one?
Assuming you have this pointer you can remove with "fixed" amount of work

REMOVE

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

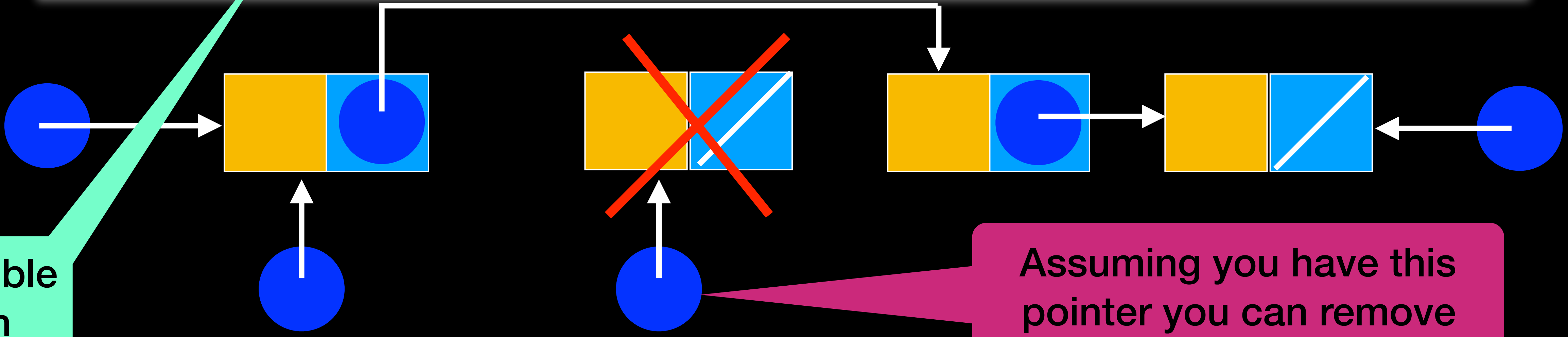
One possible solution



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

REMOVE

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

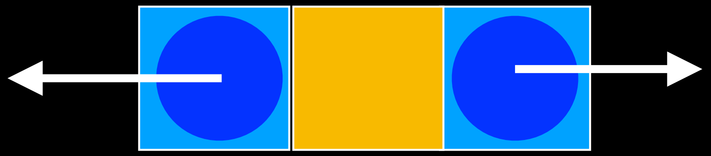


One possible solution

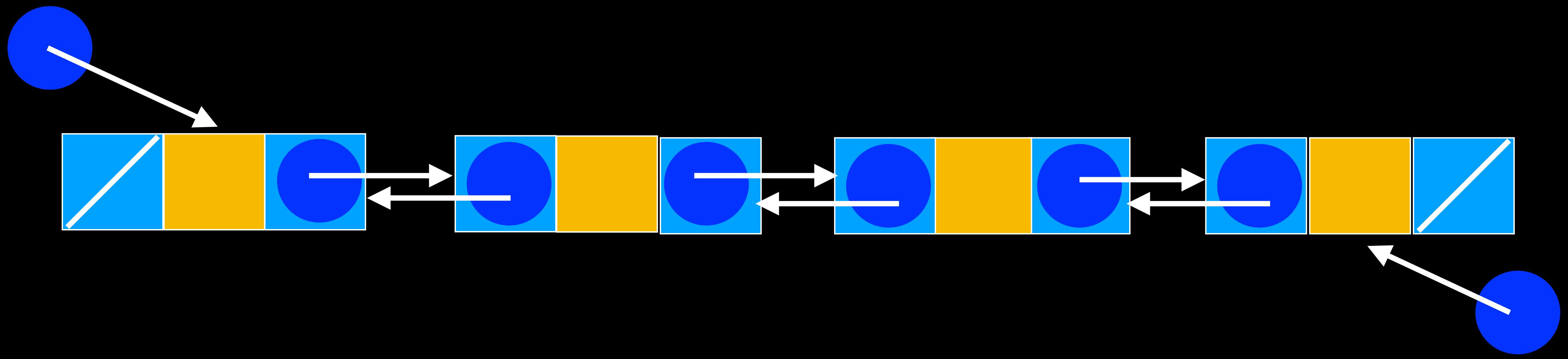
Assuming you have this pointer you can remove with “fixed” amount of work

Another Solution?

```
#ifndef NODE_H_
#define NODE_H_

template<class T>
class Node
{

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

private:
     T item;                     // A data item
     Node<T>* next_;             // Pointer to next node
     Node<T>* previous_;        // Pointer to previous node
}; // end Node



#include "Node.cpp"
#endif // NODE_H_
```



Doubly Linked List

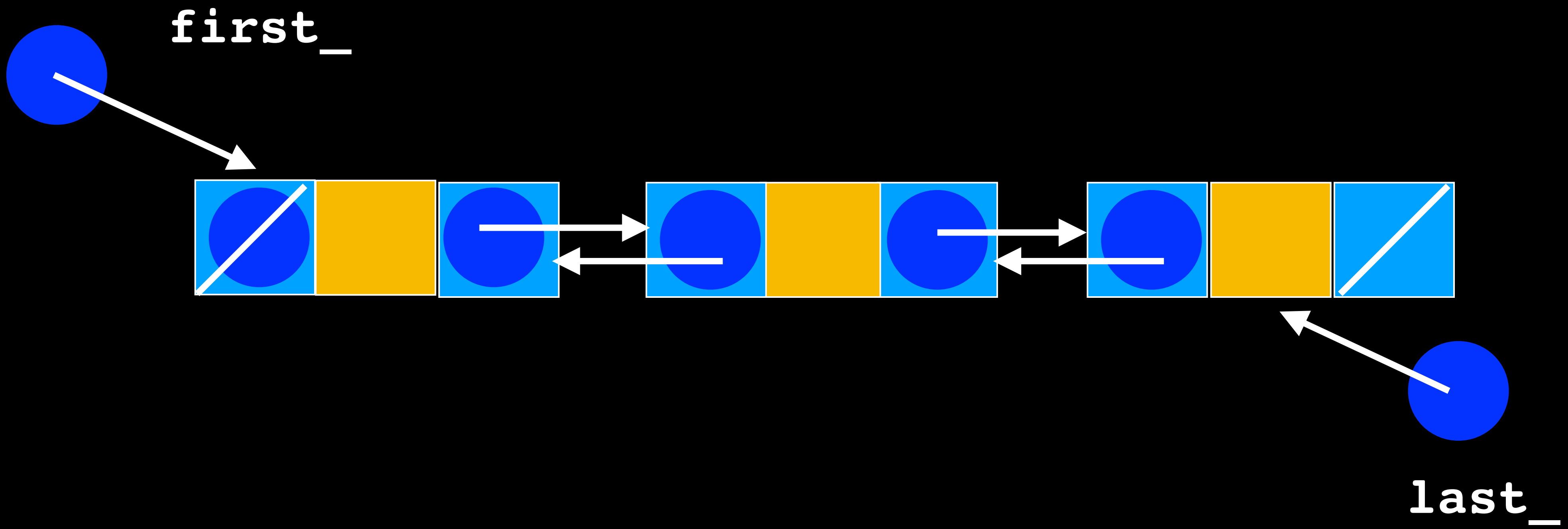


Safe programming: position not pointer - do not expose data structure to direct manipulation outside the class

```
#ifndef LIST_H_
#define LIST_H_

template<class T>
class List
{

public:
     List(); // constructor
     List(const List<T>& a_list); // copy const[illegible]or
     ~List(); // destructor
     bool isEmpty() const;
     size_t getLength() const;
     /retains list order, position is 0 to n-1, if position > n-1 it inserts at end
     bool insert(size_t position, const T& new_element);//retains list order
     bool remove(size_t position);//retains list order
     T getItem(size_t position) const;
     void clear();


private:
     Node<T>* first_;       // Pointer to first node
     Node<T>* last_;        // Pointer to last node
     size_t item_count;     // number of items in the list

     Node<T>* getPointerTo(size_t position) const;
}; // end List

#include "List.cpp"
#endif // LIST_H_
```

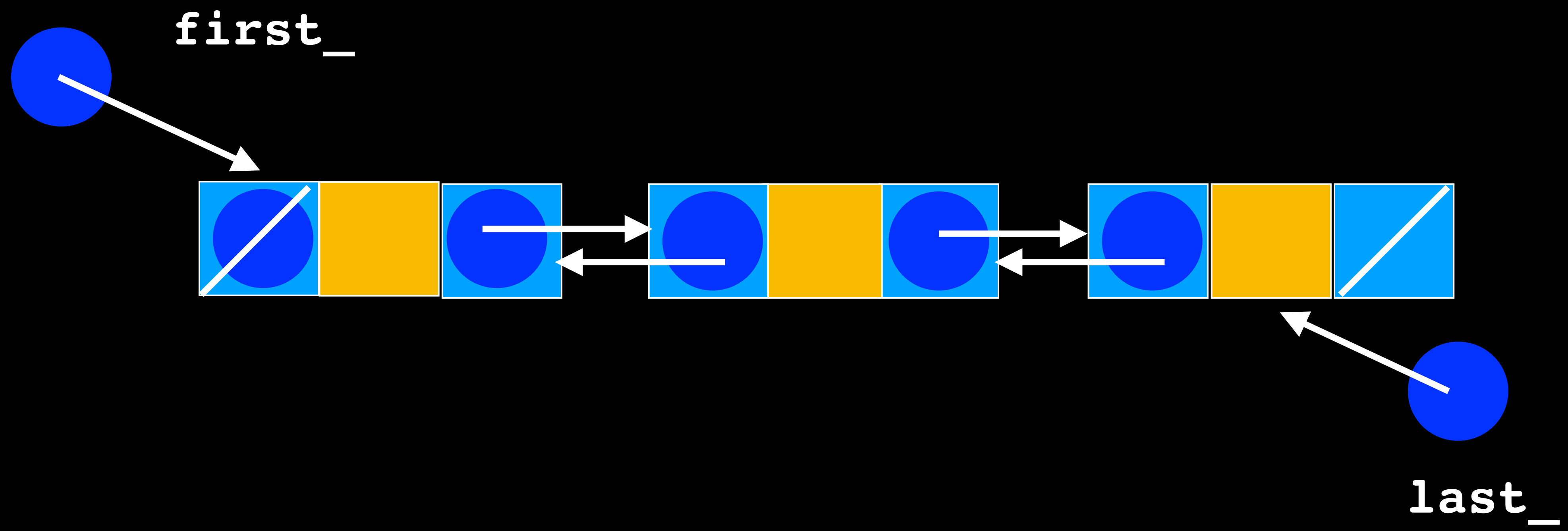
`List::insert`

What are the different cases that should be considered?

