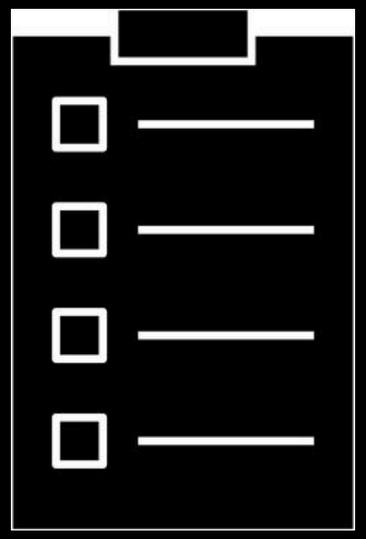
List



Tiziana Ligorio
Hunter College of The City University of New York

Today's Plan

Recap

List ADT

Recap

- Copy operations
  - Deep vs Shallow copy
  - Copy Constructor
    - invoked with `MyClass two = one;`
  - Assignment operator
    - invoked with `two = one;`

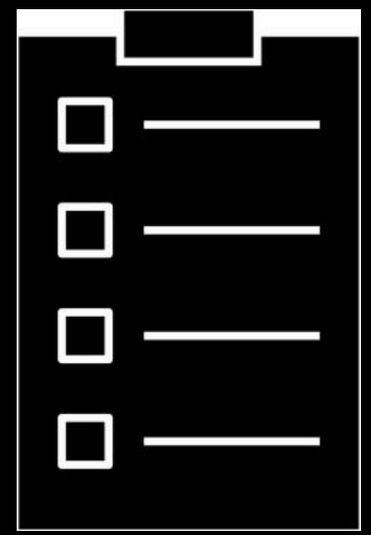
- Copy can be expensive and unnecessary when temporary

- Move operations
  - `std::move()` casts object as rvalue
  - If **rhs** can be cast as rvalue (don't care what happens to it next), move operations are more efficient.
  - Move constructor
    - invoked with `MyClass lvalue = rvalue;`
  - Move assignment operator
    - invoked with `lvalue = rvalue;`

List ADT

What makes a list?

E.g. Play**List**?

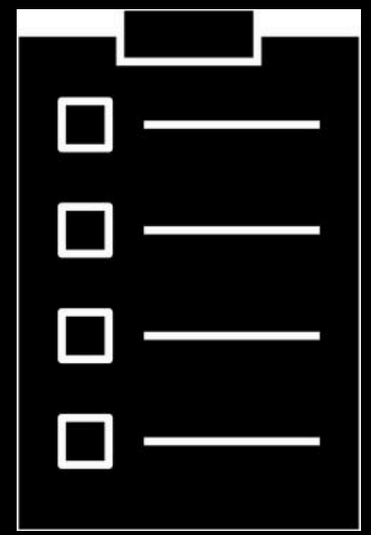


Duplicates allowed or not is not a defining factor

List ADT

What makes a list?

E.g. Play**List**?



Duplicates allowed or not is not a defining factor

ORDER!!!

Note

We will not consider Copy and Move operations in our ADTs going forward

A modern implementation should include them

The ideas are the same, so in lecture we don’t need to reconsider them each time

Your textbook does not include them (not C++ 11-20 updated)

```cpp
#ifndef LIST_H_
#define LIST_H_

template<class T>
class List
{

public:
    List(); // constructor
    List(const List<T>& a_list); // copy constructor
    ~List(); // destructor
    // assume copy and move operations
    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_
```

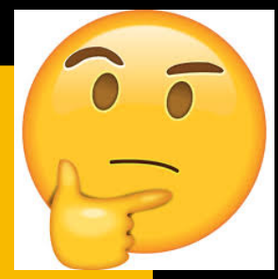
Defines the ADT

Used to represent the size of objects. Can think of it as unsigned it, but it is platform dependent.

Must choose implementation

Implementation

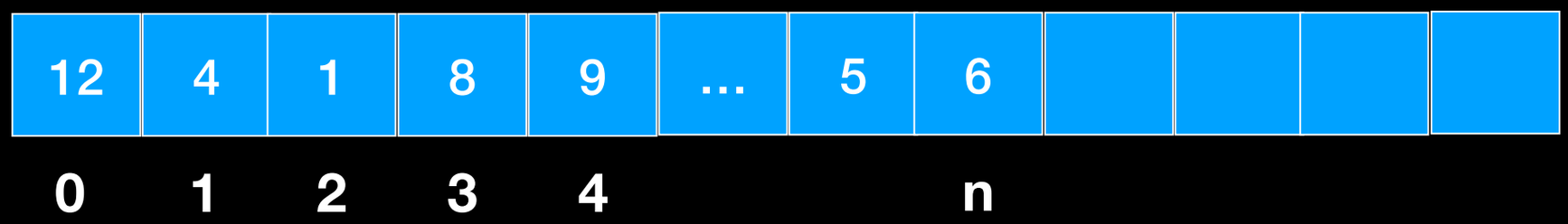
Must preserve order
No swapping tricks



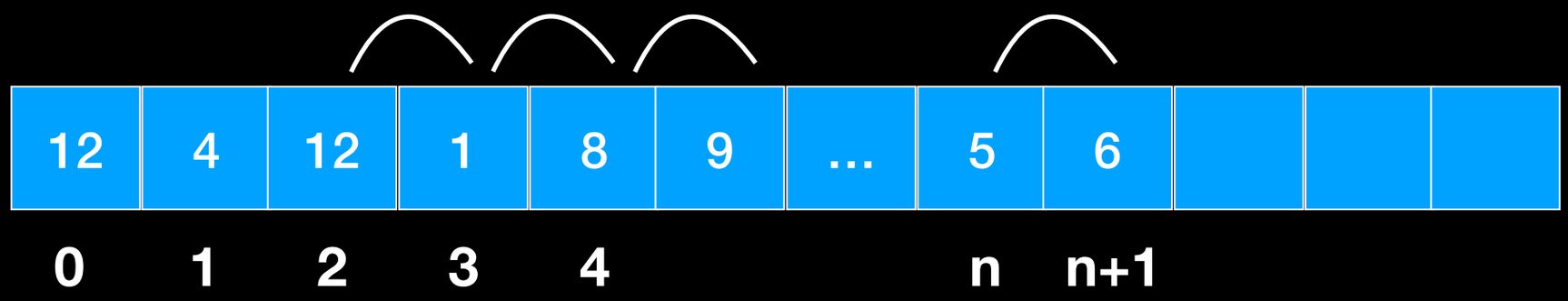
Array?

Linked Chain?

Array

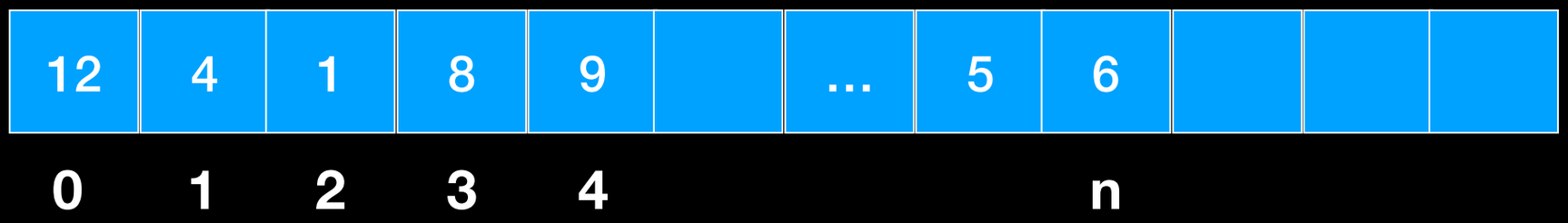



```
insert(2, 12)
```



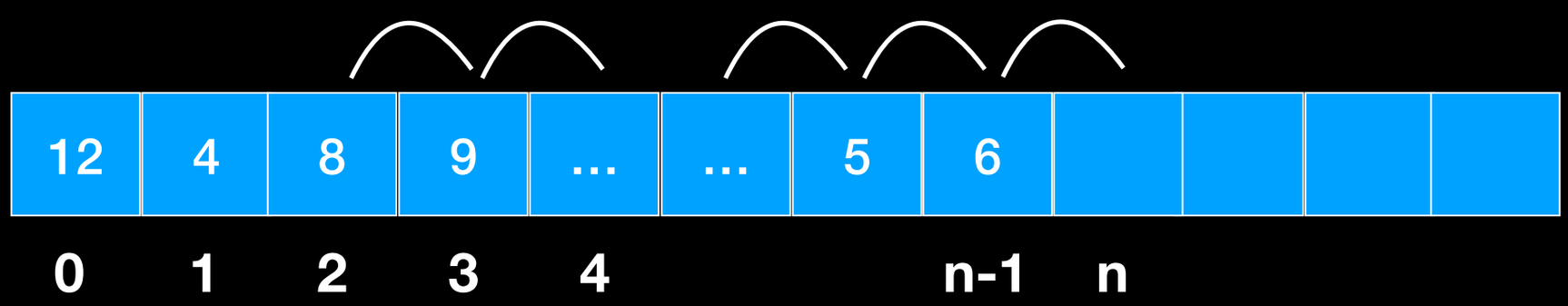
Must shift `n-(position+1)` elements

Array



`remove(2)`

similarly



Must shift `n-position` elements

Array Analysis

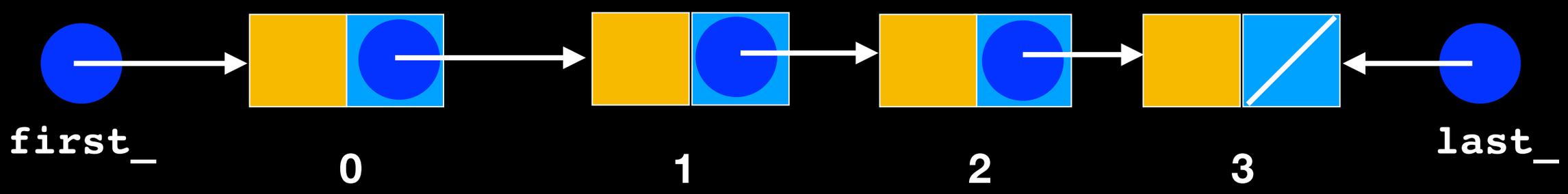
With Array both insert and remove are O(n)

Number of operations depends on size of List

Can we do better?

What makes a list?

Order is implied

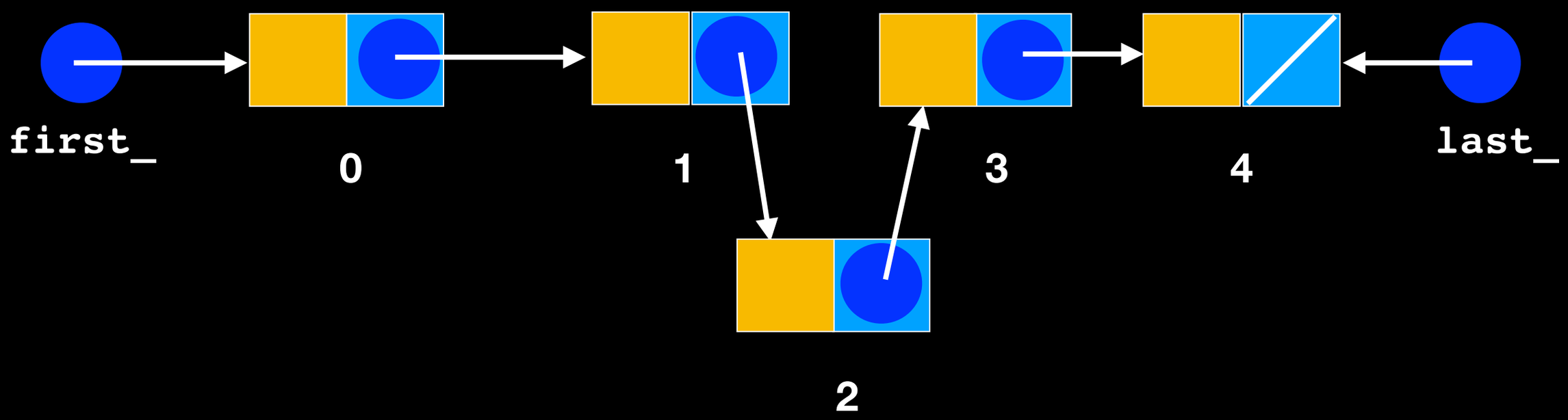


What makes a list?

Order is implied

Insertion and removal from middle retains order

No shifting necessary

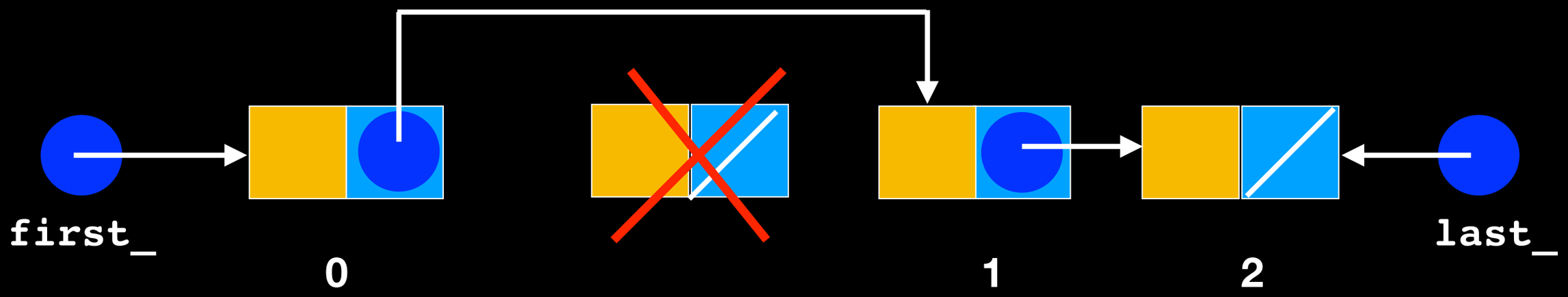


What makes a list?

Order is implied

Insertion and removal from middle retains order

No shifting necessary



What's the catch?

What's the catch?