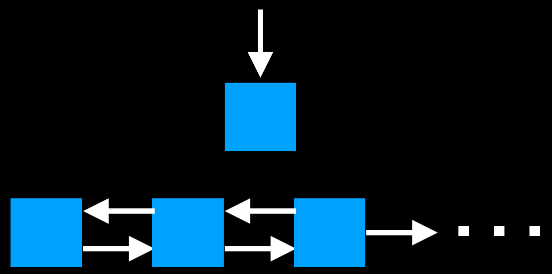


Lecture Activity

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



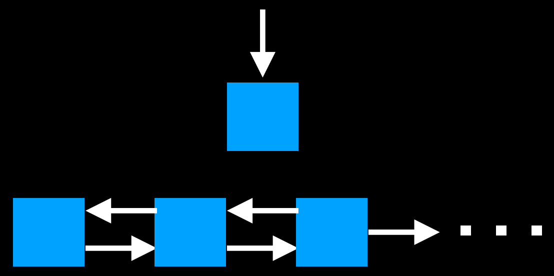
Instantiate new node

Obtain pointer

Connect new node to chain

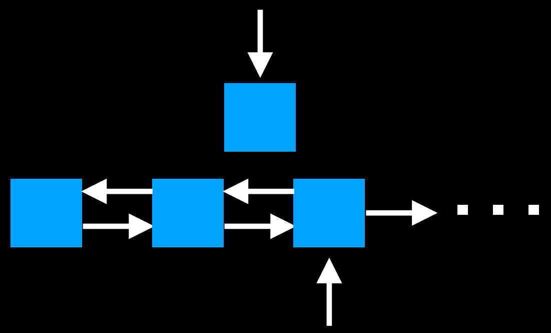
Reconnect the relevant nodes

Pseudocode

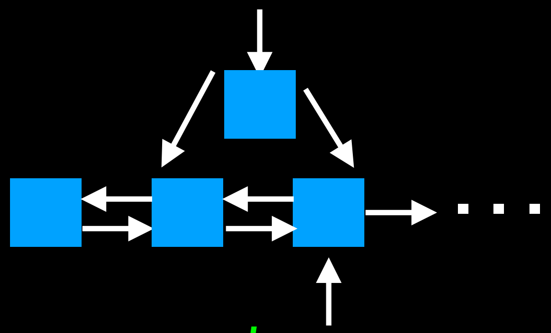


Instantiate new node

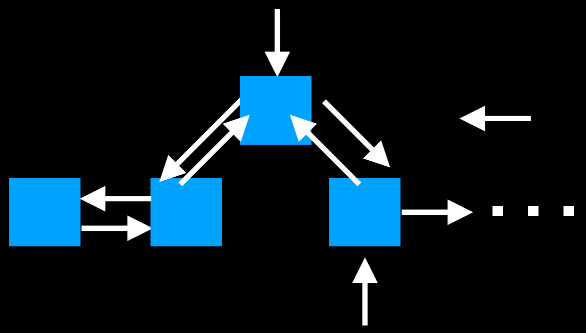
Obtain pointer



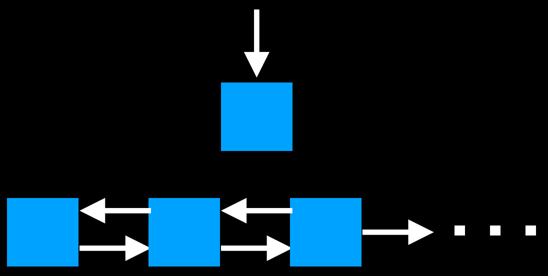
Connect new node to chain



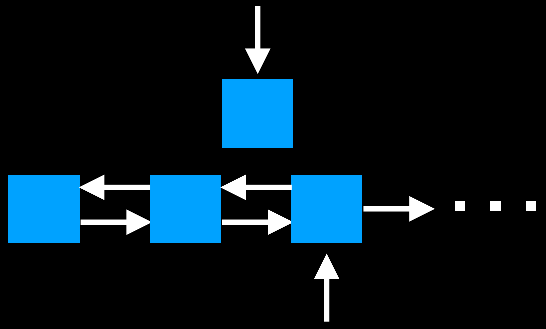
Reconnect the relevant nodes



Pseudocode

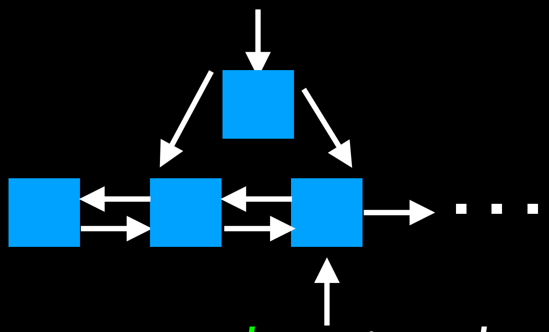


Instantiate new node to be inserted and set its value

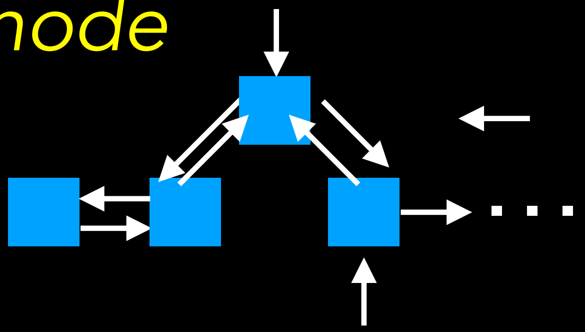


Obtain pointer to node currently at position 2

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->previous

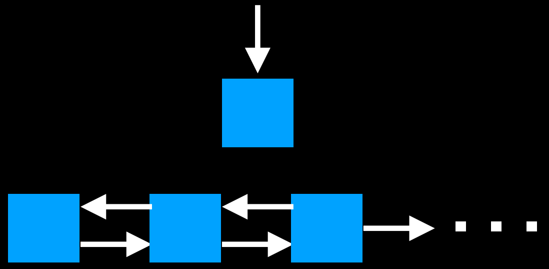


Reconnect the relevant nodes in the chain by pointing position->previous->next to the new node and position->previous to the new node

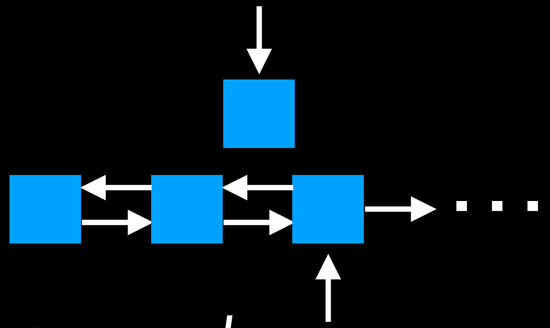


Order Matters!

More Pseudocodey

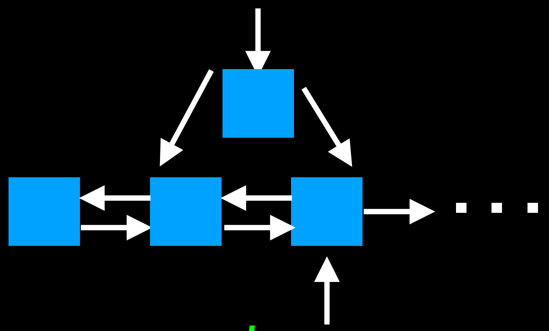


Instantiate new node new_ptr = new Node() and new_ptr->setItem()



Obtain pointer position_ptr = getPointerTo(2)

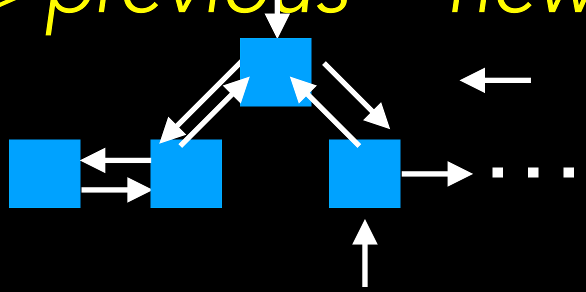
Connect new node to chain new_ptr->next = position_ptr and new_ptr->previous = temp->previous