No random access

As opposed to arrays or vectors with direct indexing

O(n): each insertion and removal must traverse `position+1` nodes

Here too, number of operations depends on size of List

O(1) Constant

O(n) 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		

O(1) Constant

O(n) 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		

O(1) Constant

O(n) 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		

✓ O(1) Constant

✗ O(n) depends on # of items

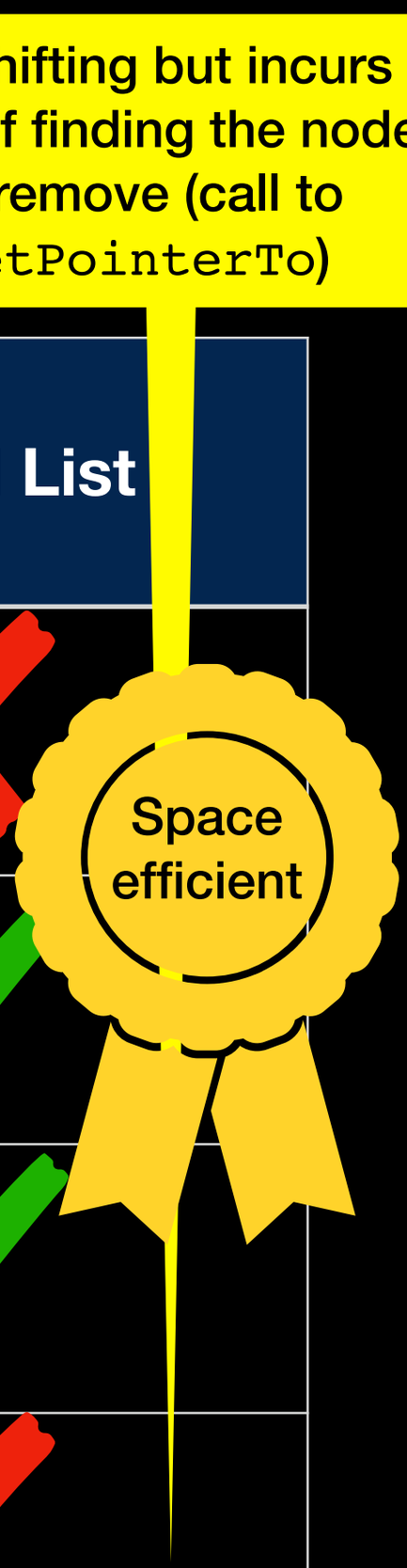
	Arrays	Linked List
Random/direct access	✓	✗
Retain order with Insert and remove At the back	✓	✓
Retain order with insert and remove at front	✗	✓
Retain order with insert and remove In the middle		

O(1) Constant

O(n) depends on # of items

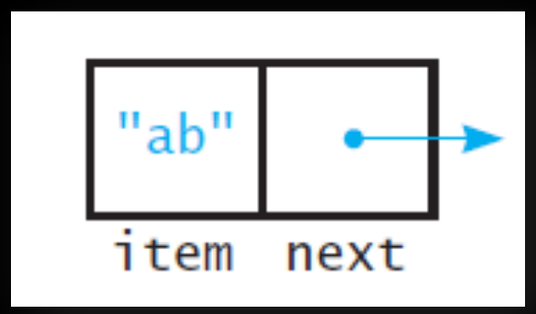
No shifting but incurs cost of finding the node to remove (call to `getPointerTo`)

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



Singly-Linked List

Use the same node as the LinkedBag

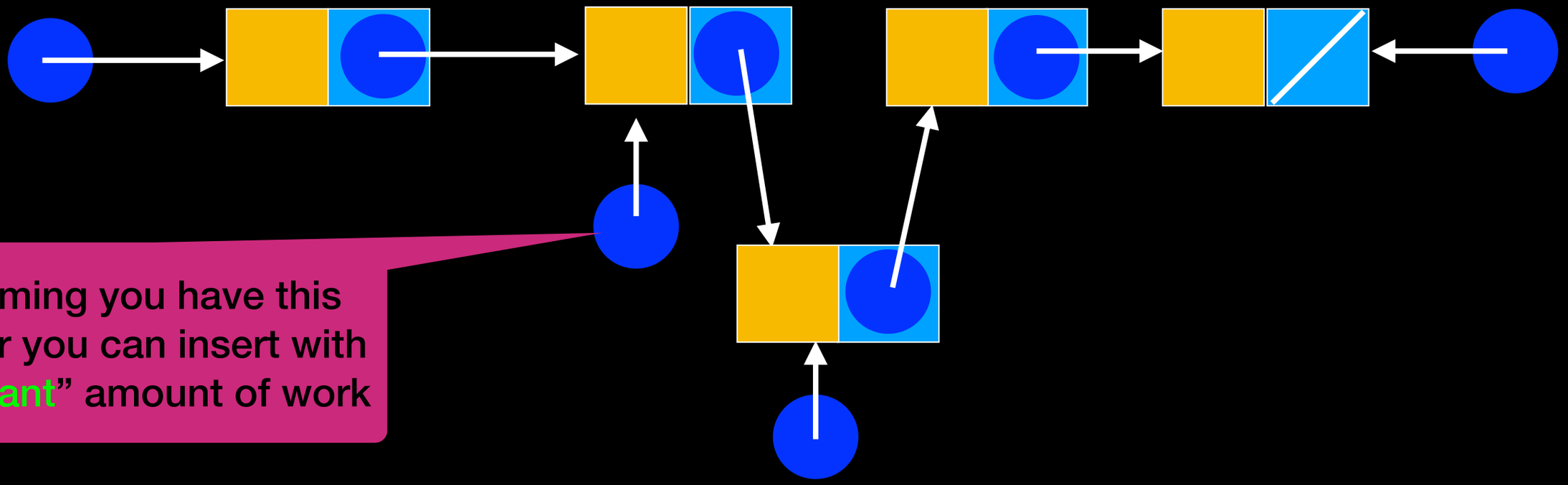


```
#ifndef LIST_H_
#define LIST_H_

#include “Node.hpp"

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:
     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_
```

Additional pointer to last node.

INSERT

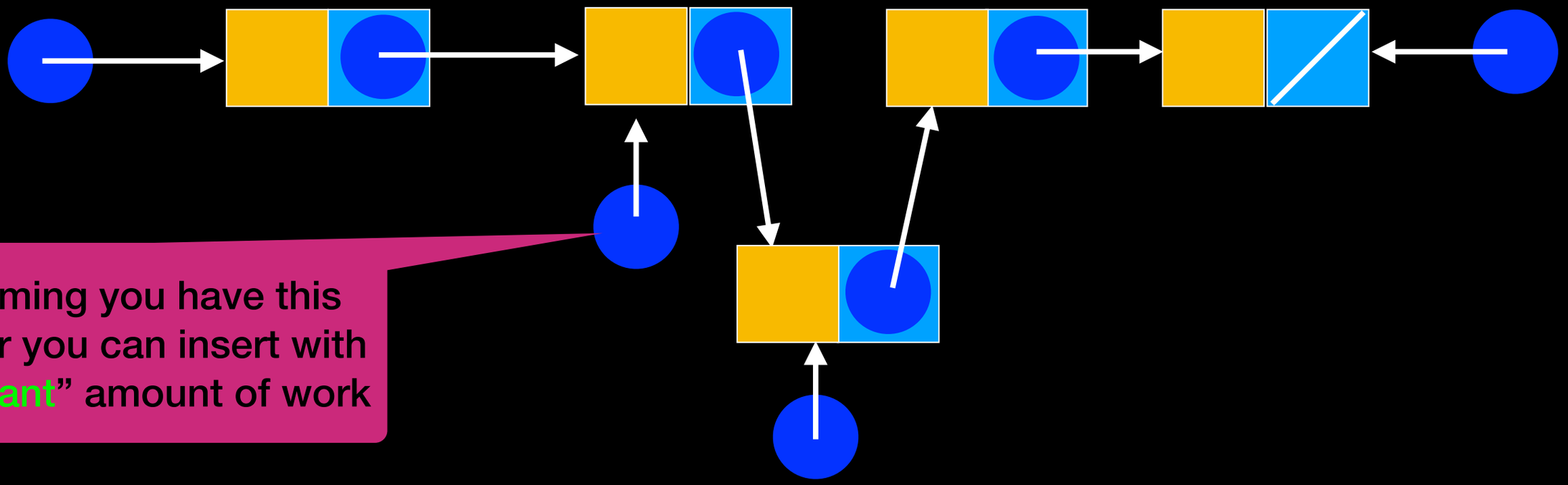
```
void insert(size_t position, T new_element);
```



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

INSERT

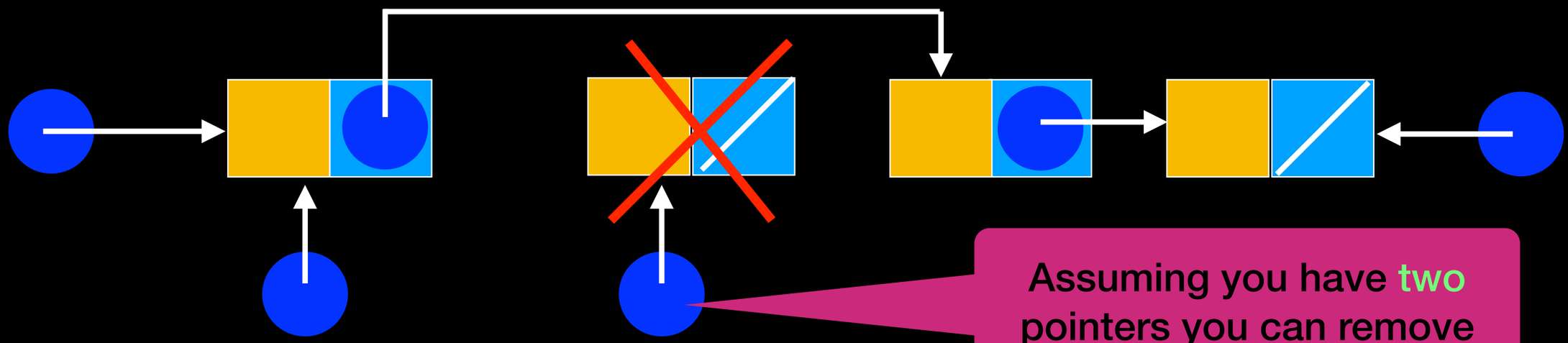
```
void insert(size_t position, T new_element);
```



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

REMOVE

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



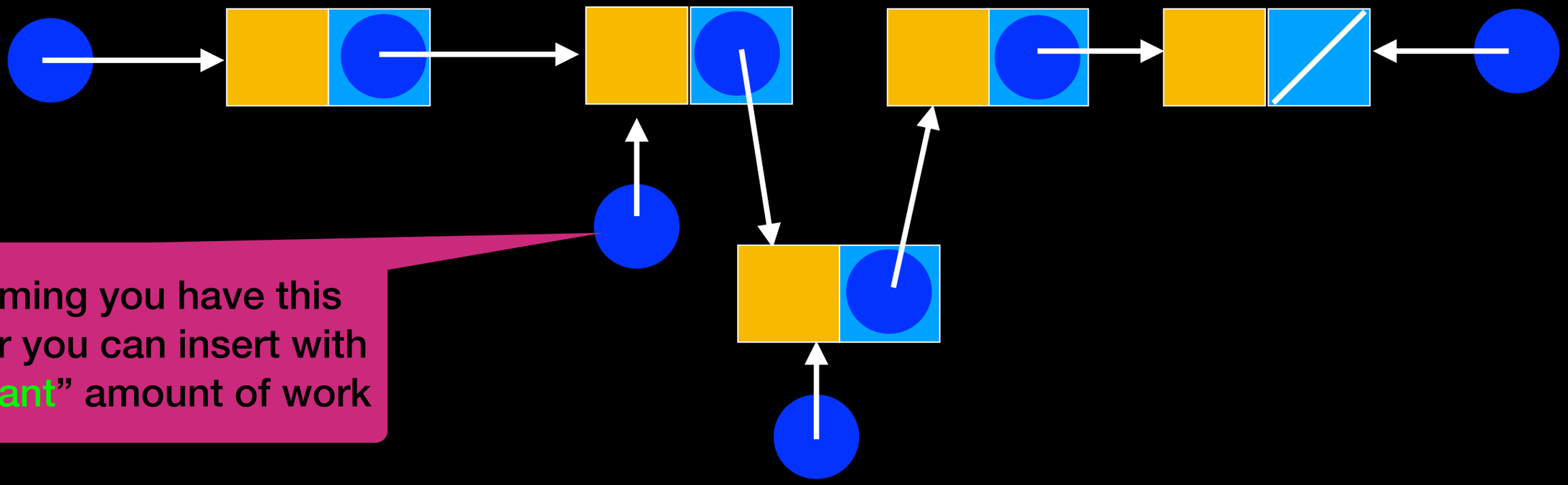
Assuming you have two pointers you can remove with “constant” amount of

Caveat

Find the pointer to the node before inserting/ removing —> traversal: O(n) - *depends on number of elements in list*

INSERT

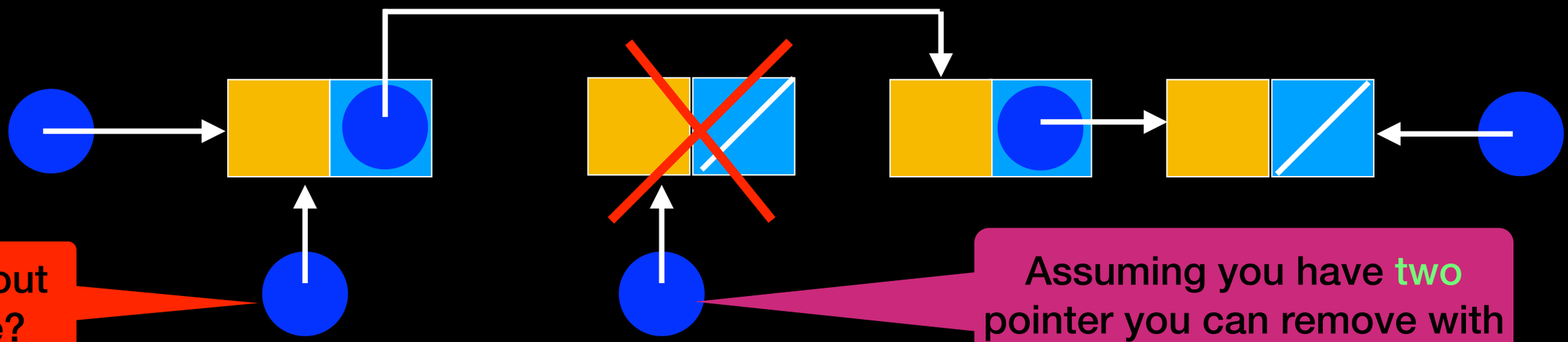
```
void insert(size_t position, T new_element);
```



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

REMOVE

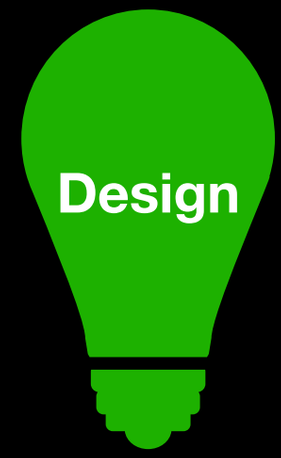
```
void remove(size_t position);
```



What about this one?