Reconnect the relevant nodes

position_ptr->previous->next = new_ptr and

position->previous = new_ptr



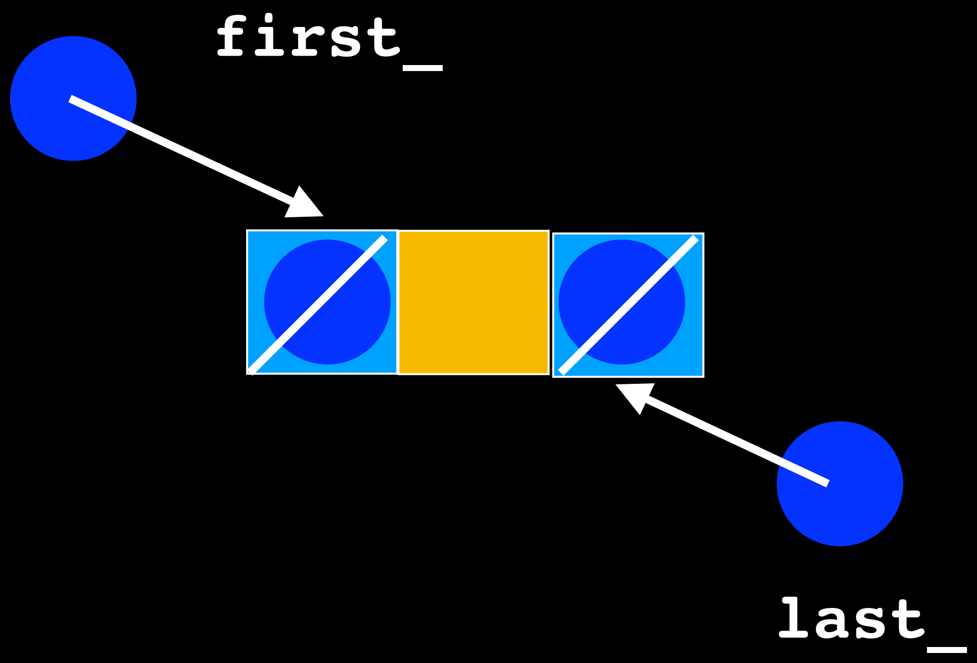
List::insert

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

      // Attach new node to chain

      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
      {
         // 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
      return true;
}  // end insert
```

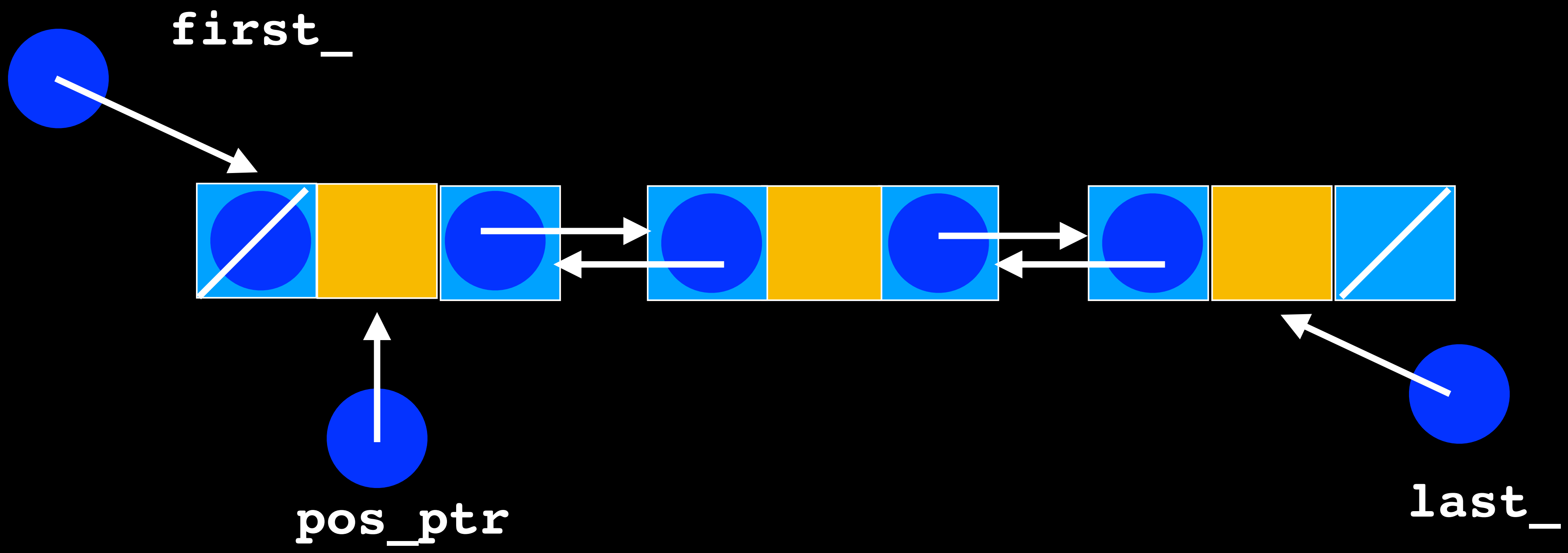
```
   if (first_ == nullptr)
    {
        // Insert first node
        new_node_ptr->setNext(nullptr);
        new_node_ptr->setPrevious(nullptr);
        first_ = new_node_ptr;
        last_ = new_node_ptr;
    }
```

Always insert

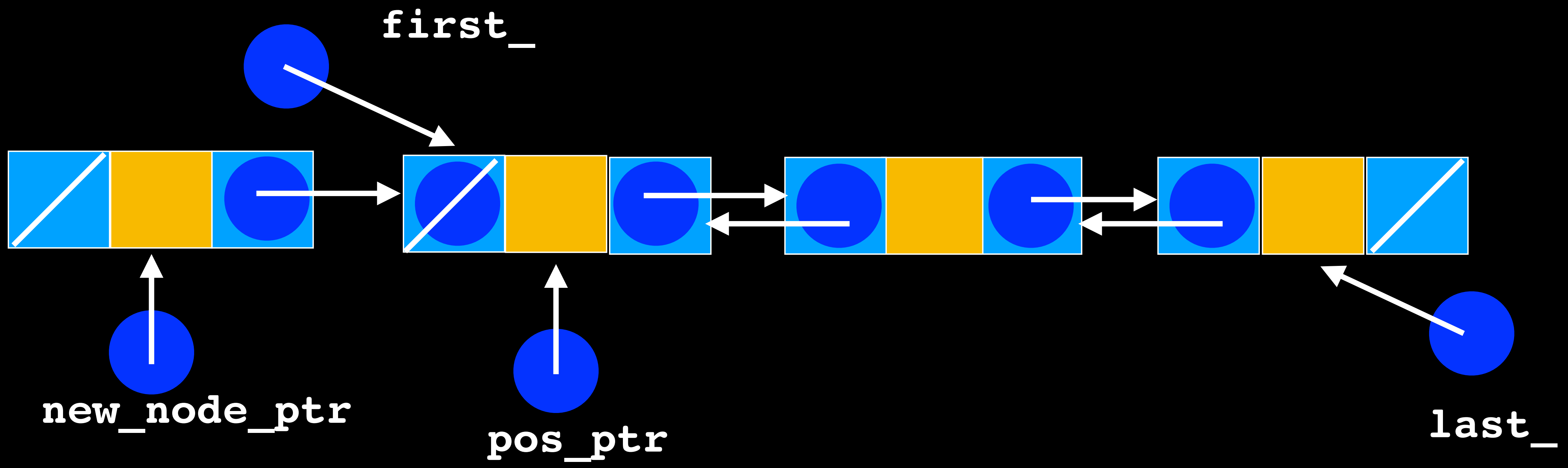
```
if (first_ == nullptr)
{
    // Insert first node
    new_node_ptr->setNext(nullptr);
    new_node_ptr->setPrevious(nullptr);
    first_ = new_node_ptr;
    last_ = new_node_ptr;
}
```



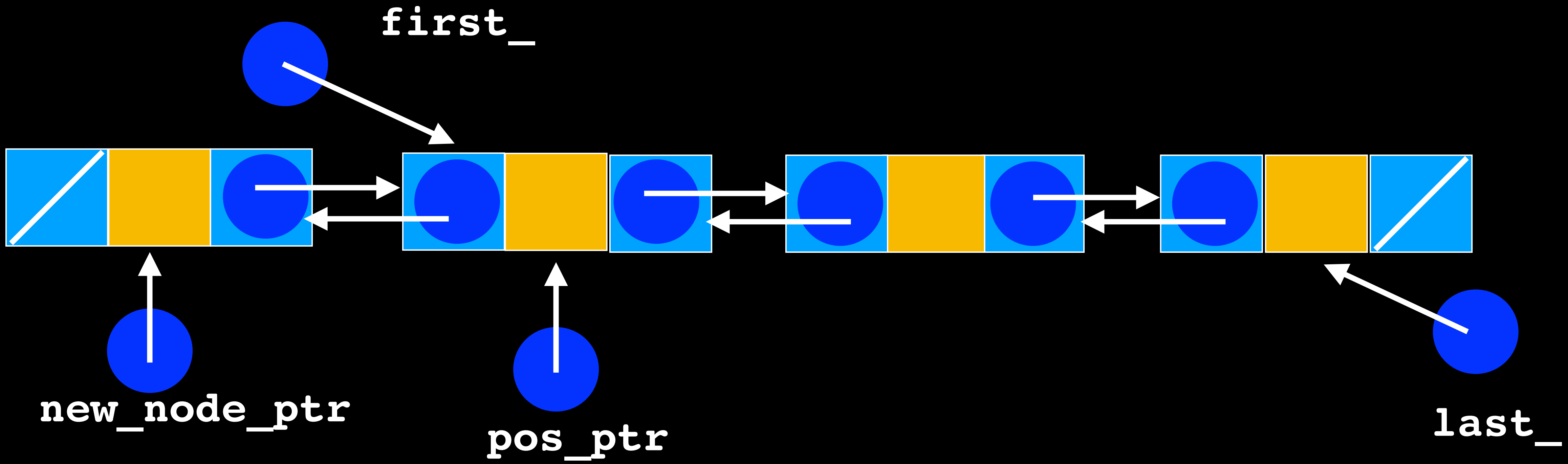
```
else if (pos_ptr == first_)
      {
         // Insert new node at beginning of chain
         new_node_ptr->setNext(first_);
         new_node_ptr->setPrevious(nullptr);
         first_->setPrevious(new_node_ptr);
         first_ = new_node_ptr;
      }
```



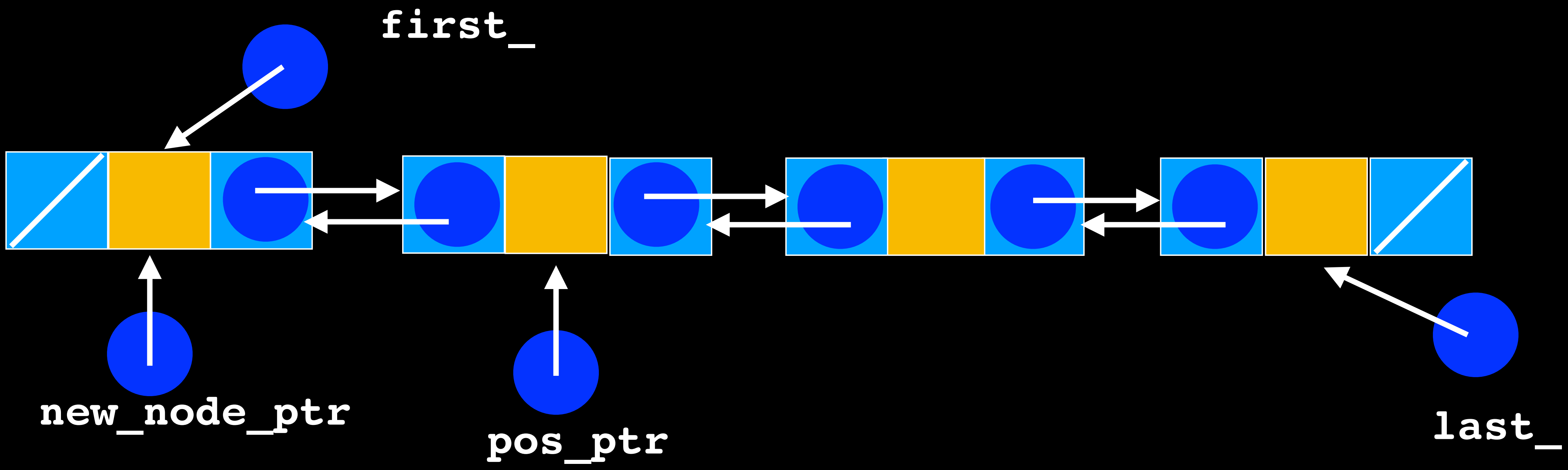
```
else if (pos_ptr == first_)
     {
        // Insert new node at beginning of chain
        new_node_ptr->setNext(first_);
        new_node_ptr->setPrevious(nullptr);
        first_->setPrevious(new_node_ptr);
        first_ = new_node_ptr;
     }
```



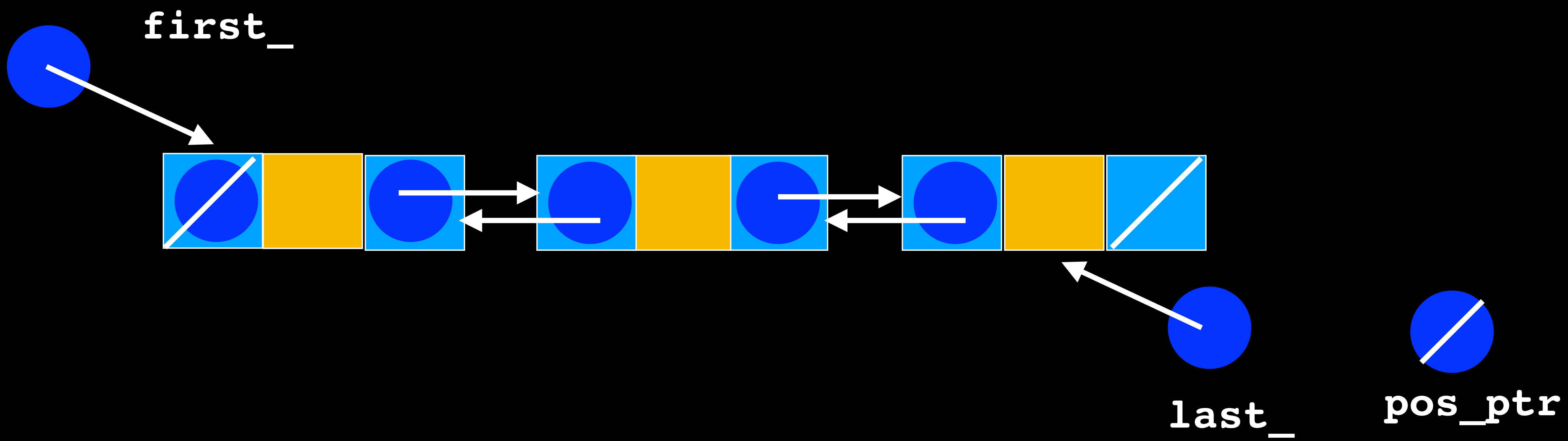
```
else if (pos_ptr == first_)
      {
         // Insert new node at beginning of chain
         new_node_ptr->setNext(first_);
         new_node_ptr->setPrevious(nullptr);
         first_->setPrevious(new_node_ptr);
         first_ = new_node_ptr;
      }
```



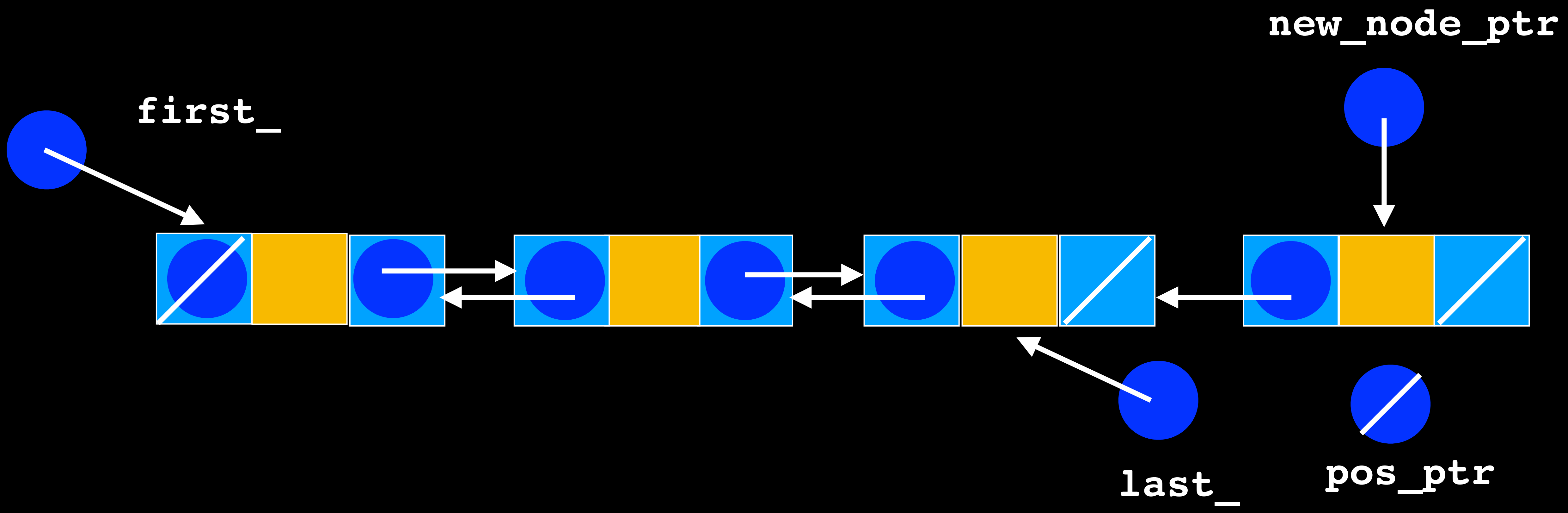
```
else if (pos_ptr == first_)