Assuming you have two pointer you can remove with “constant” amount of work

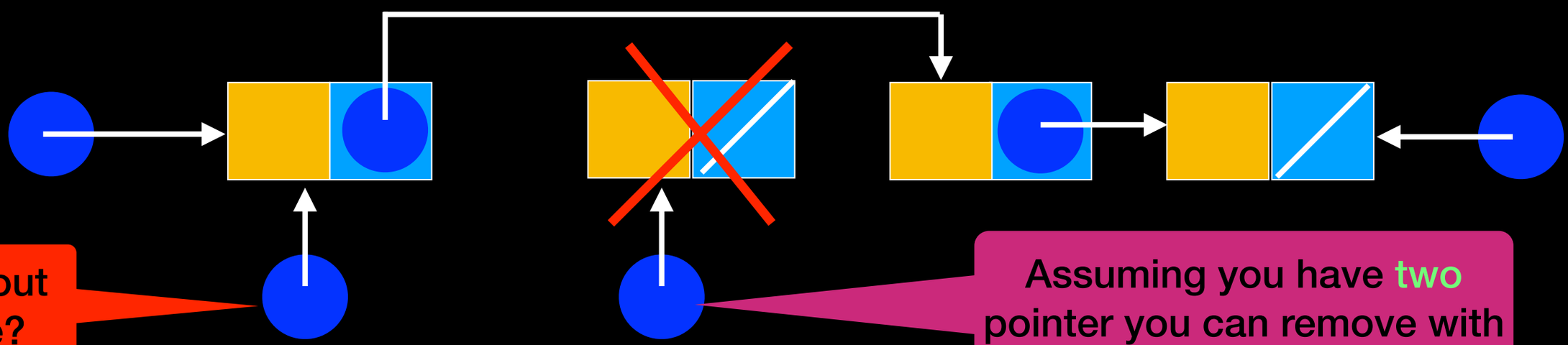
Lecture Activity



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 any node in the chain

REMOVE

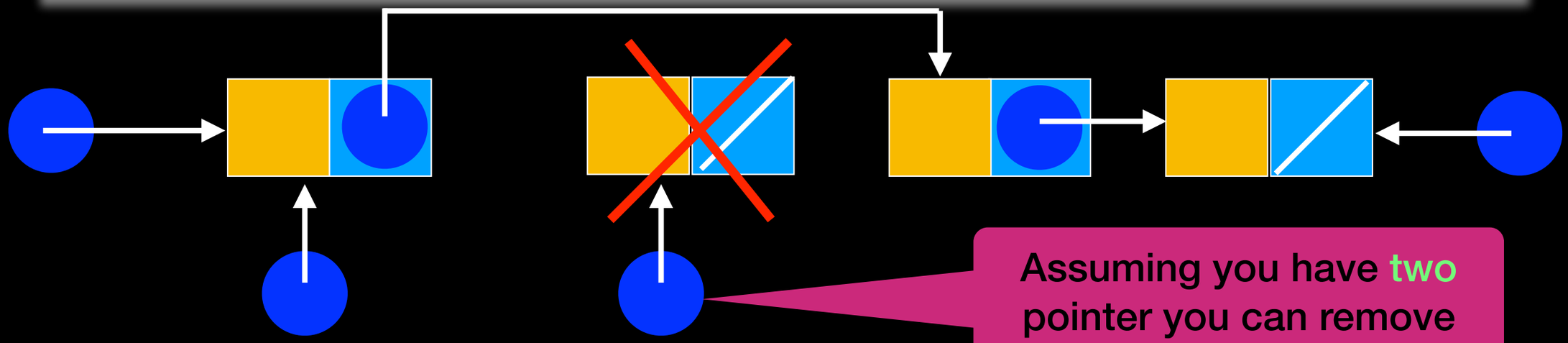
```
void remove(size_t position);
```



One possible solution:
Remove makes **two calls to** `getPointerTo`,
One with `position` and another with `position-1`

REMOVE

```
void remove(size_t position);
Node<T>* getPointerTo(size_t position);
```

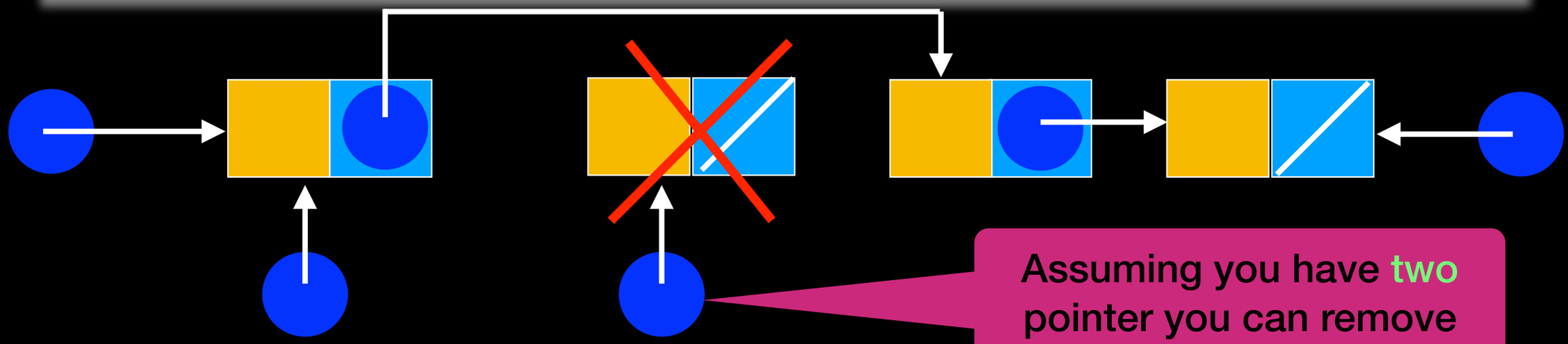


Assuming you have **two** pointer you can remove with “**constant**” amount of

Another possible solution:
Remove sets
`position-1 = position->getNext()`

REMOVE

```
void remove(size_t position);
Node<T>* getPointerTo(size_t position);
```

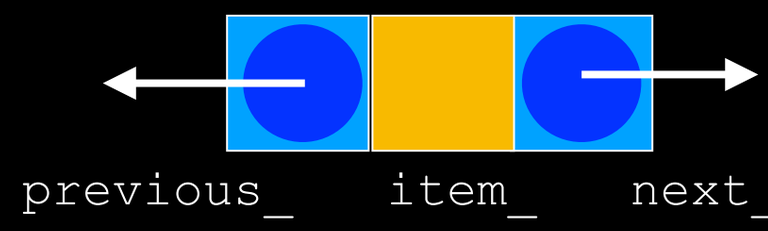


Assuming you have two pointer you can remove with “constant” amount of

Another Approach

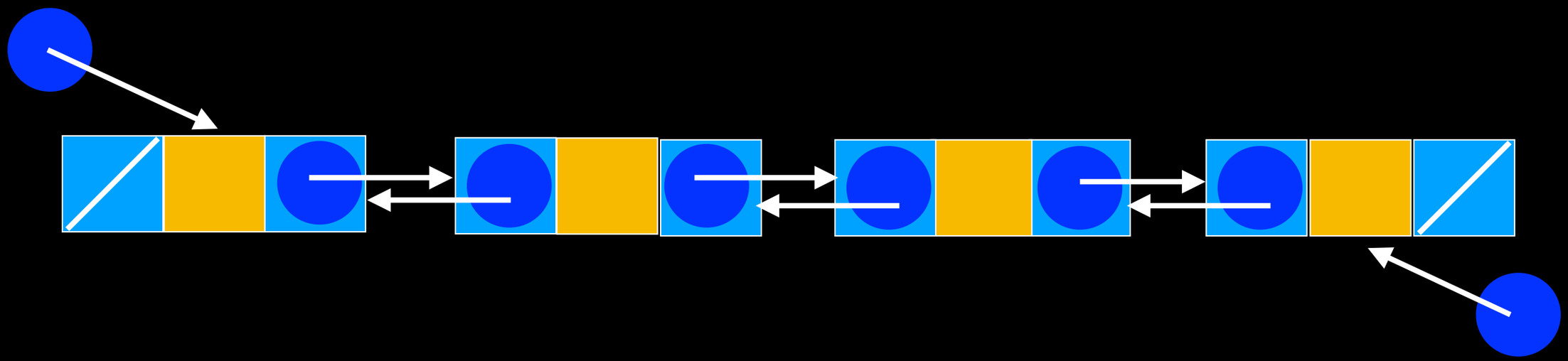
Doubly Linked List

New node with `previous_` and `next_` pointers



```
#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

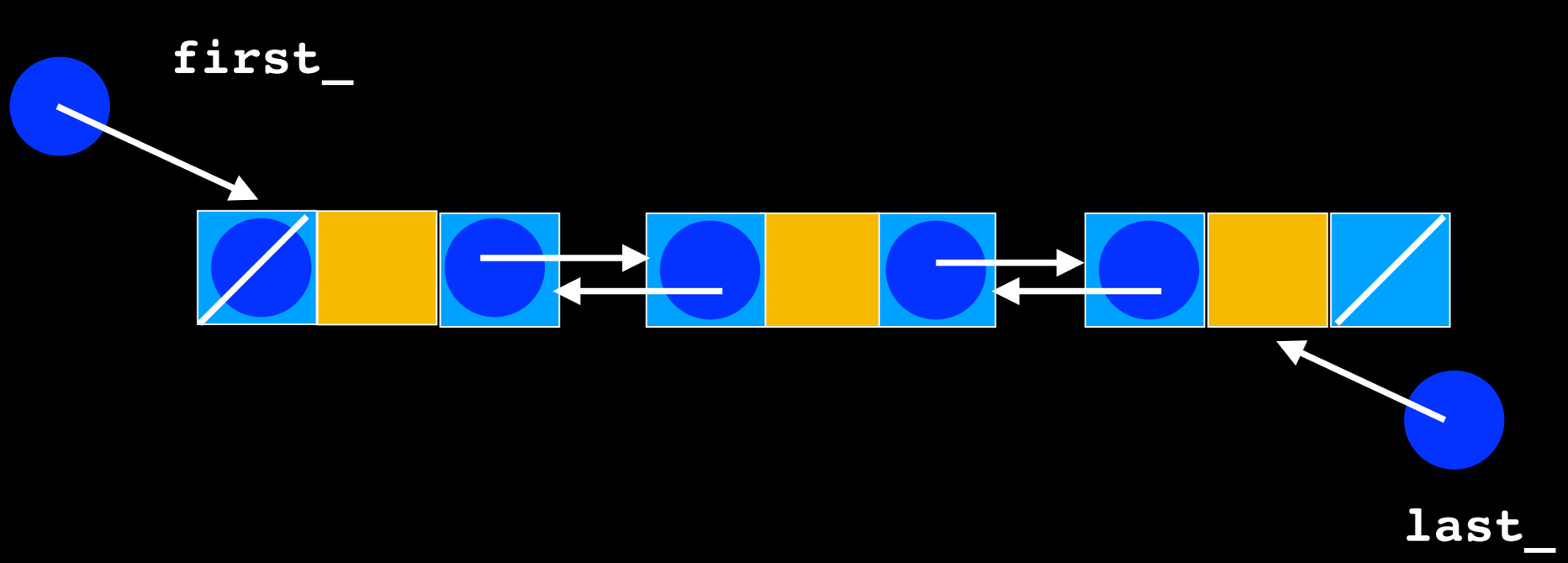


```cpp
#ifndef LIST_H_
#define LIST_H_

#include “Node.hpp"