      {
         // Insert new node at beginning of chain
         new_node_ptr->setNext(first_);
         new_node_ptr->setPrevious(nullptr);
         first_->setPrevious(new_node_ptr);
         first_ = new_node_ptr;
      }
```



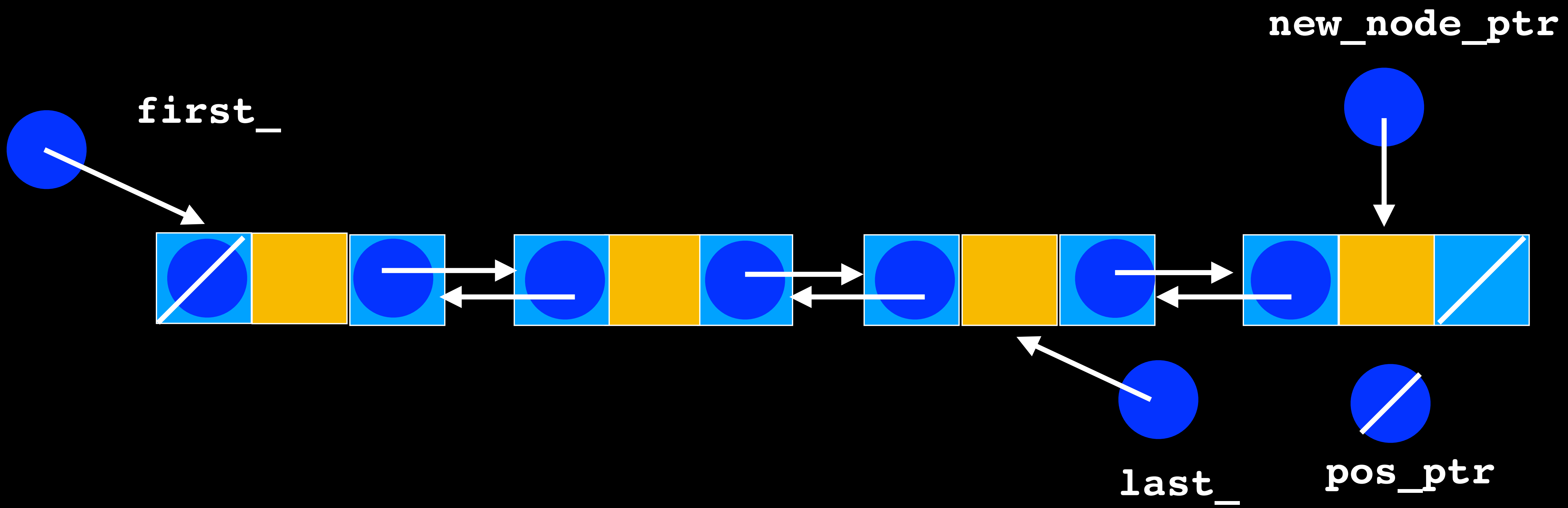
```
else if (pos_ptr == nullptr)
{
    //insert at end of list
    new_node_ptr->setNext(nullptr);
    new_node_ptr->setPrevious(last_);
    last_->setNext(new_node_ptr);
    last_ = new_node_ptr;
}
```



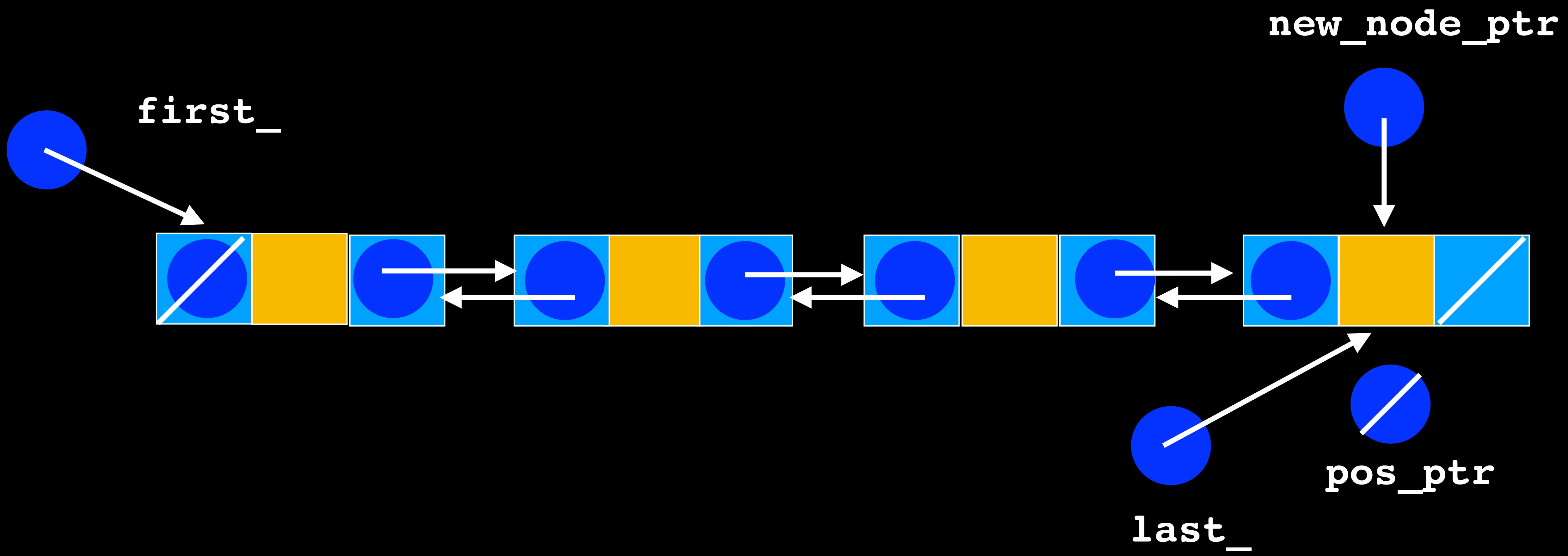
```
else if (pos_ptr == nullptr)
{
    //insert at end of list
    new_node_ptr->setNext(nullptr);
    new_node_ptr->setPrevious(last_);
    last_->setNext(new_node_ptr);
    last_ = new_node_ptr;
}
```



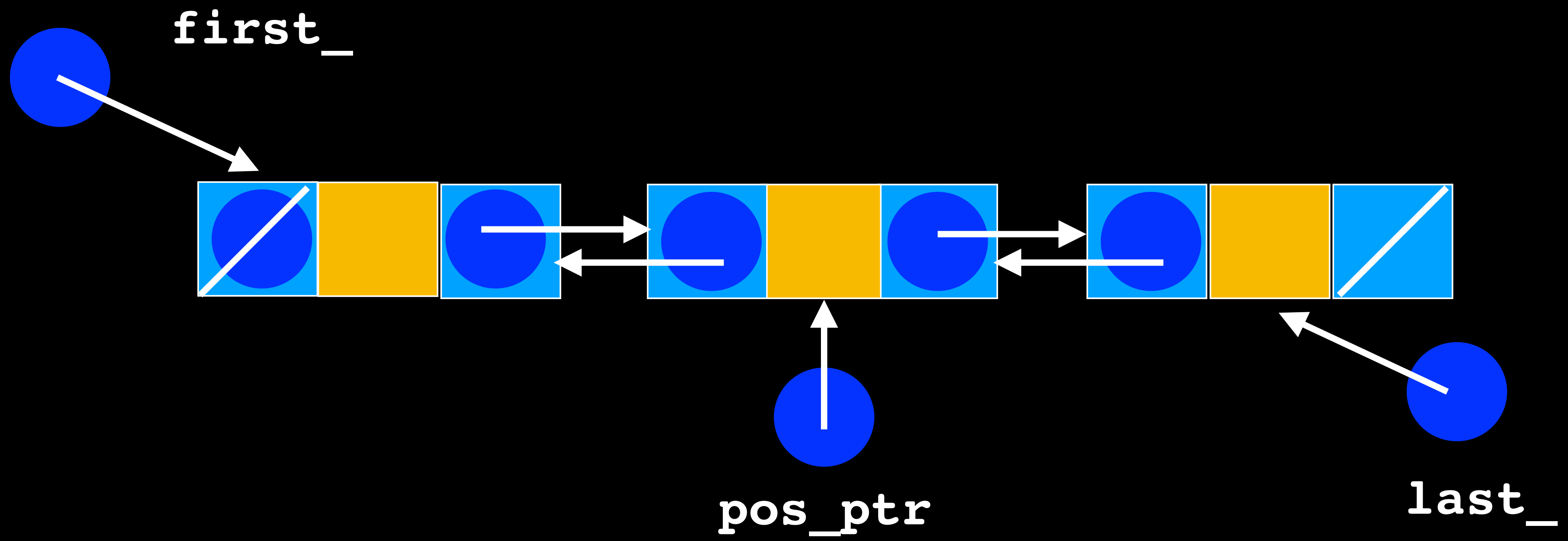
```
else if (pos_ptr == nullptr)
{
    //insert at end of list
    new_node_ptr->setNext(nullptr);
    new_node_ptr->setPrevious(last_);
    last_->setNext(new_node_ptr);
    last_ = new_node_ptr;
}
```



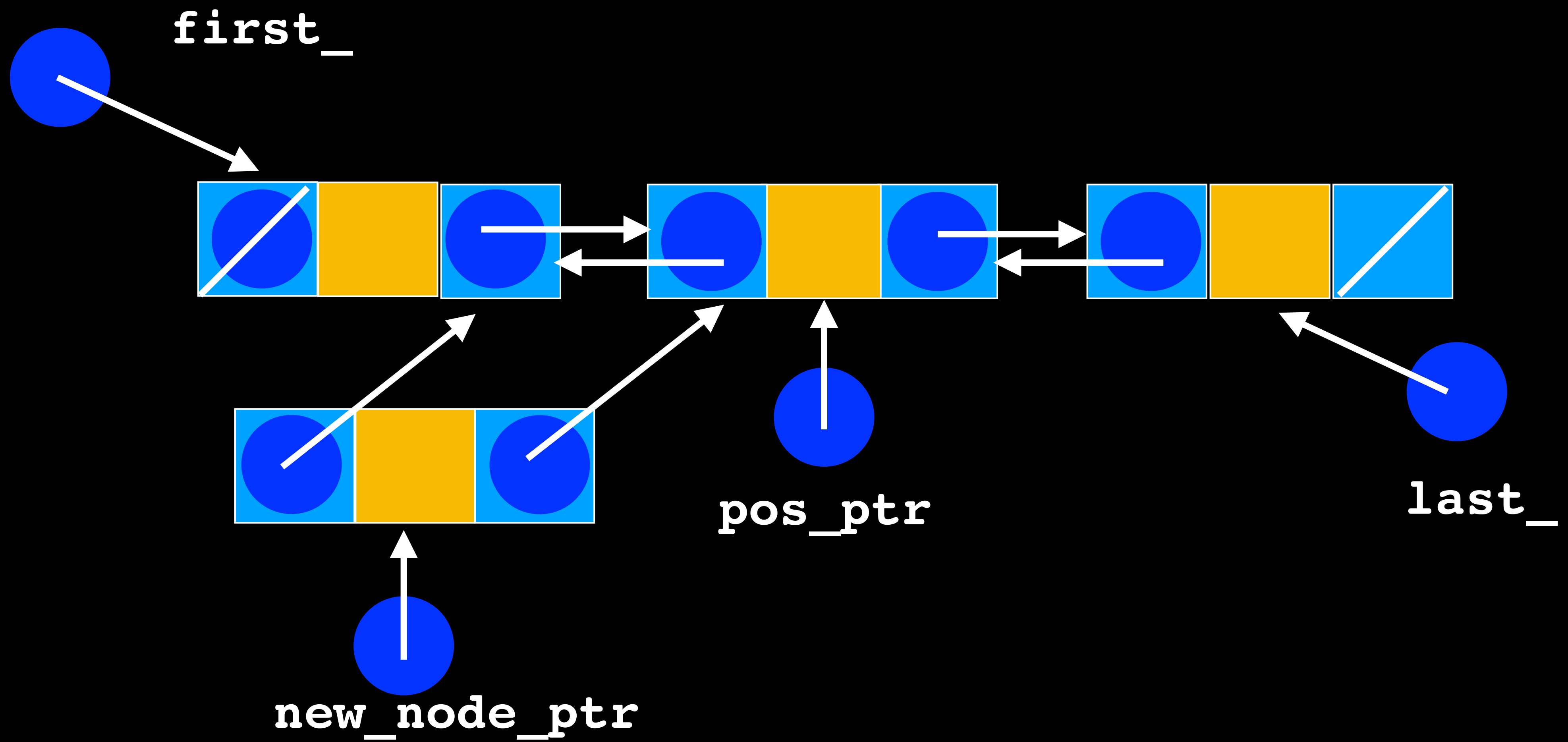
```
else if (pos_ptr == nullptr)
{
    //insert at end of list
    new_node_ptr->setNext(nullptr);
    new_node_ptr->setPrevious(last_);
    last_->setNext(new_node_ptr);
    last_ = new_node_ptr;
}
```



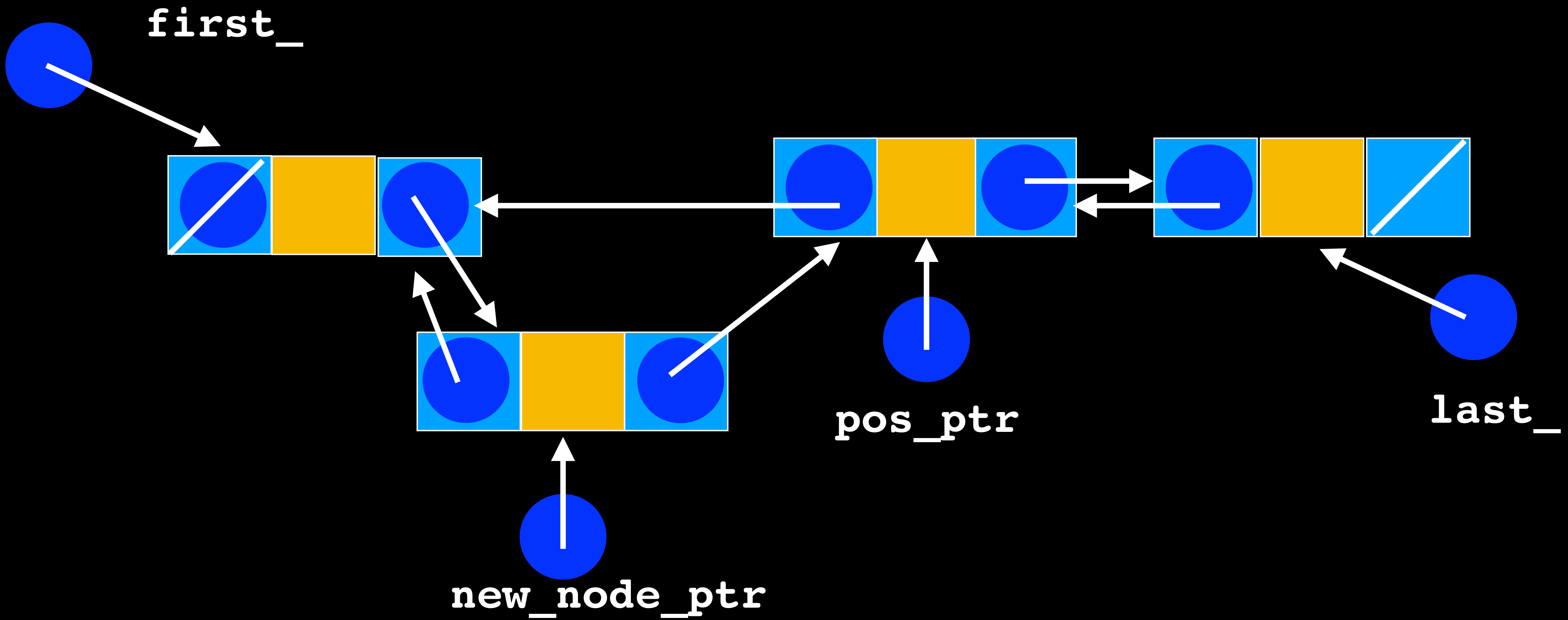
```
else
    {
       // Insert new node before node to which position points
       new_node_ptr->setNext(pos_ptr);
       new_node_ptr->setPrevious(pos_ptr->getPrevious());
       pos_ptr->getPrevious()->setNext(new_node_ptr);
       pos_ptr->setPrevious(new_node_ptr);
    }  // end if
```



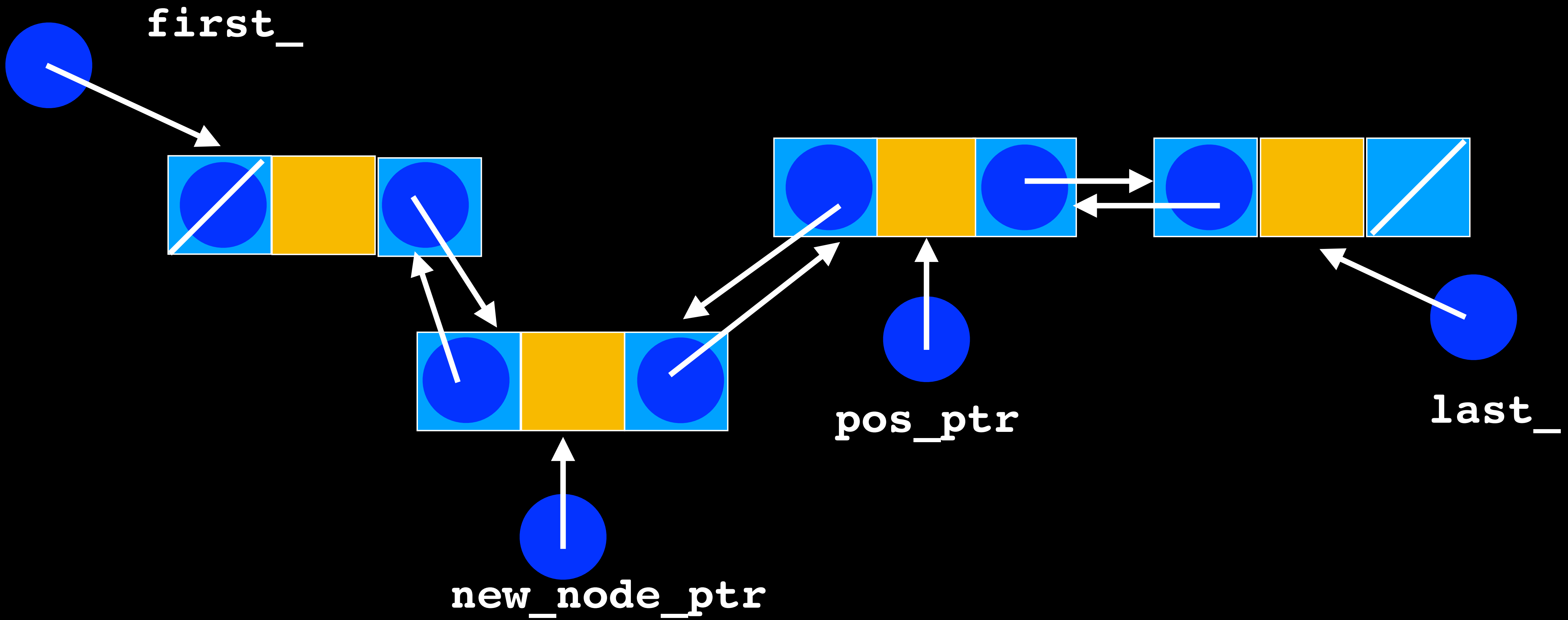
```
else
    {
       // Insert new node before node to which position points
       new_node_ptr->setNext(pos_ptr);
       new_node_ptr->setPrevious(pos_ptr->getPrevious());
       pos_ptr->getPrevious()->setNext(new_node_ptr);
       pos_ptr->setPrevious(new_node_ptr);
    }  // end if
```



```
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
```

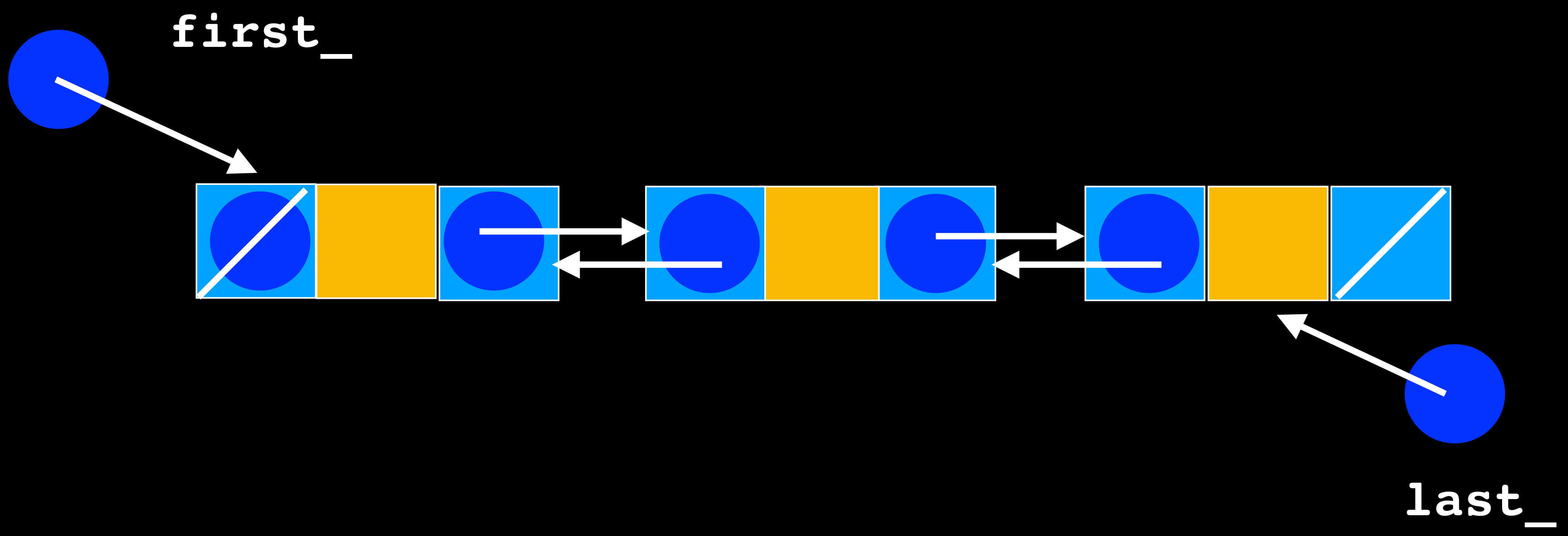