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);//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_
```

Same interface as Singly-Linked list but uses different node

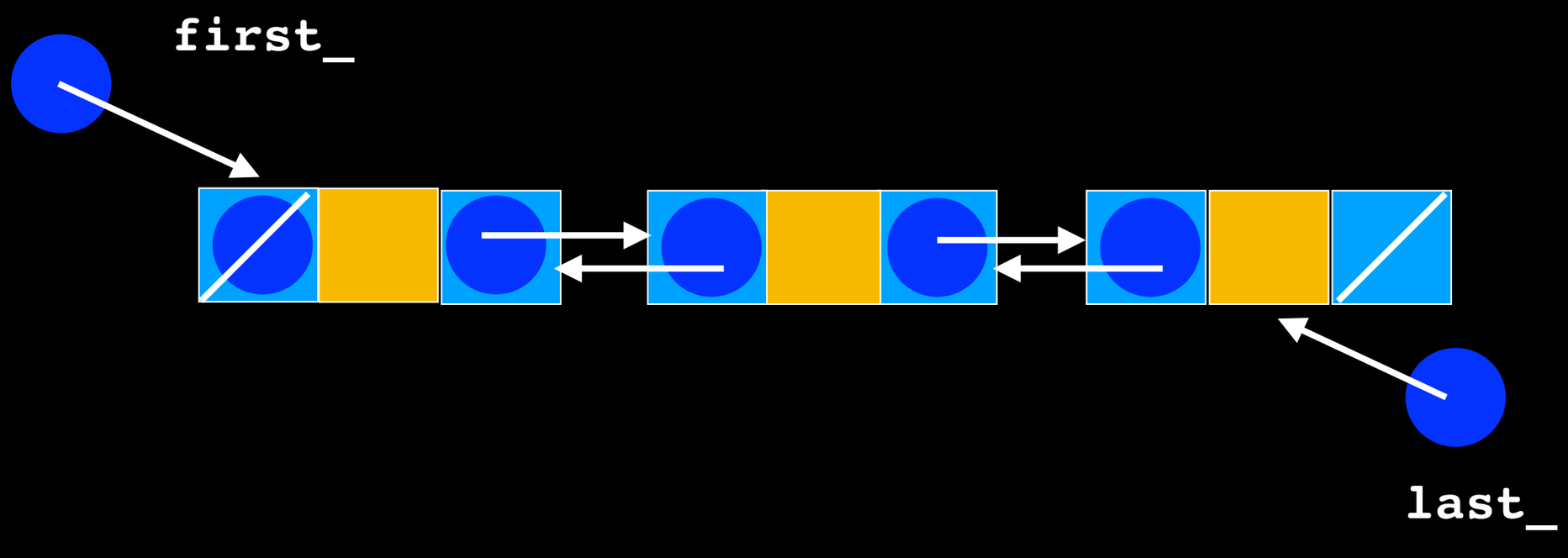
List::insert

What are the different cases that should be considered?

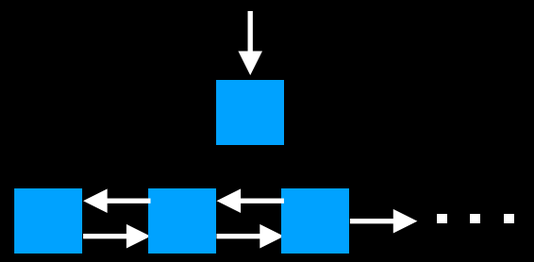


Lecture Activity

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



Pseudocode



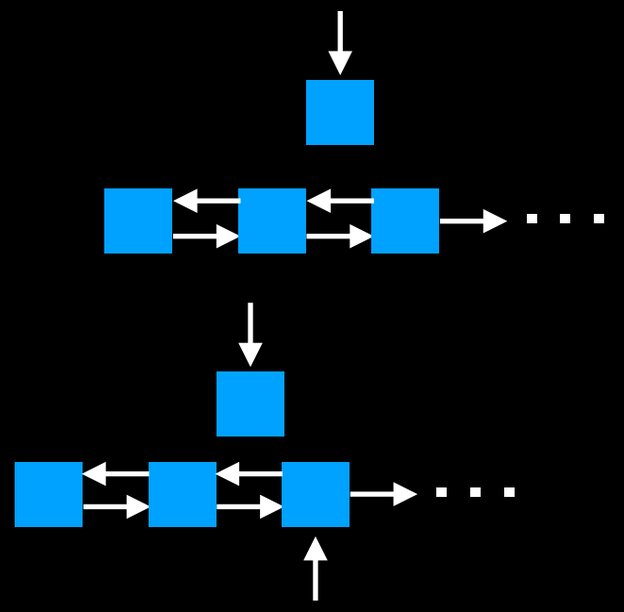
Instantiate new node

Obtain pointer

Connect new node to chain

Reconnect the relevant nodes

Pseudocode



Instantiate new node

Obtain pointer

Connect new node to chain

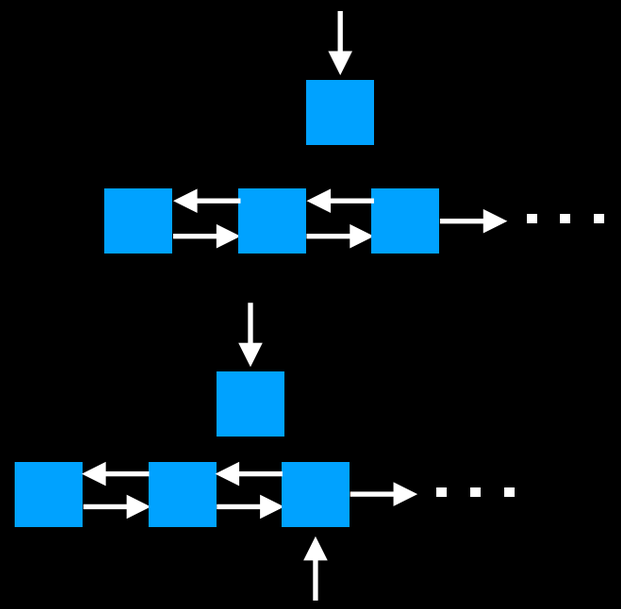
Reconnect the relevant nodes

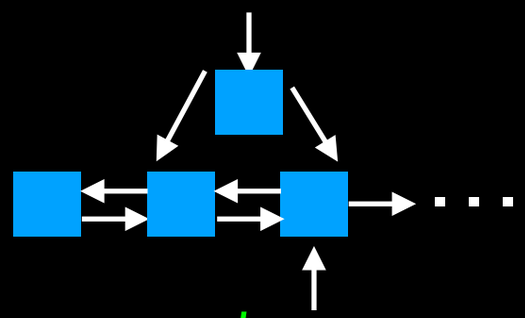
Pseudocode

Instantiate new node

Obtain pointer

Connect new node to chain

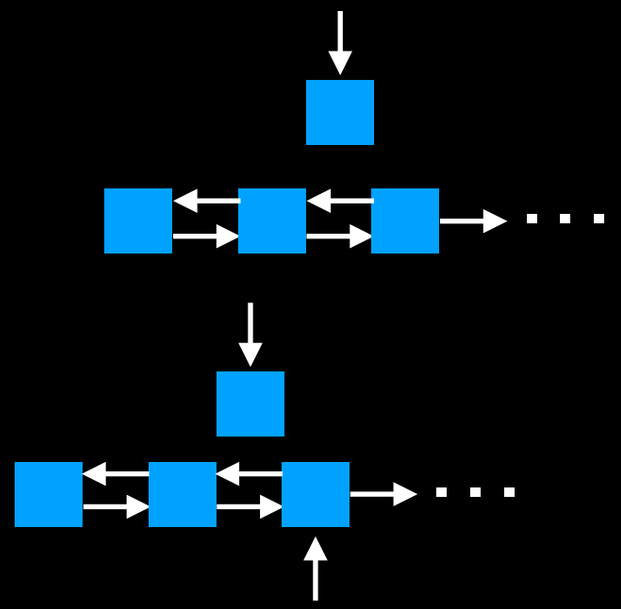




Reconnect the relevant nodes

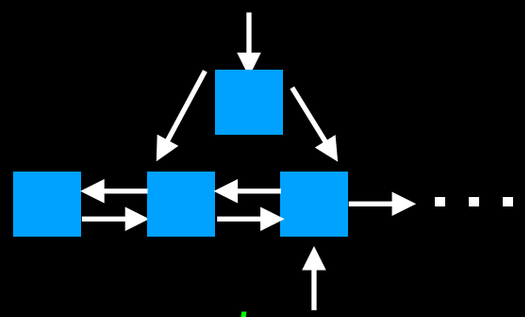
Pseudocode

Instantiate new node

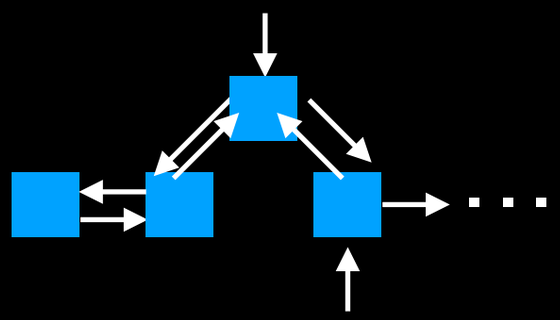


Obtain pointer

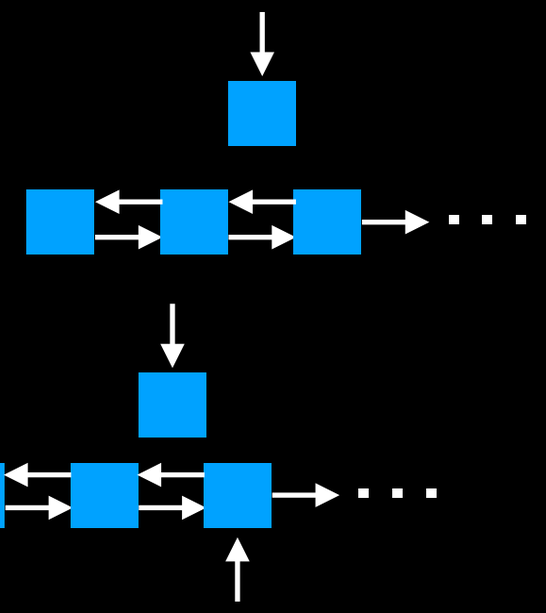
Connect new node to chain



Reconnect the relevant nodes



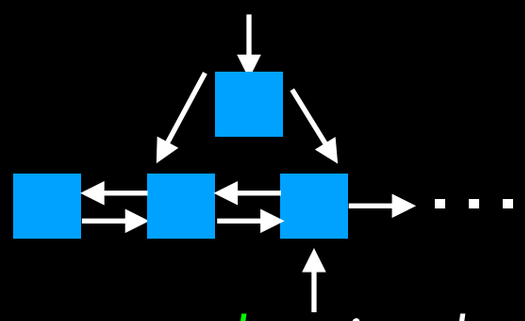
Pseudocode



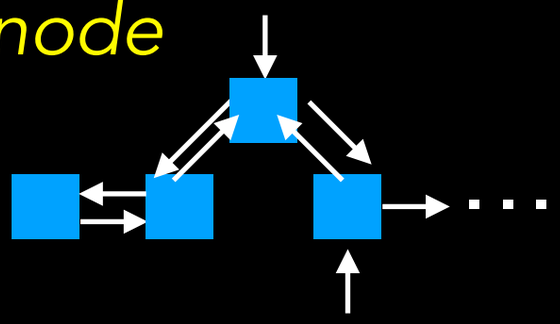
Instantiate new node to be inserted and set its value

Obtain pointer to node currently at position

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

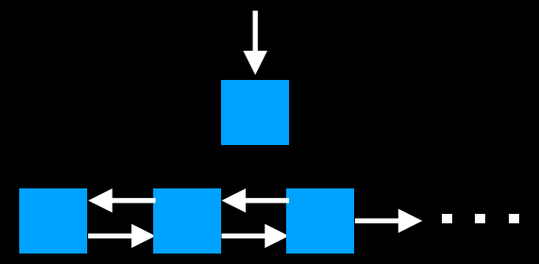


Reconnect the relevant nodes in the chain by pointing position->previous->next to the new node and position->previous to the new node



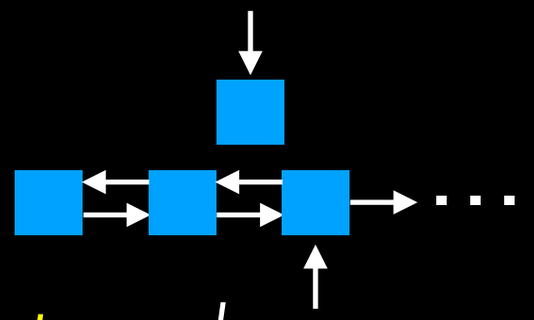
Order Matters!

More Pseudocodey

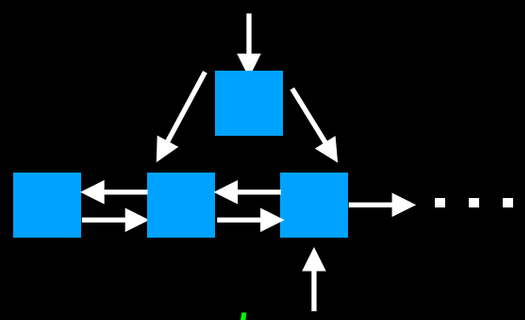


Instantiate new node new_ptr = new Node() and new_ptr->setItem()

Obtain pointer position_ptr = getPointerTo(position)

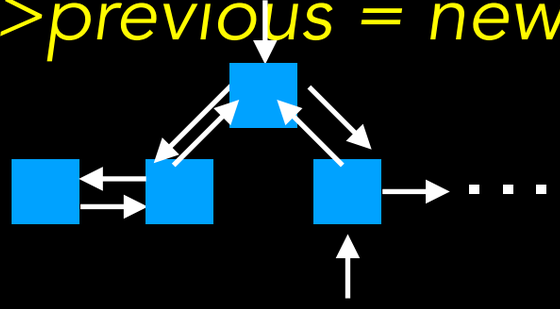


Connect new node to chain new_ptr->next = position_ptr and new_ptr->previous = temp->previous



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

   2  else if (pos_ptr == first_)
      {
         // Insert new node at beginning of list
         new_node_ptr->setNext(first_);
         new_node_ptr->setPrevious(nullptr);
         first_->setPrevious(new_node_ptr);
         first_ = new_node_ptr;
      }
   3  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;
      }
   4  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
```

O(n)

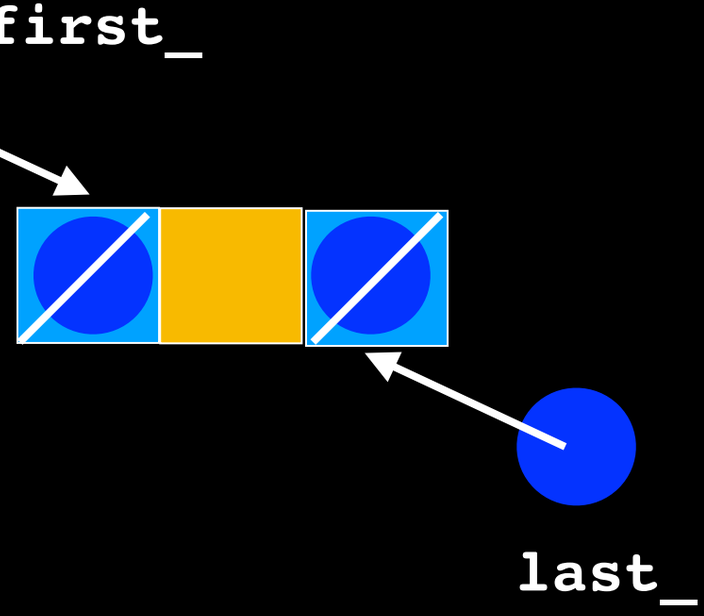
```
if (first_ == nullptr)
 {
      // Insert first node
1     new_node_ptr->setNext(nullptr);
      new_node_ptr->setPrevious(nullptr);
      first_ = new_node_ptr;
      last_ = new_node_ptr;
 }
```