```
else
   {
      // Insert new node before node to which position points
      new_node_ptr->setNext(pos_ptr);
      new_node_ptr->setPrevious(pos_ptr->getPrevious());
      pos_ptr->getPrevious()->setNext(new_node_ptr);
      pos_ptr->setPrevious(new_node_ptr);
   }  // end if
```



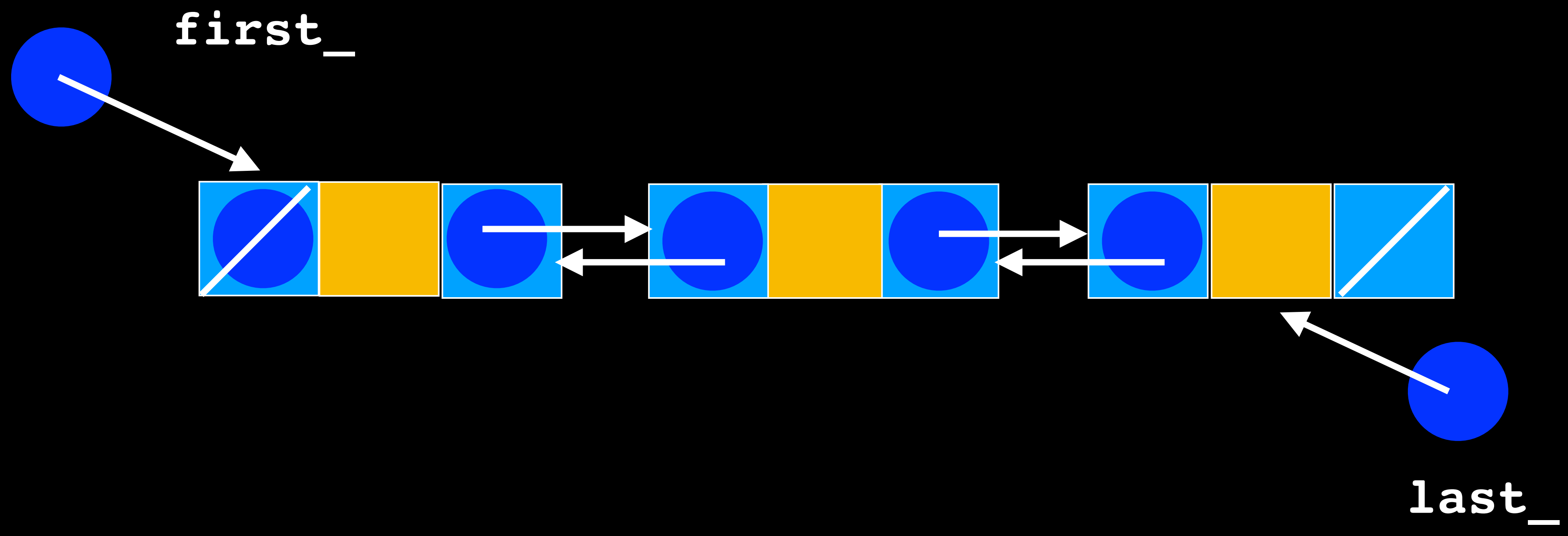
List::remove

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 item_count - 1, and there is a node at position 2)

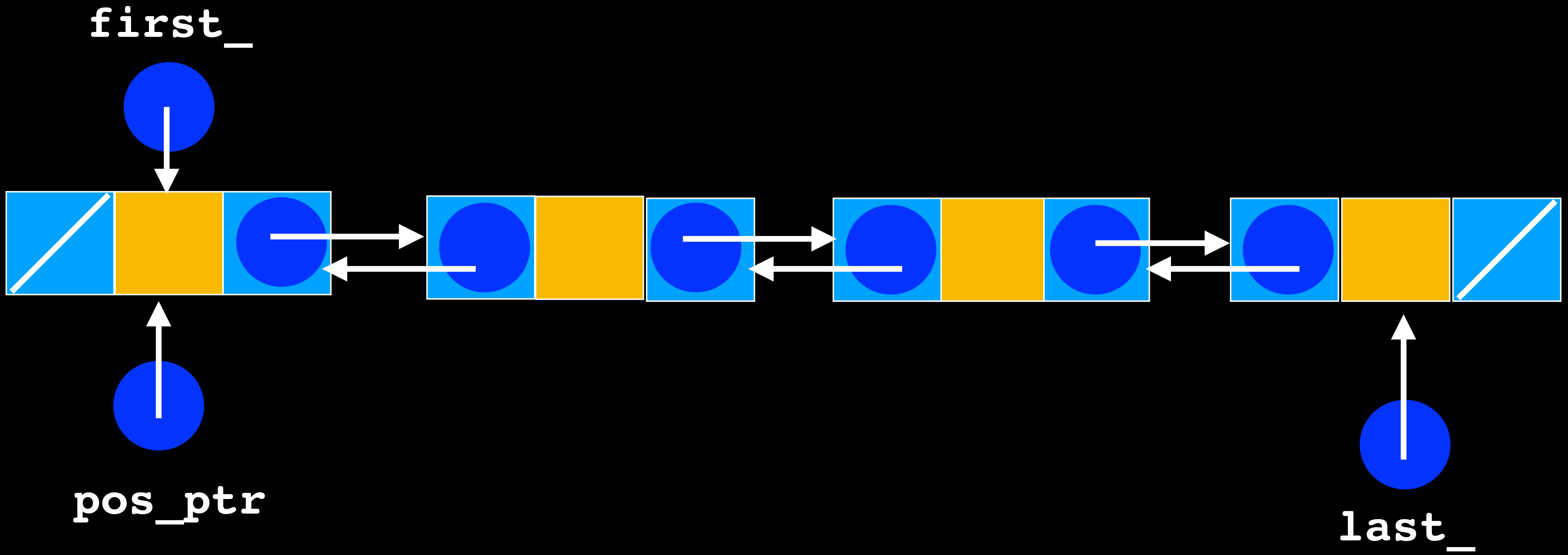


List::Remove

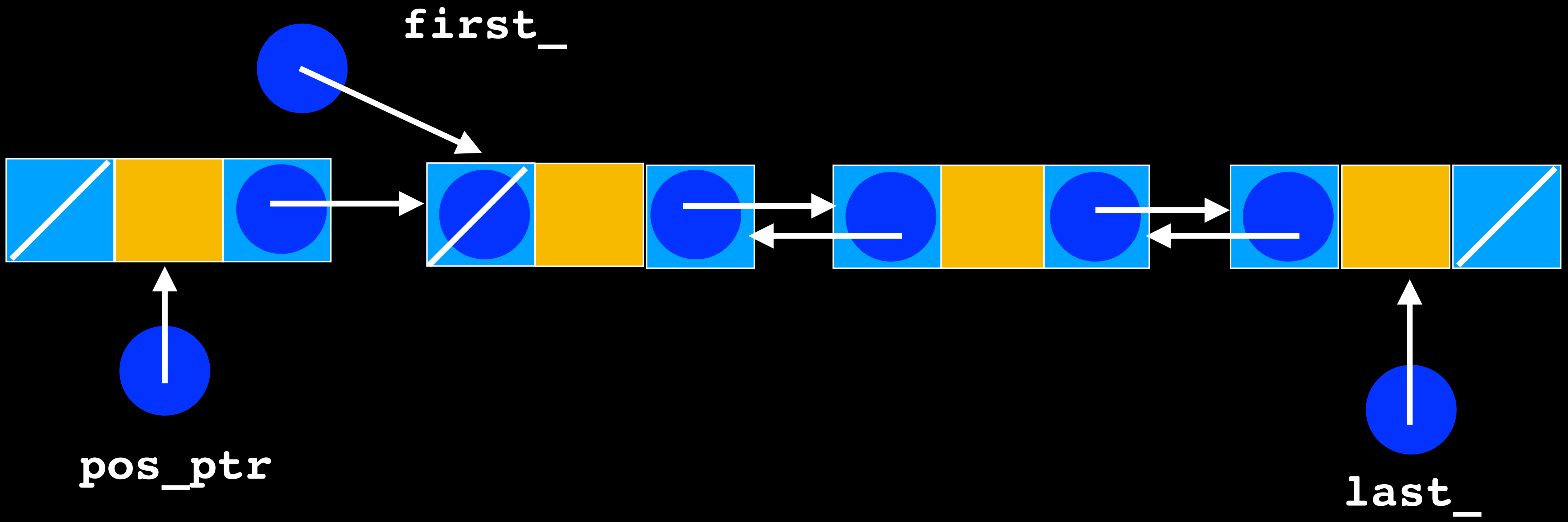
```
template<class T>
bool List<T>::remove(size_t position)
{
    // get pointer to position
    Node<T>* pos_ptr = getPointerTo(position);
   if (pos_ptr == nullptr) // no node at position
      return false;
   else
   {
      // Remove node from chain


      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;
}
```

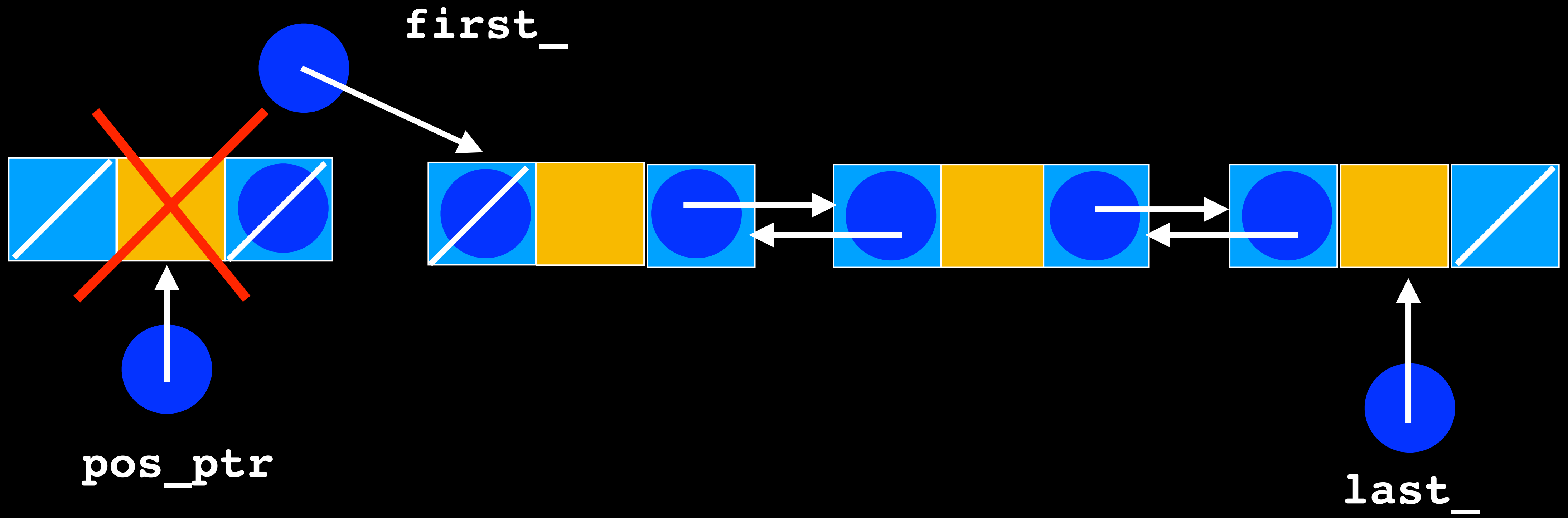
```
// Remove node from chain
if (pos_ptr == first_)
{
    // Remove first node
    first_ = pos_ptr->getNext();
    first_->setPrevious(nullptr);