4 O(1) cases

Always insert

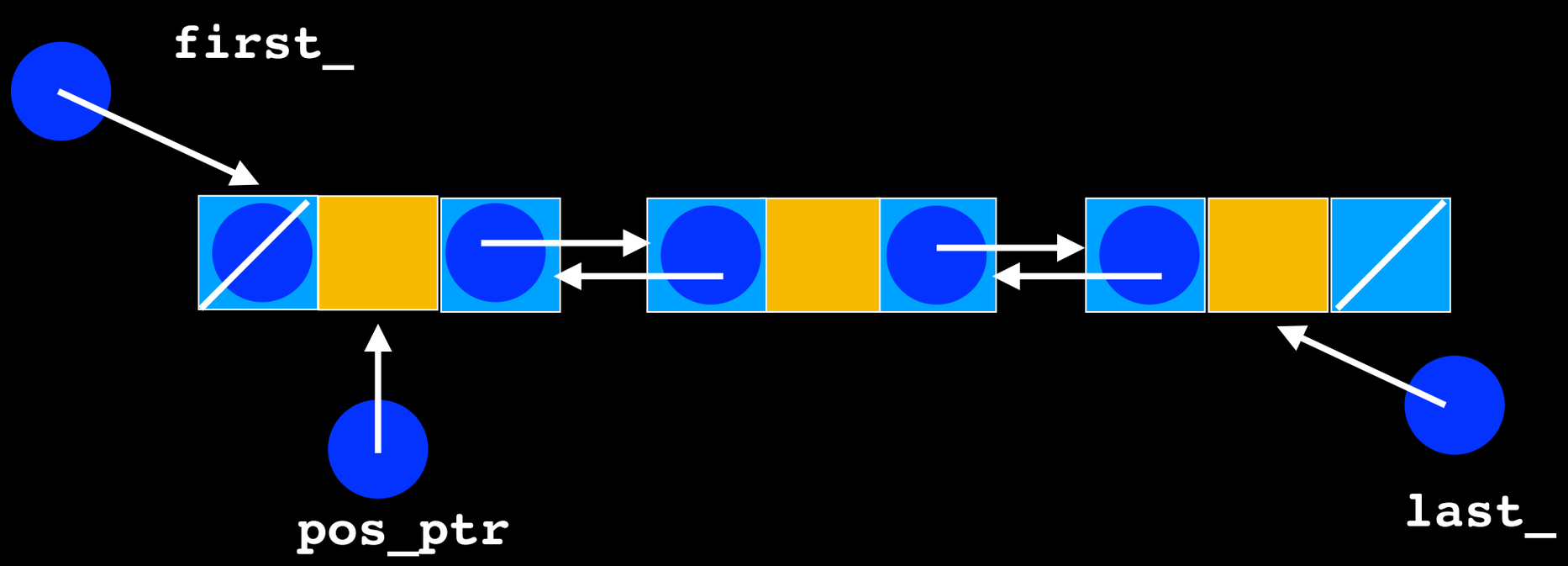
1

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



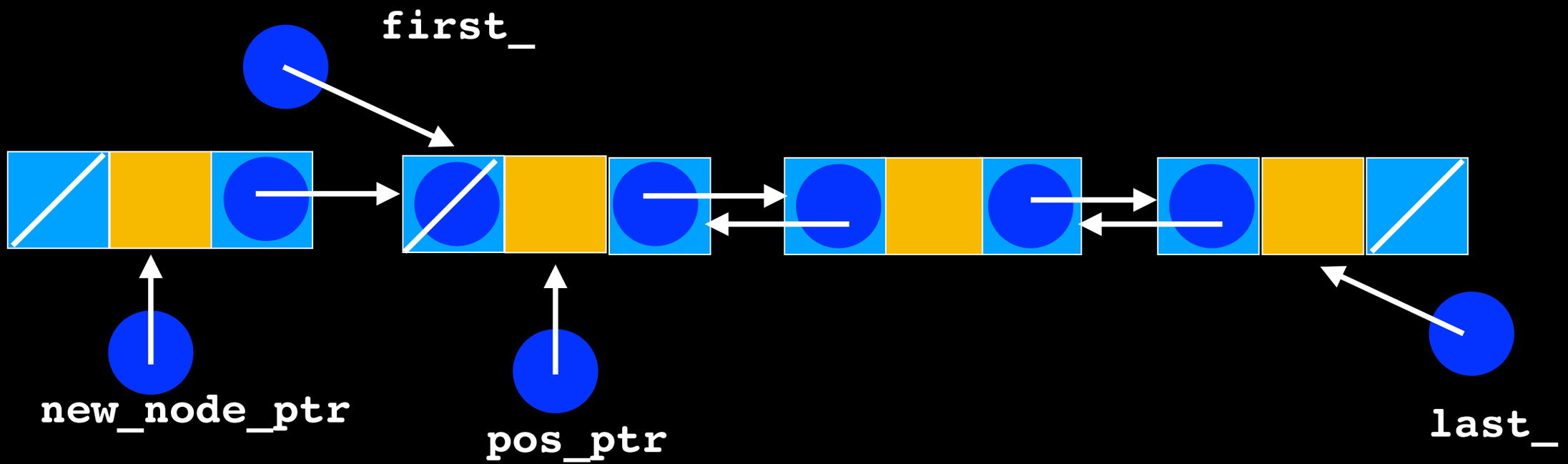
2

```
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;
        }
```



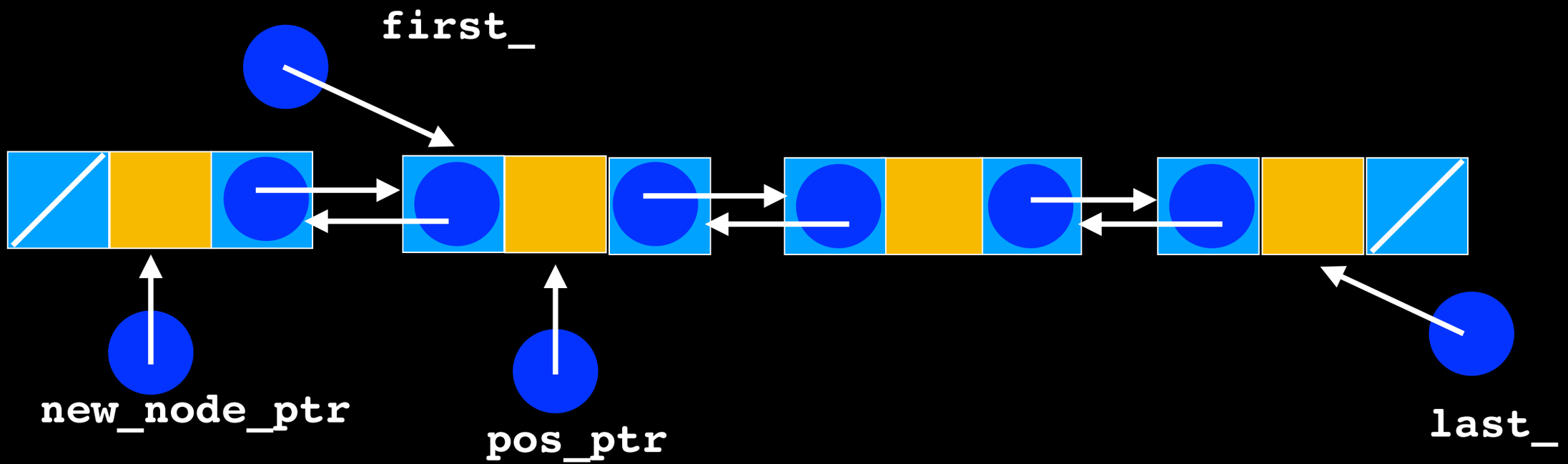
2

```
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;
        }
```



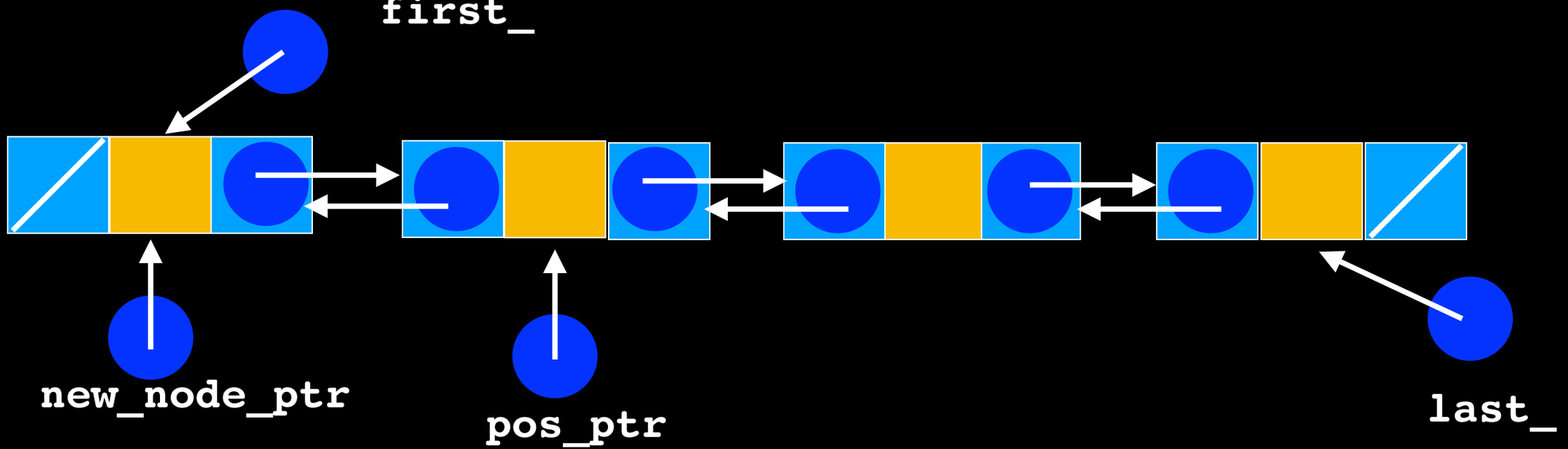
2

```
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;
        }
```



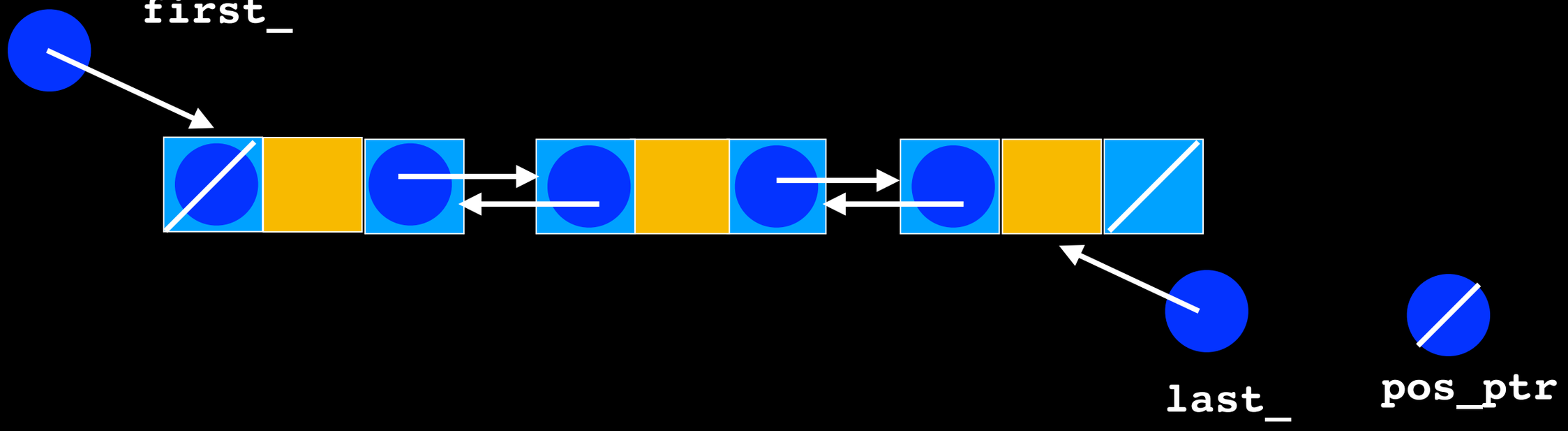
2

```
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;
        }
```



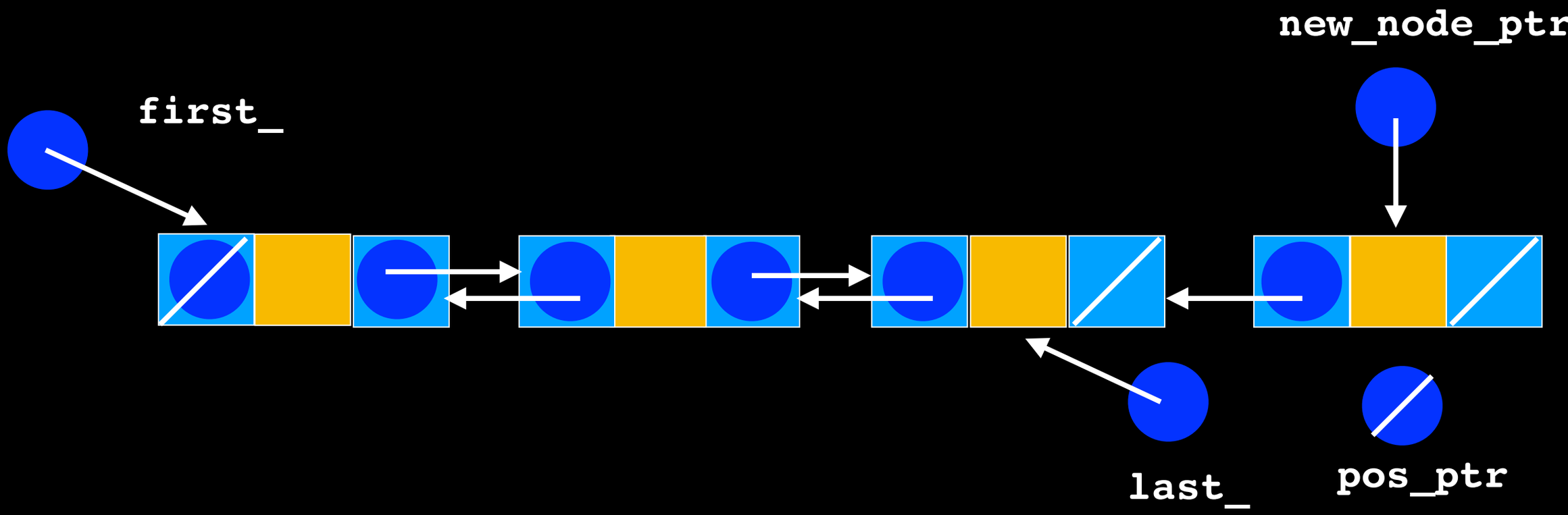
3

```
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;
}
```

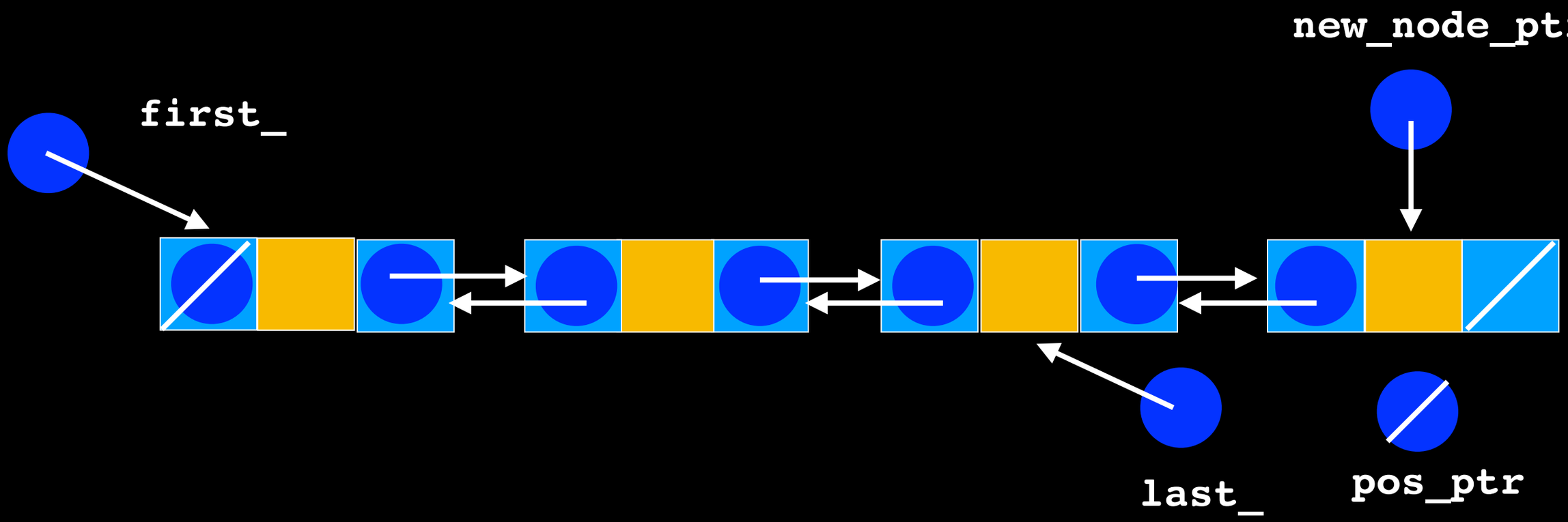


3

```
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;
}
```

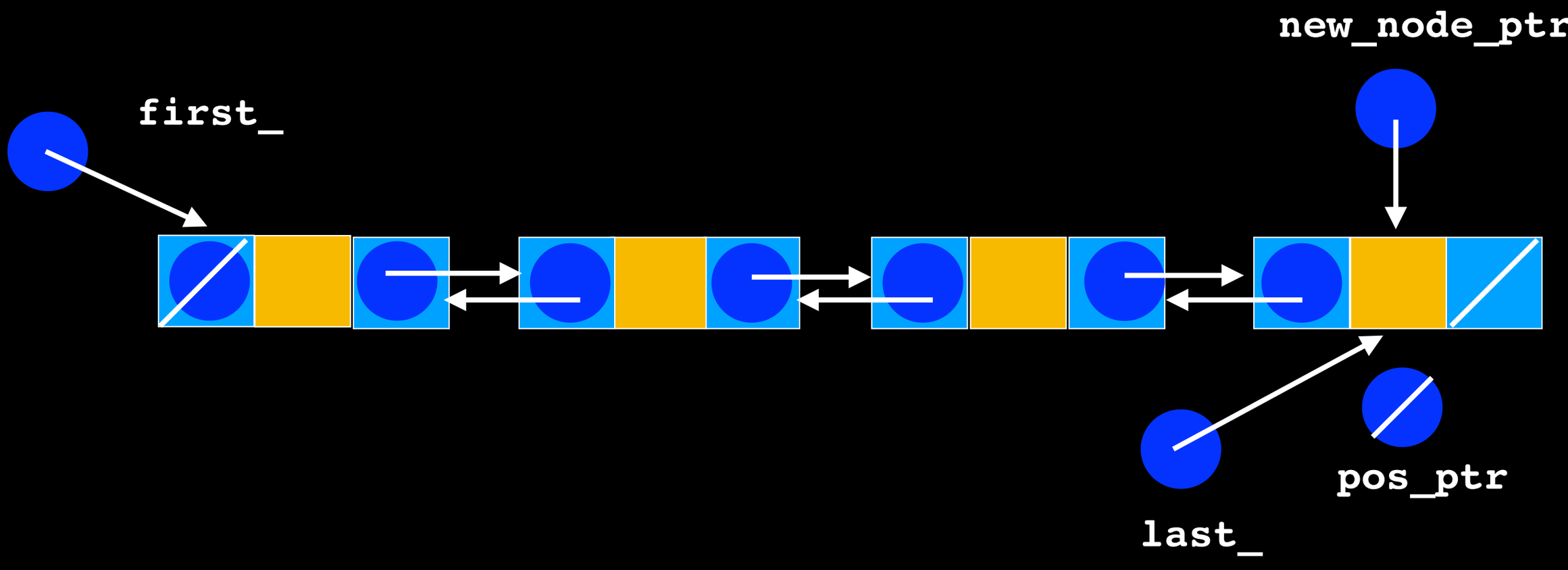


```
3  else if (pos_ptr == nullptr)
   {
       //insert at end of list
       new_node_ptr->setNext(nullptr);
       new_node_ptr->setPrevious(last_);
       last_->setNext(new_node_ptr);
       last_ = new_node_ptr;
   }
```

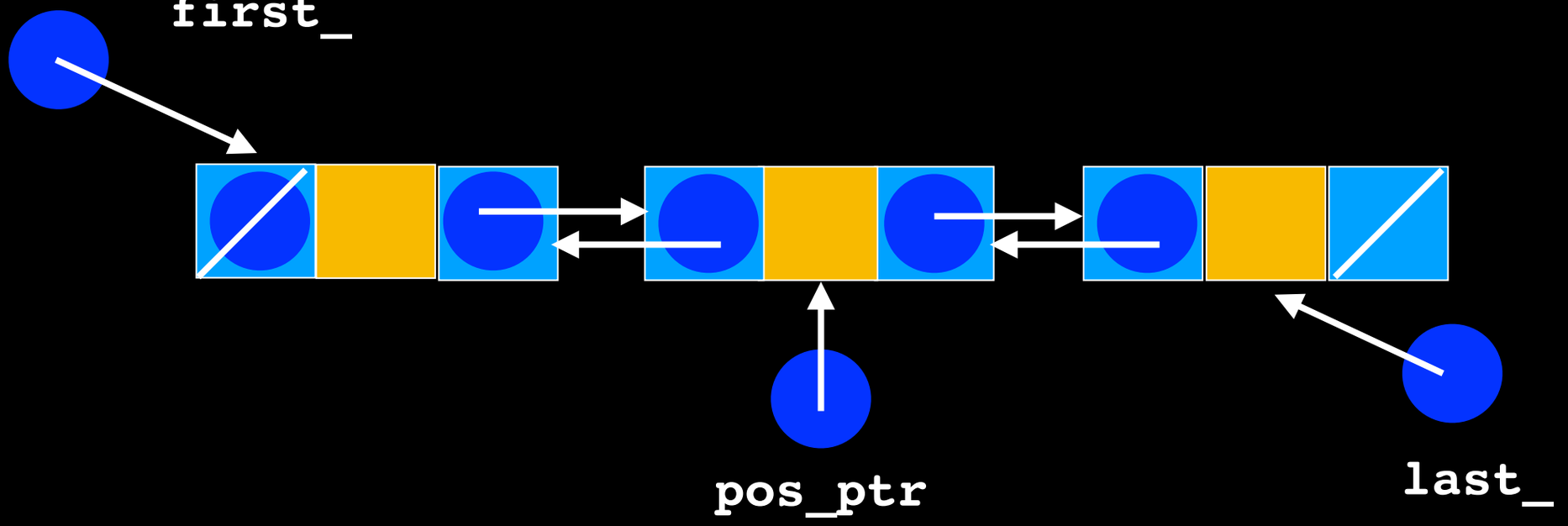


3

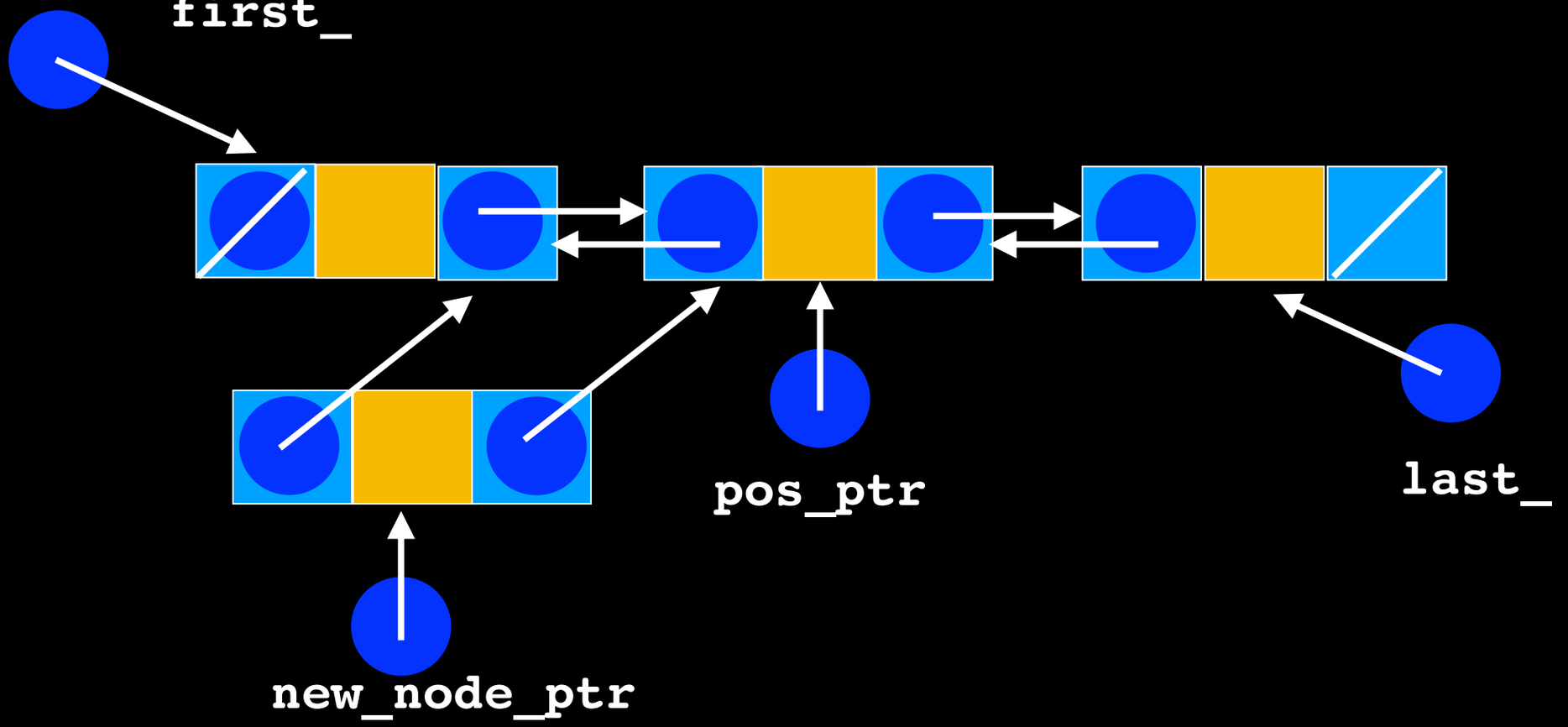
```
else if (pos_ptr == nullptr)
{
    //insert at end of list
    new_node_ptr->setNext(nullptr);
    new_node_ptr->setPrevious(last_);
    last_->setNext(new_node_ptr);
    last_ = new_node_ptr;
}
```