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



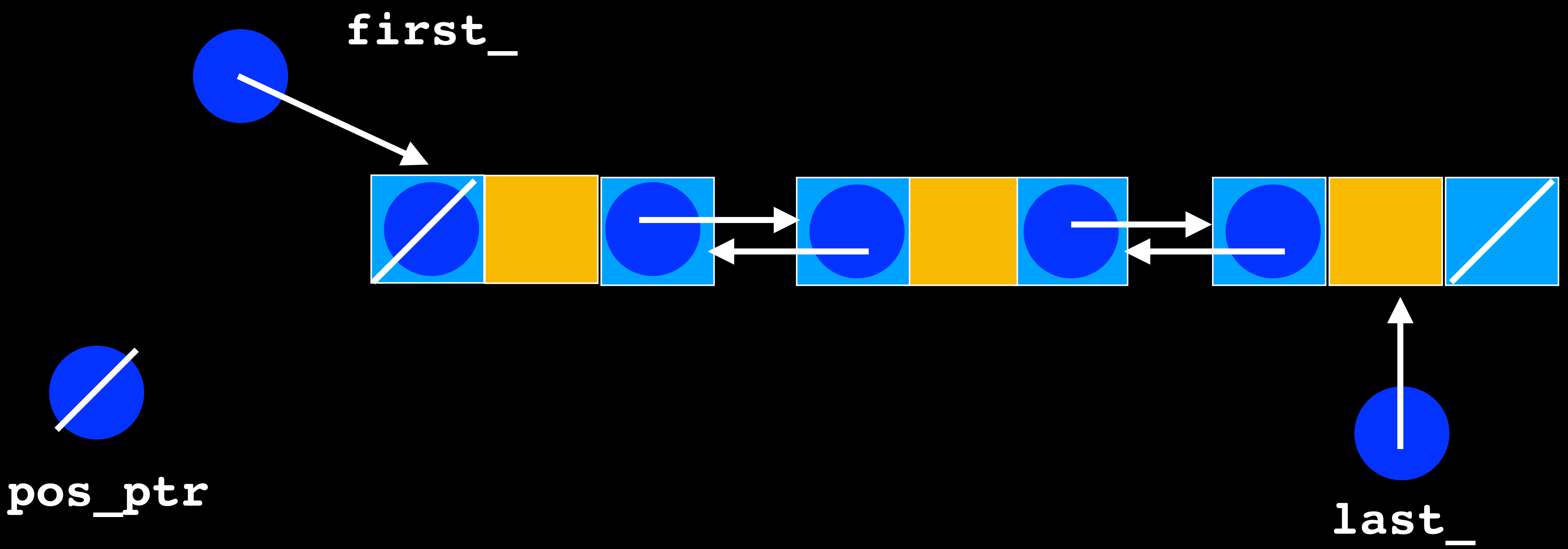
```
// Remove node from chain
if (pos_ptr == first_)
{
    // Remove first node
    first_ = pos_ptr->getNext();
    first_->setPrevious(nullptr);

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



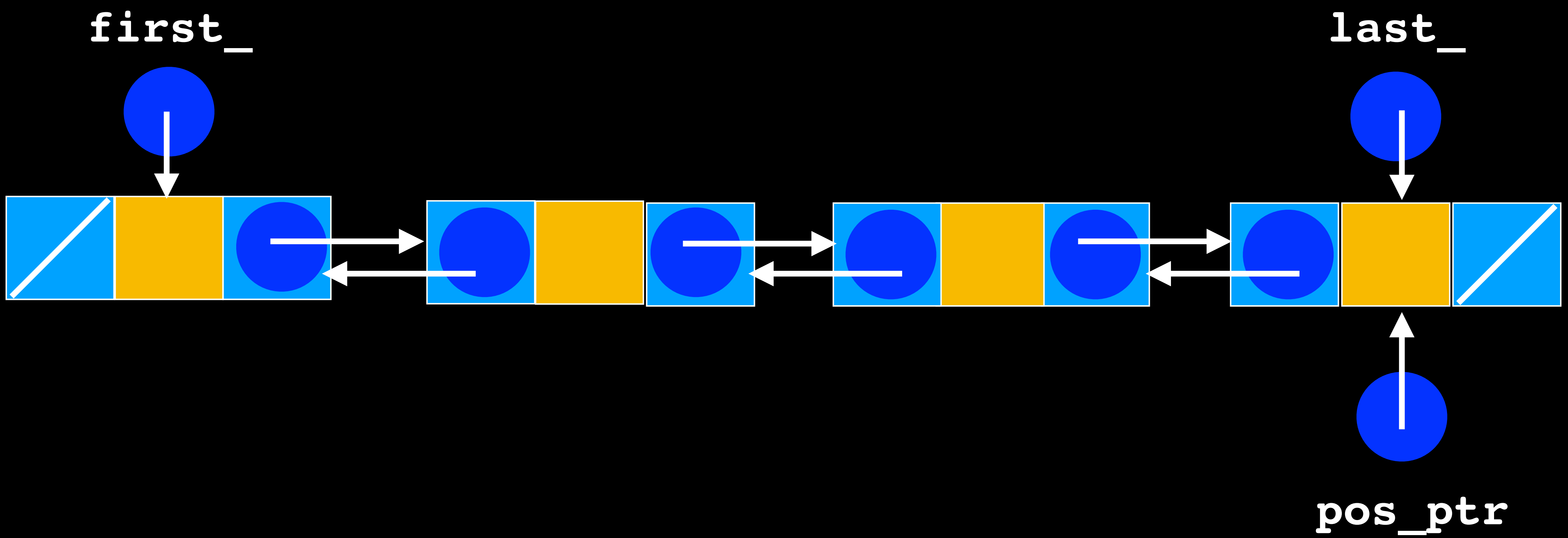
```
// Remove node from chain
if (pos_ptr == first_)
{
    // Remove first node
    first_ = pos_ptr->getNext();
    first_->setPrevious(nullptr);

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



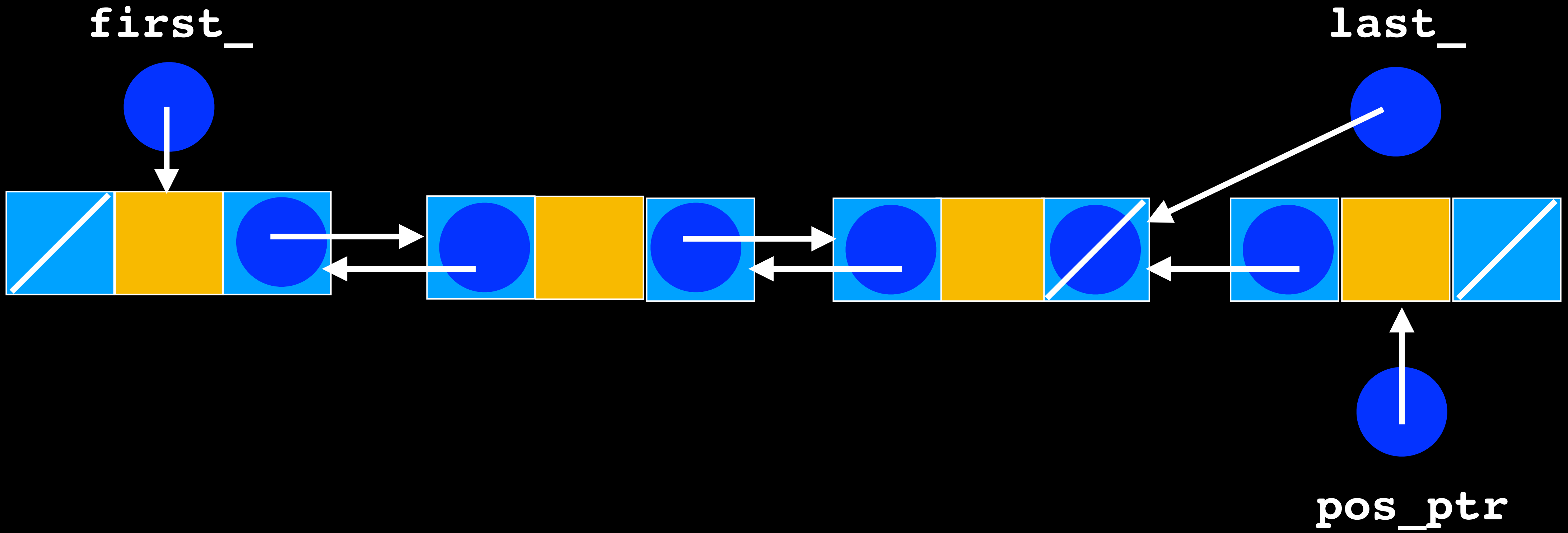
```
// Remove node from chain
if (pos_ptr == first_)
{
    // Remove first node
    first_ = pos_ptr->getNext();
    first_->setPrevious(nullptr);

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



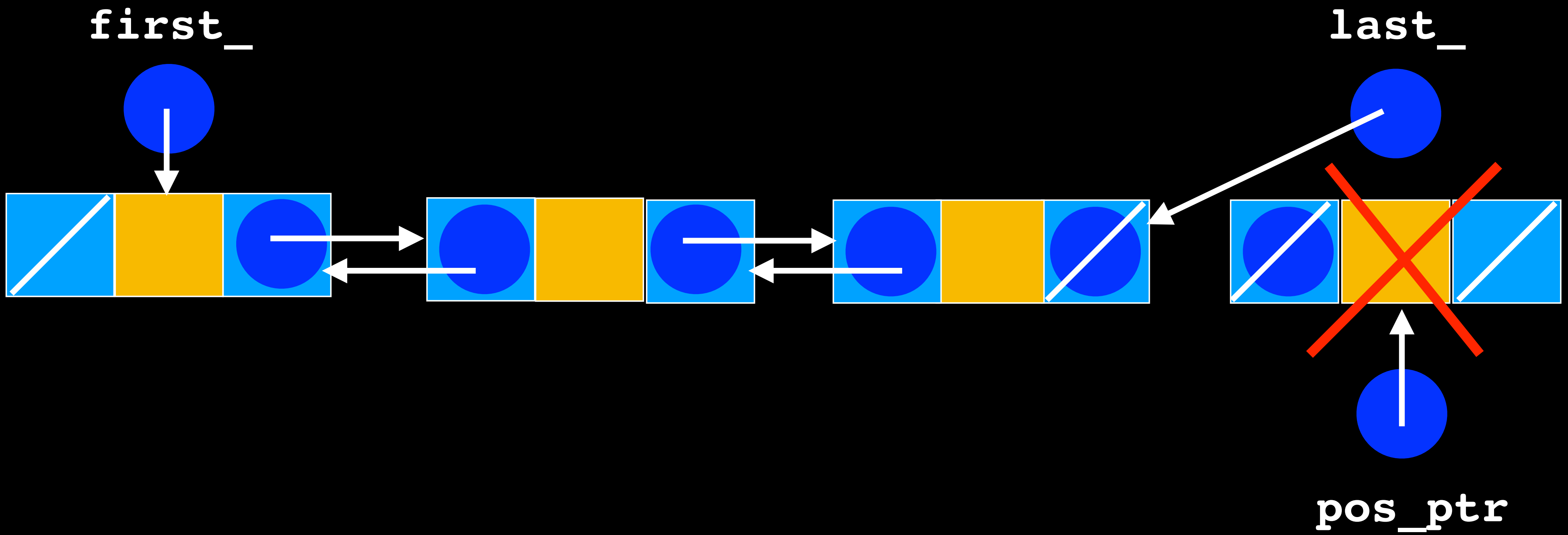
```
else if (pos_ptr == last_ )
{
    //remove last_  node
    last_  = pos_ptr->getPrevious();
    last_ ->setNext(nullptr);