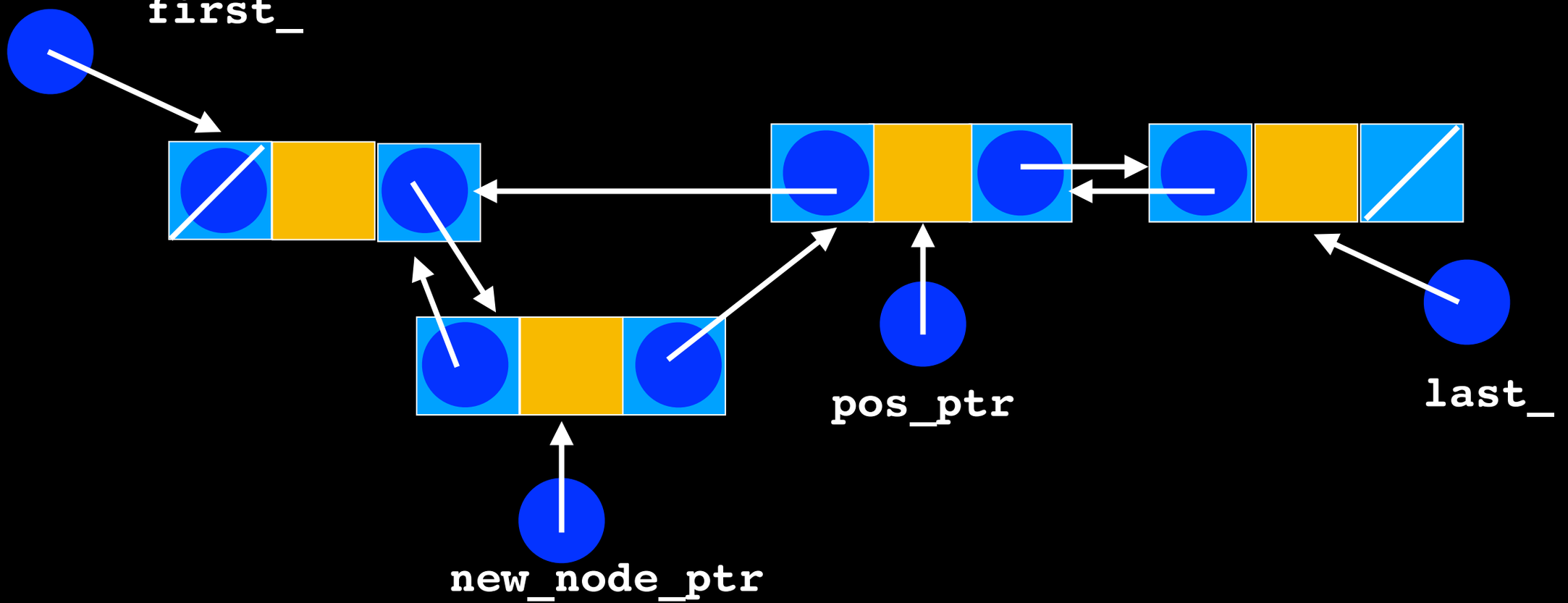
```
4 else
      {
         // Insert new node before node to which position points
         new_node_ptr->setNext(pos_ptr);
         new_node_ptr->setPrevious(pos_ptr->getPrevious());
         pos_ptr->getPrevious()->setNext(new_node_ptr);
         pos_ptr->setPrevious(new_node_ptr);
      }  // end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



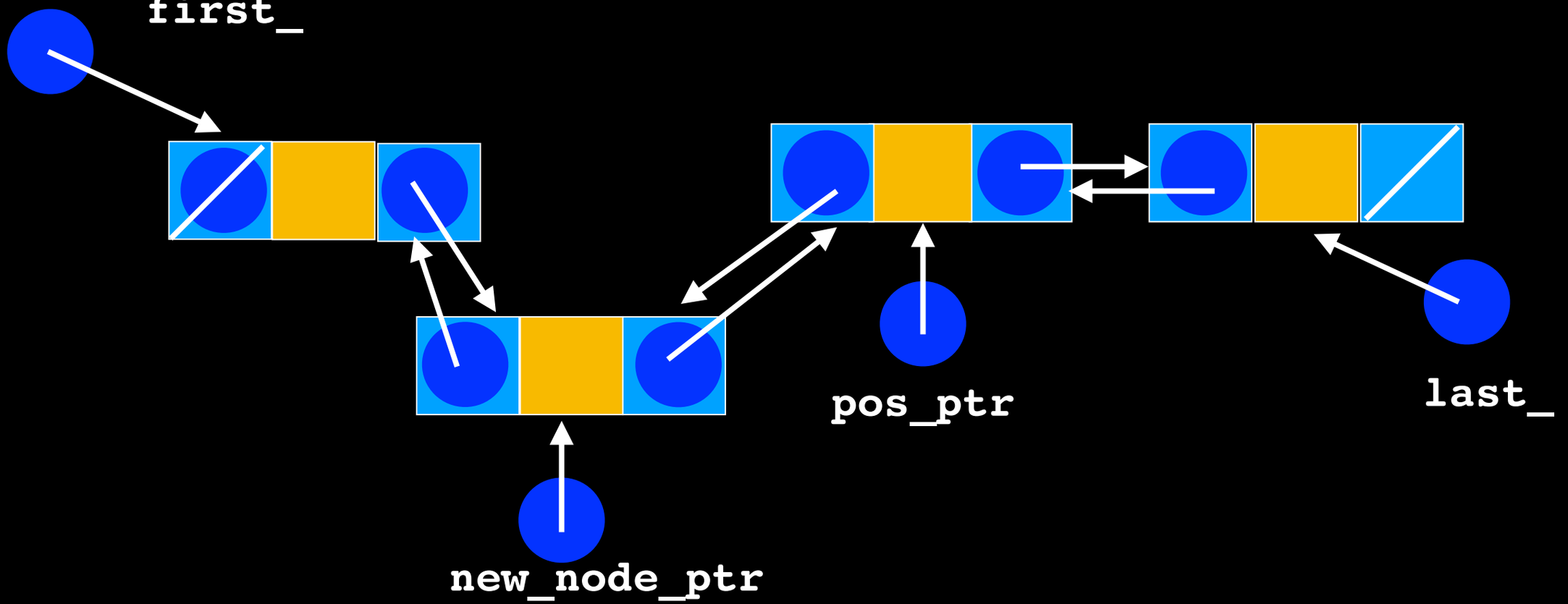
```
4 else
      {
         // Insert new node before node to which position points
         new_node_ptr->setNext(pos_ptr);
         new_node_ptr->setPrevious(pos_ptr->getPrevious());
         pos_ptr->getPrevious()->setNext(new_node_ptr);
         pos_ptr->setPrevious(new_node_ptr);
      }  // end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



```
4 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
```

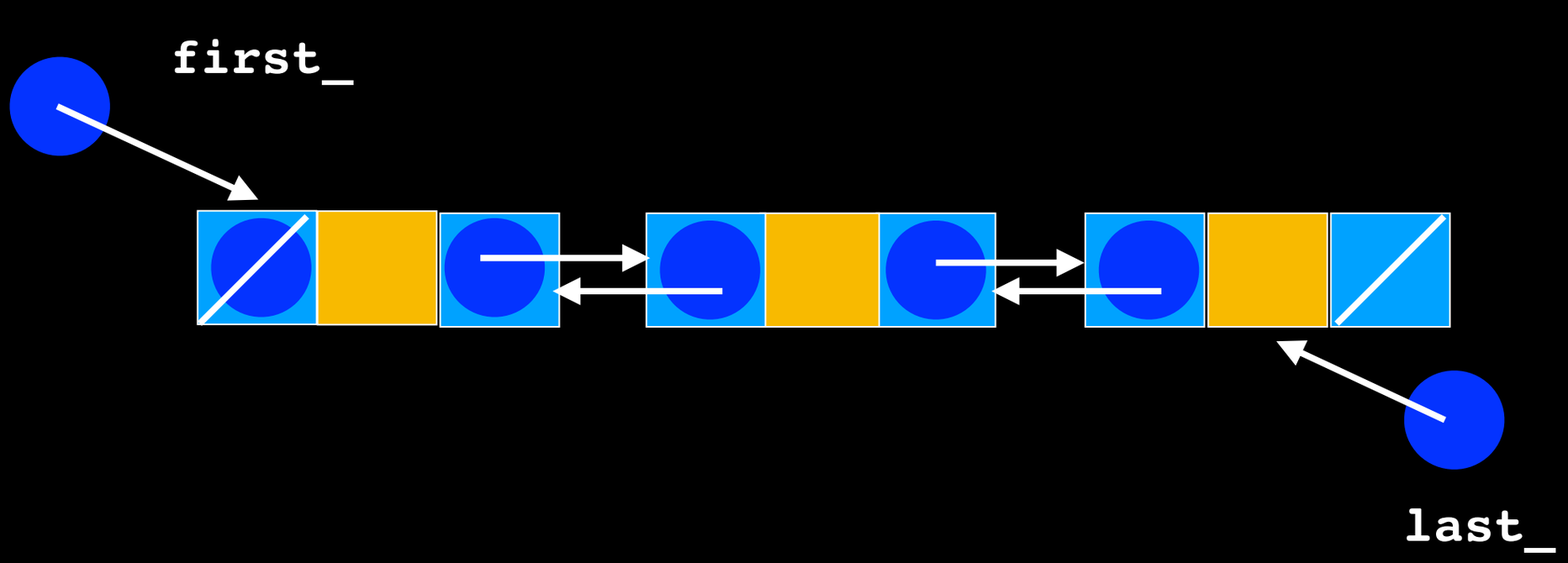


```
4 else
      {
         // Insert new node before node to which position points
         new_node_ptr->setNext(pos_ptr);
         new_node_ptr->setPrevious(pos_ptr->getPrevious());
         pos_ptr->getPrevious()->setNext(new_node_ptr);
         pos_ptr->setPrevious(new_node_ptr);
      }  // end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



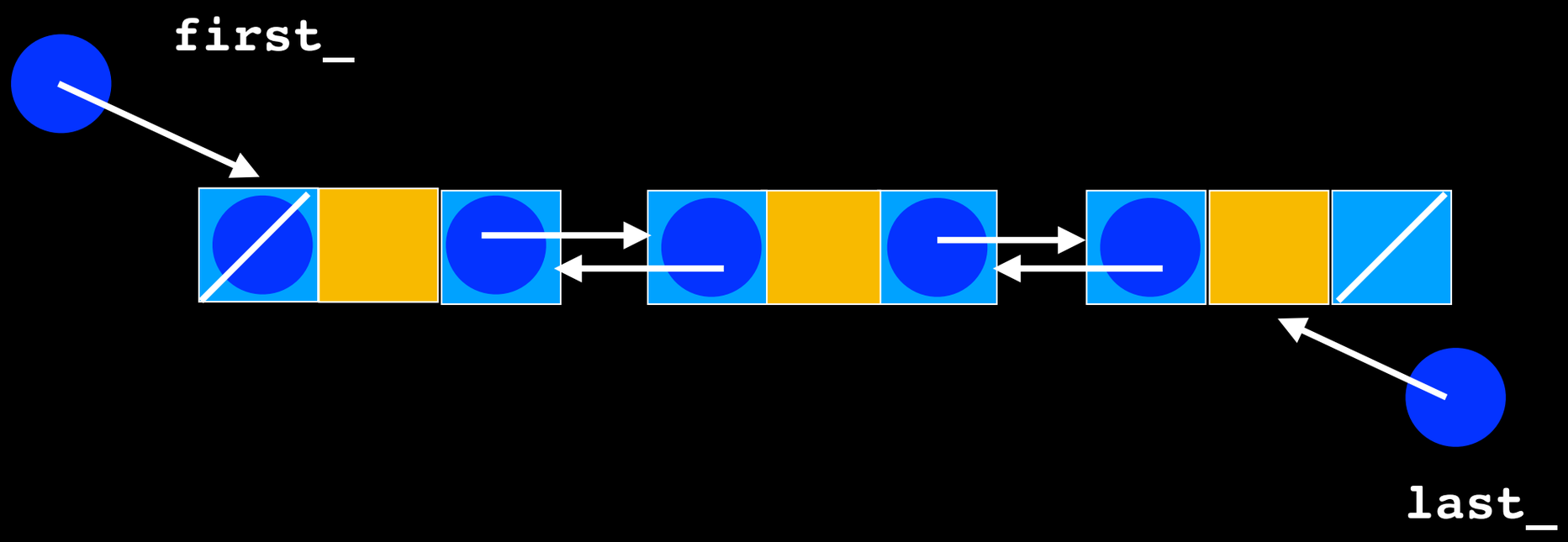
List::remove

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
```

0 O(n)

1

```
if ((pos_ptr == first_) && pos_ptr == last_))
{
     // Only one node
     first_ = nullptr;
     last_ = nullptr;

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

2

```
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;
}
```

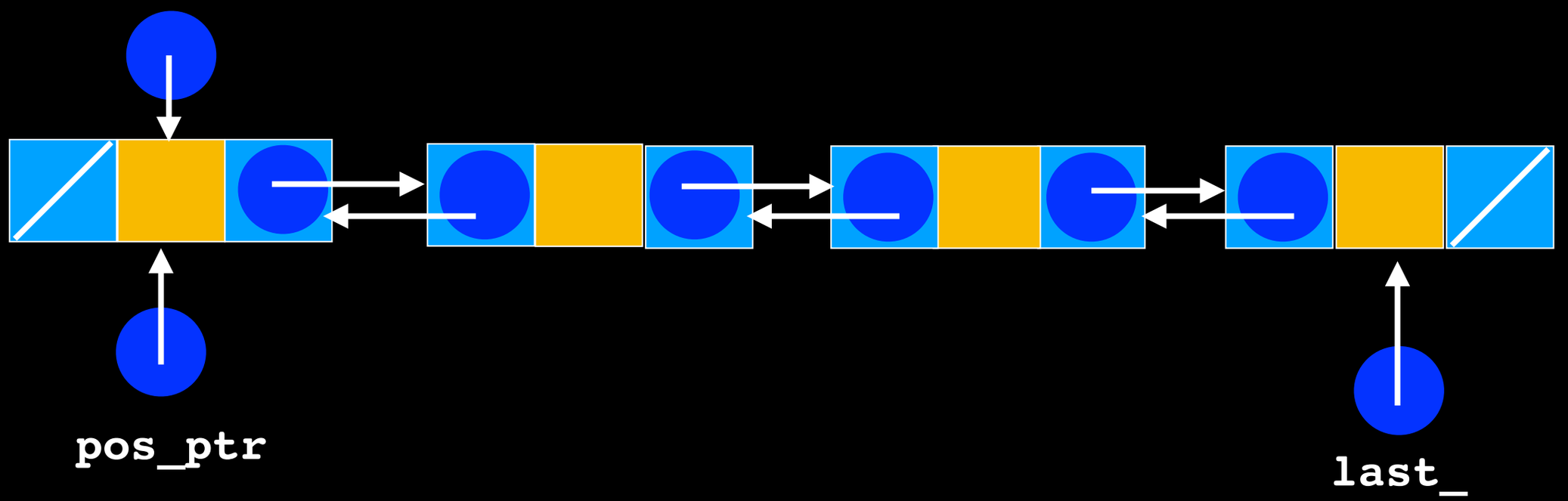
3

4

5 O(1) cases

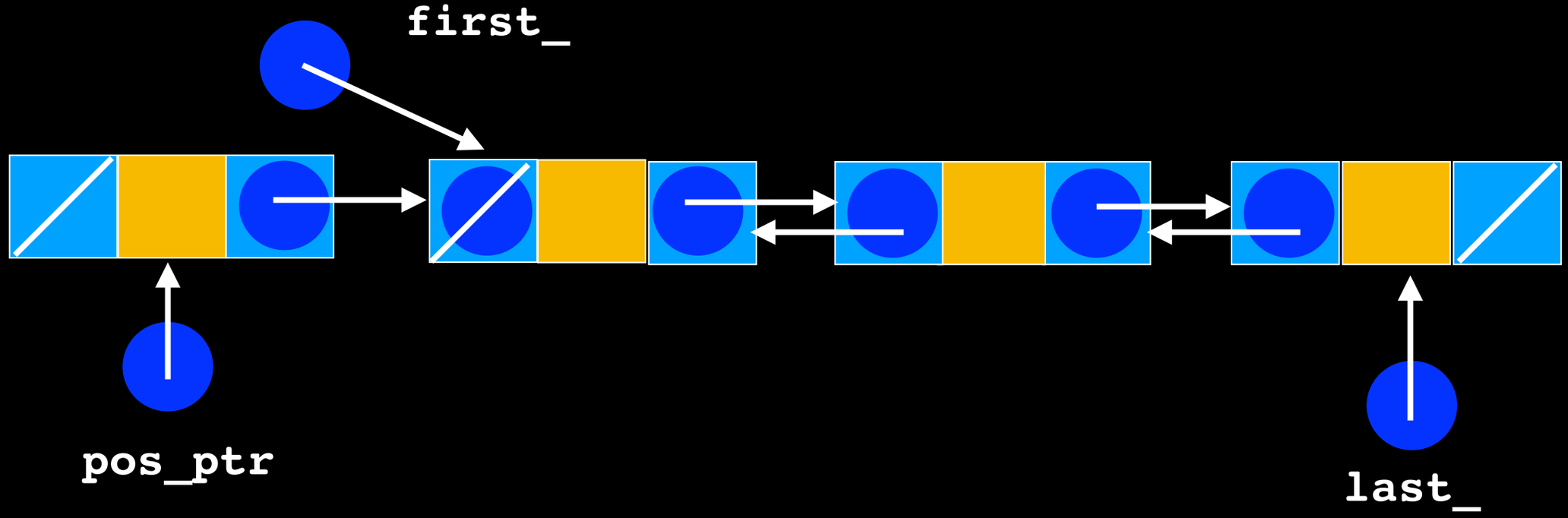
2