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



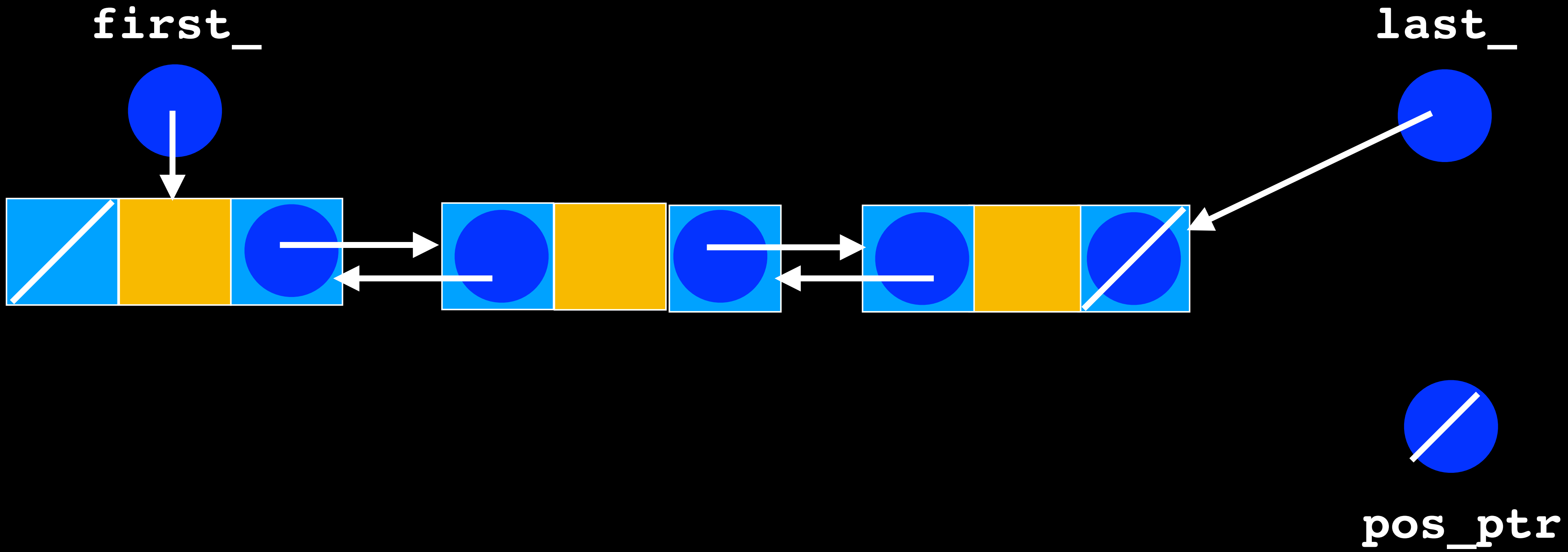
```
else if (pos_ptr == last_ )
{
    //remove last_  node
    last_  = pos_ptr->getPrevious();
    last_ ->setNext(nullptr);

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



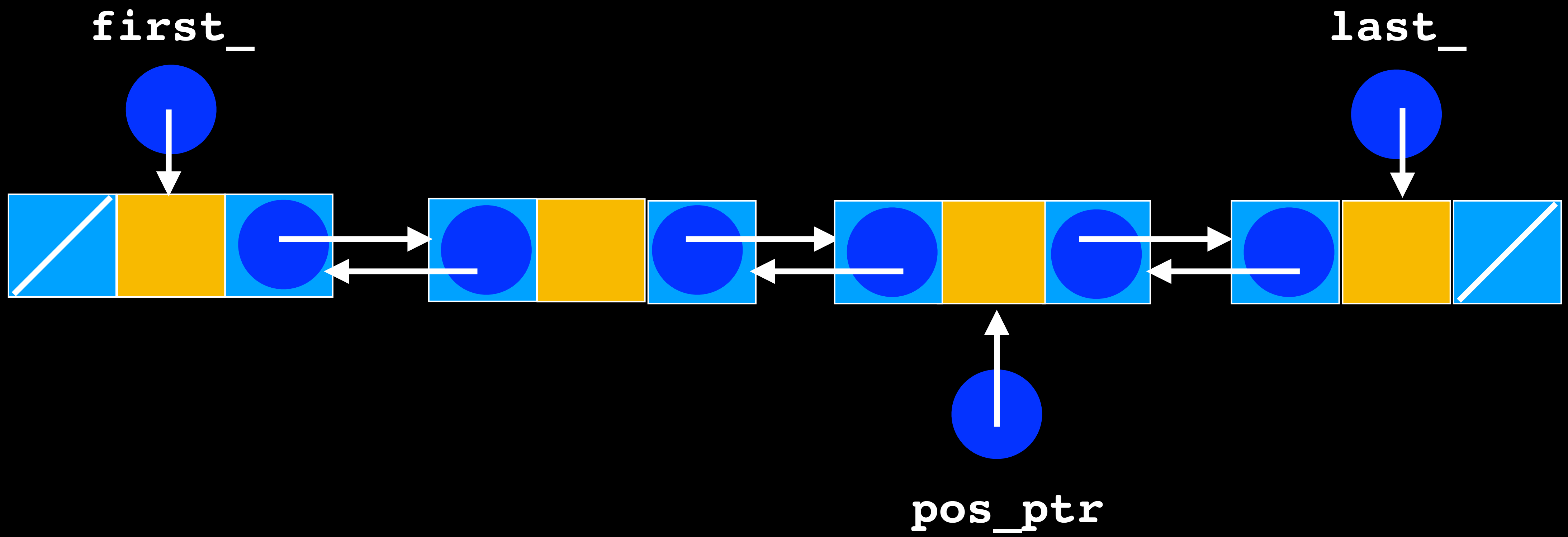
```
else if (pos_ptr == last_ )
{
    //remove last_  node
    last_  = pos_ptr->getPrevious();
    last_ ->setNext(nullptr);

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



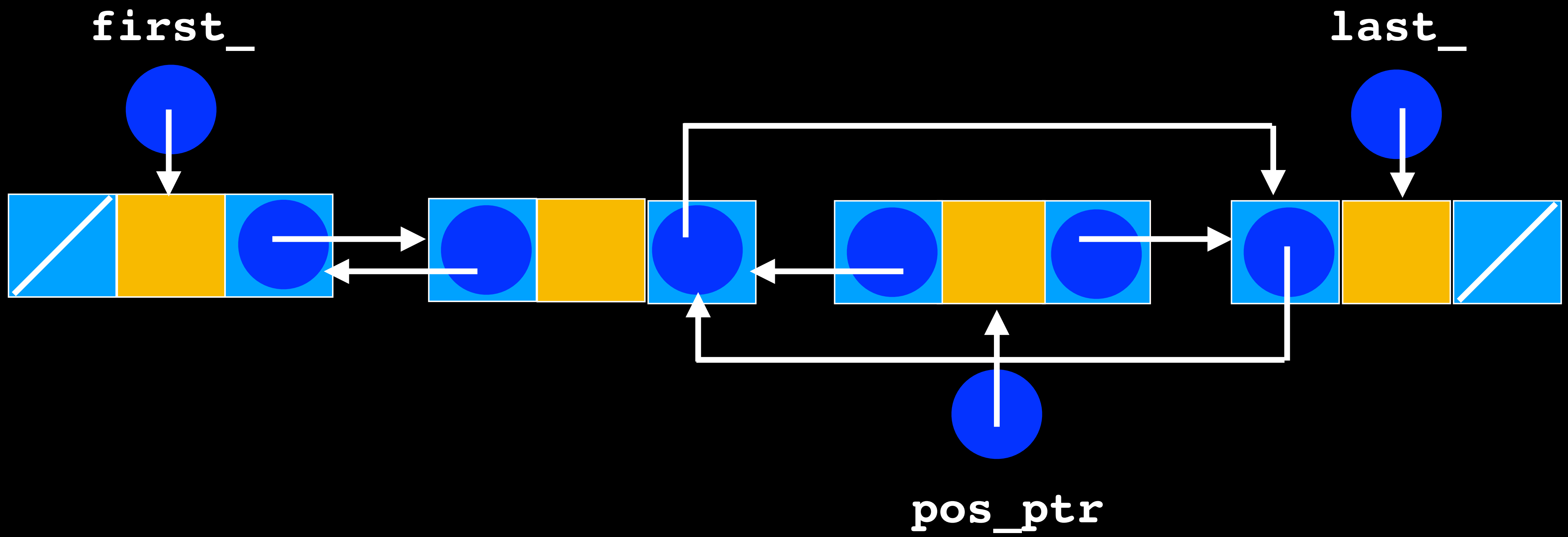
```
else if (pos_ptr == last_ )
{
    //remove last_  node
    last_  = pos_ptr->getPrevious();
    last_ ->setNext(nullptr);

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



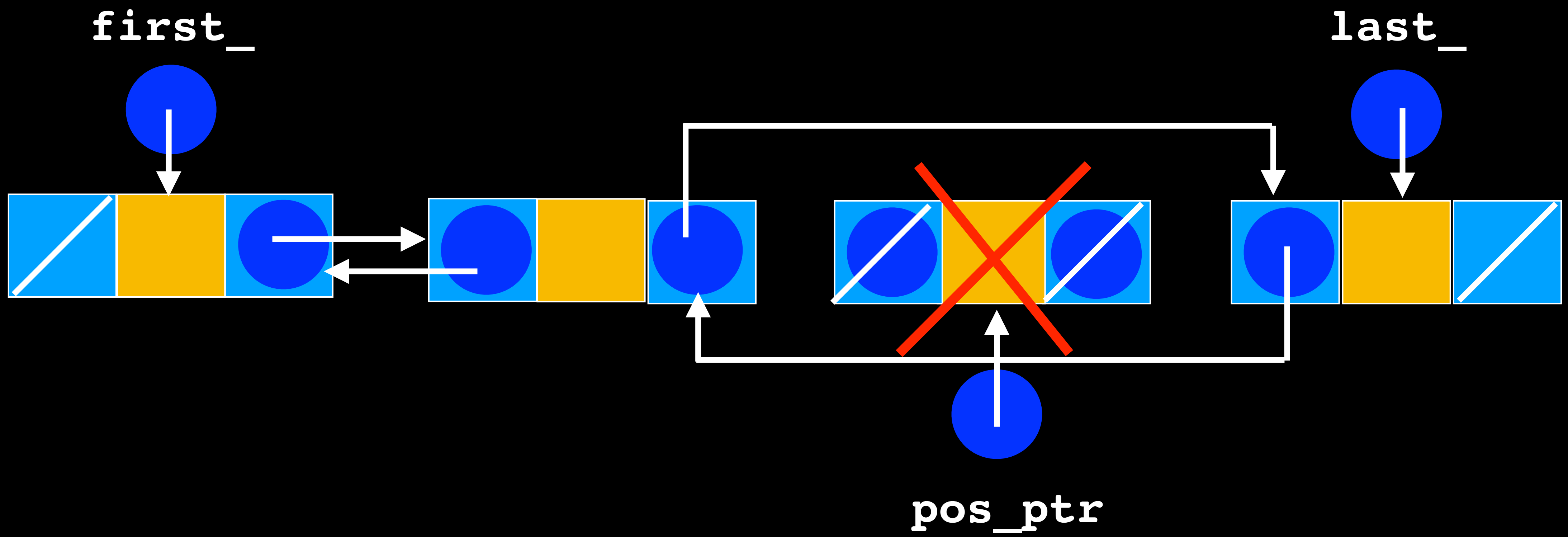
```
else if (pos_ptr != nullptr)
{
    //Remove from the middle
    pos_ptr->getPrevious()->setNext(pos_ptr->getNext());
    pos_ptr->getNext()->setPrevious(pos_ptr->getPrevious());

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



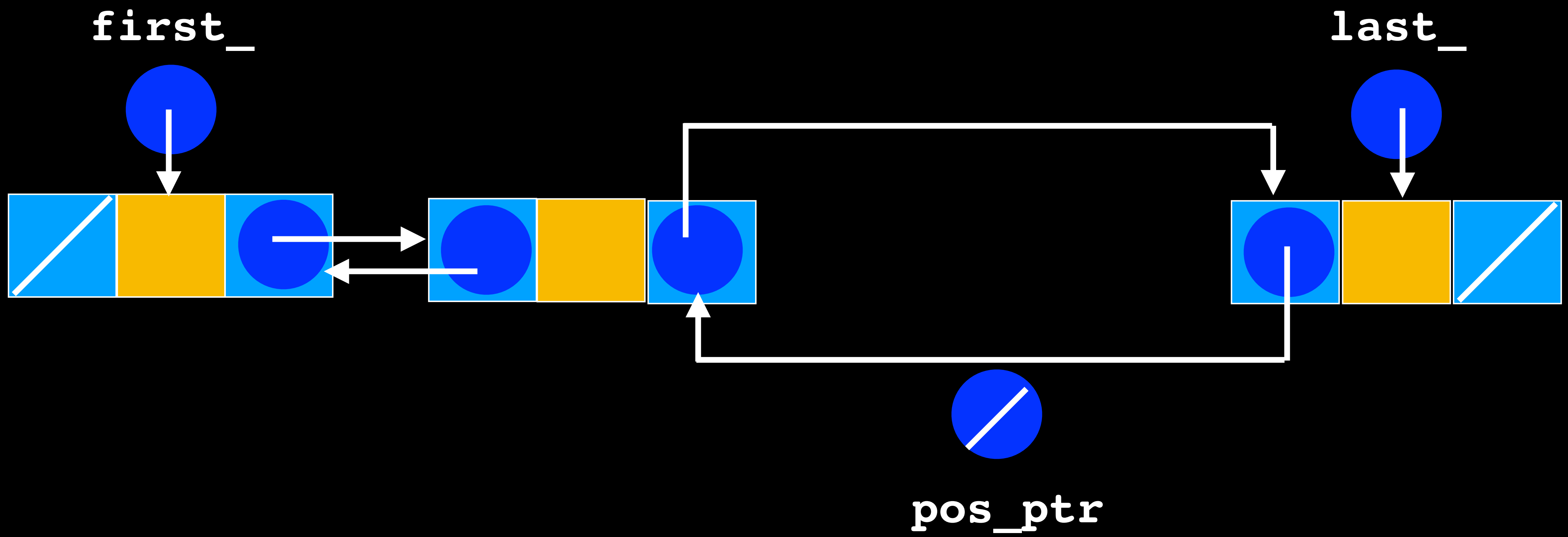
```
else if (pos_ptr != nullptr)
{
    //Remove from the middle
    pos_ptr->getPrevious()->setNext(pos_ptr->getNext());
    pos_ptr->getNext()->setPrevious(pos_ptr->getPrevious());

    // 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<class T>
Node<T>* List<T>::getPointerTo(size_t position) const
{

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

List::getItem

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

Problem: return type is T
There is no “default” or null valueto indicate uninitialized object