```
// 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;
}
```



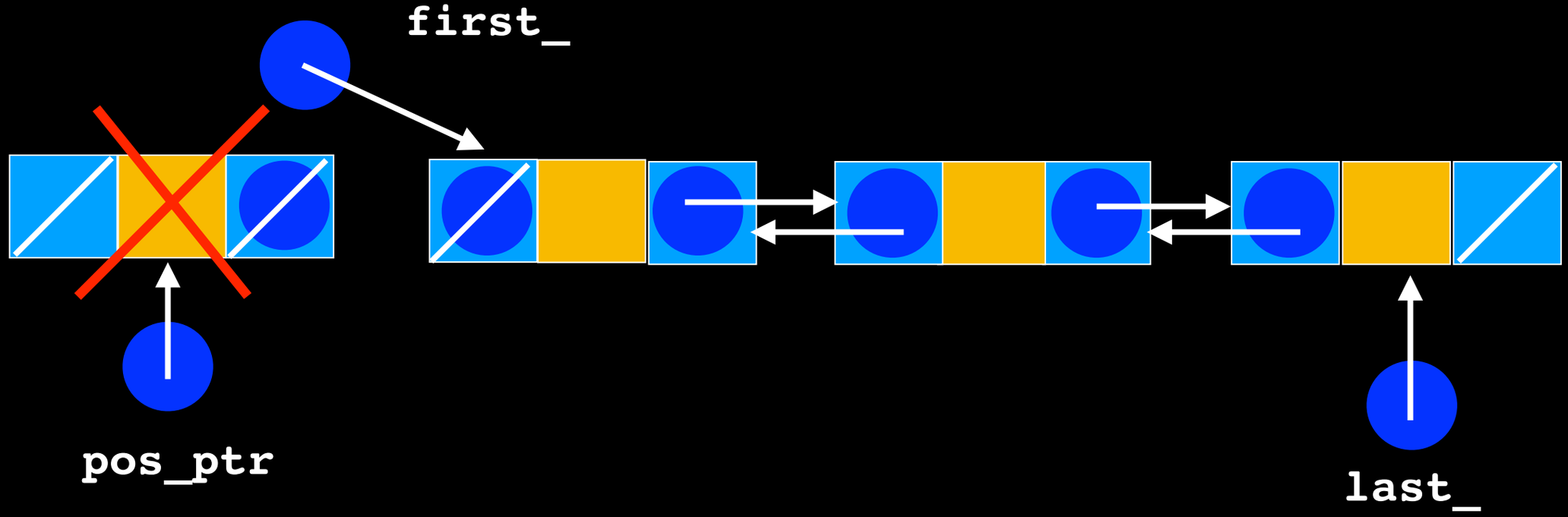
2

```
// 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;
}
```



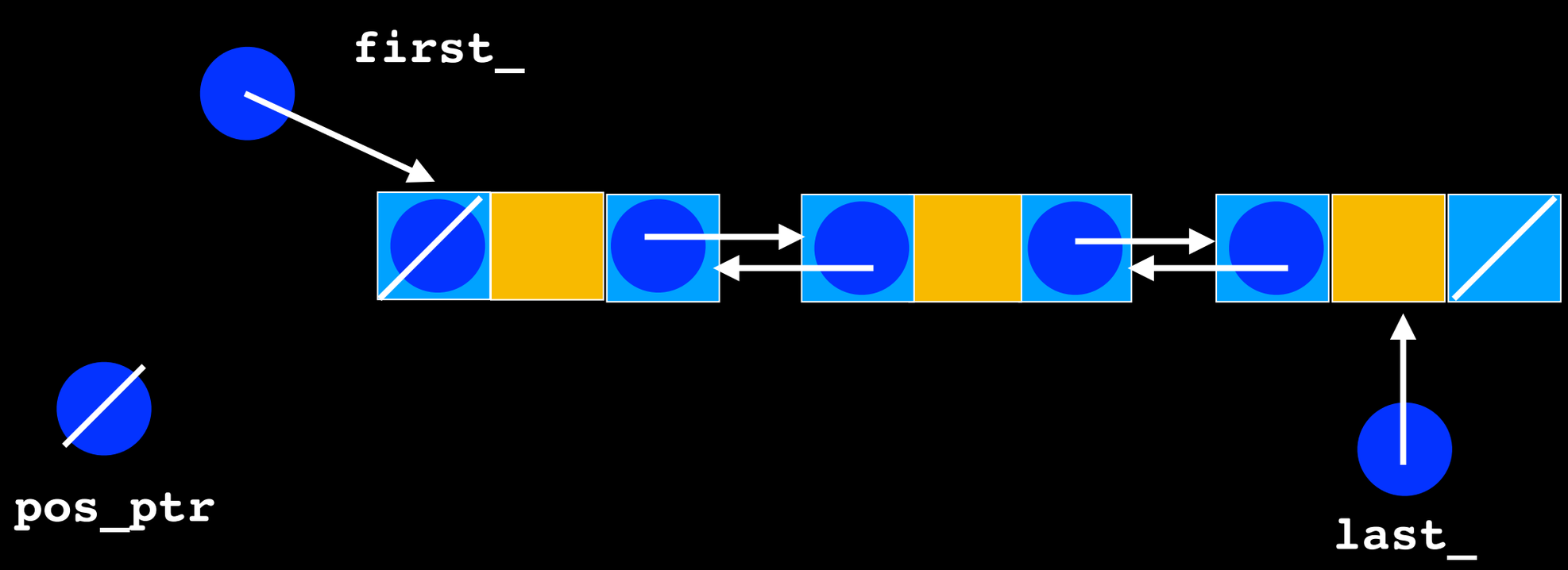
2

```
// 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;
}
```

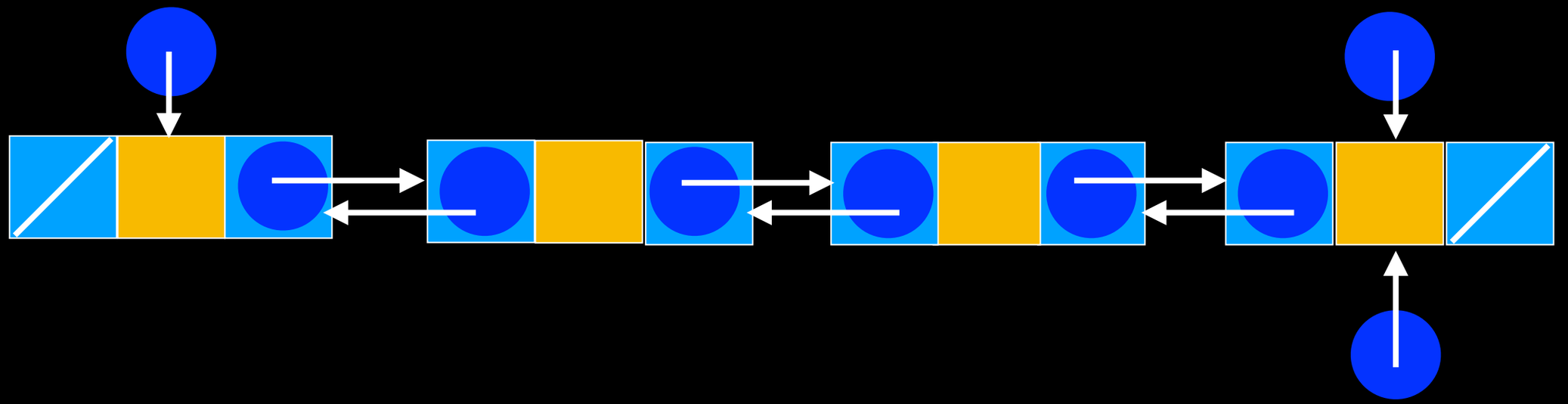


2

```
// 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;
}
```

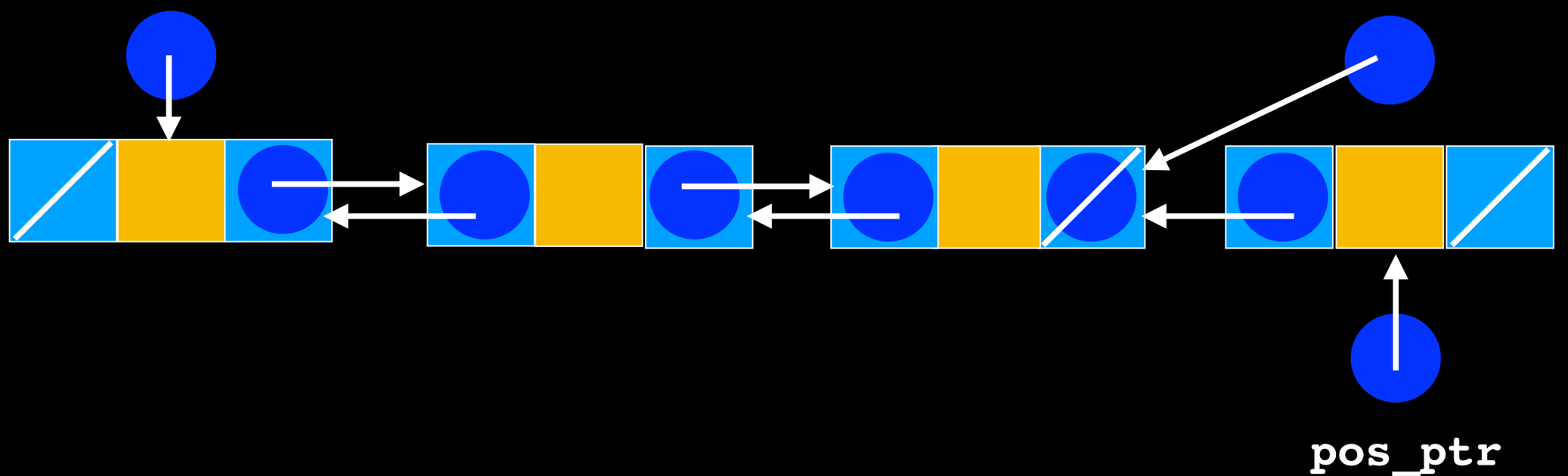


3
```
else if (pos_ptr == last_ )
{
    //remove last_  node
    last_  = pos_ptr->getPrevious();
    last_ ->setNext(nullptr);

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



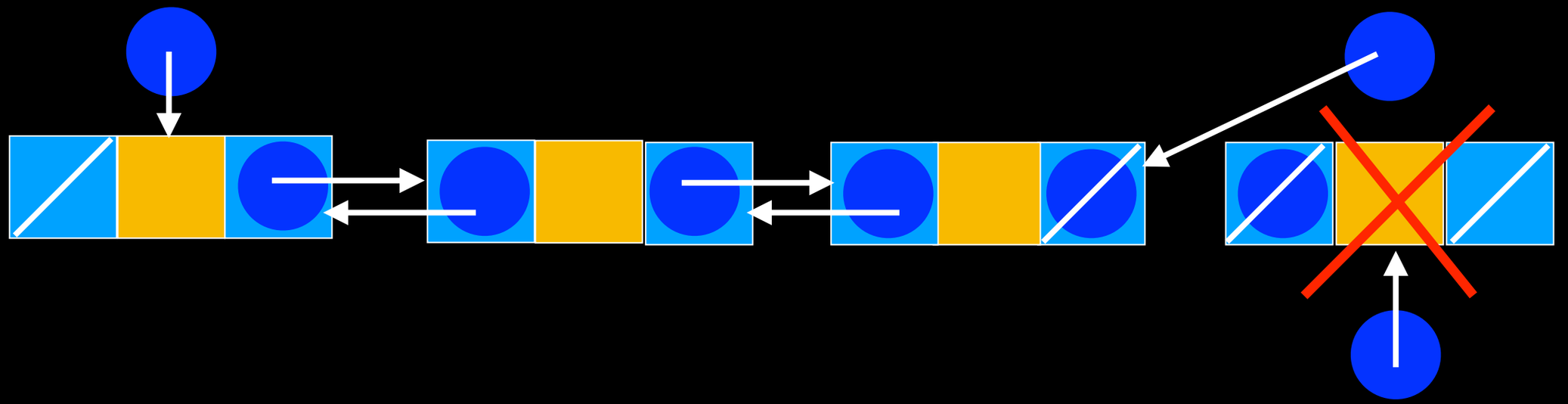
3

```
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;
}
```



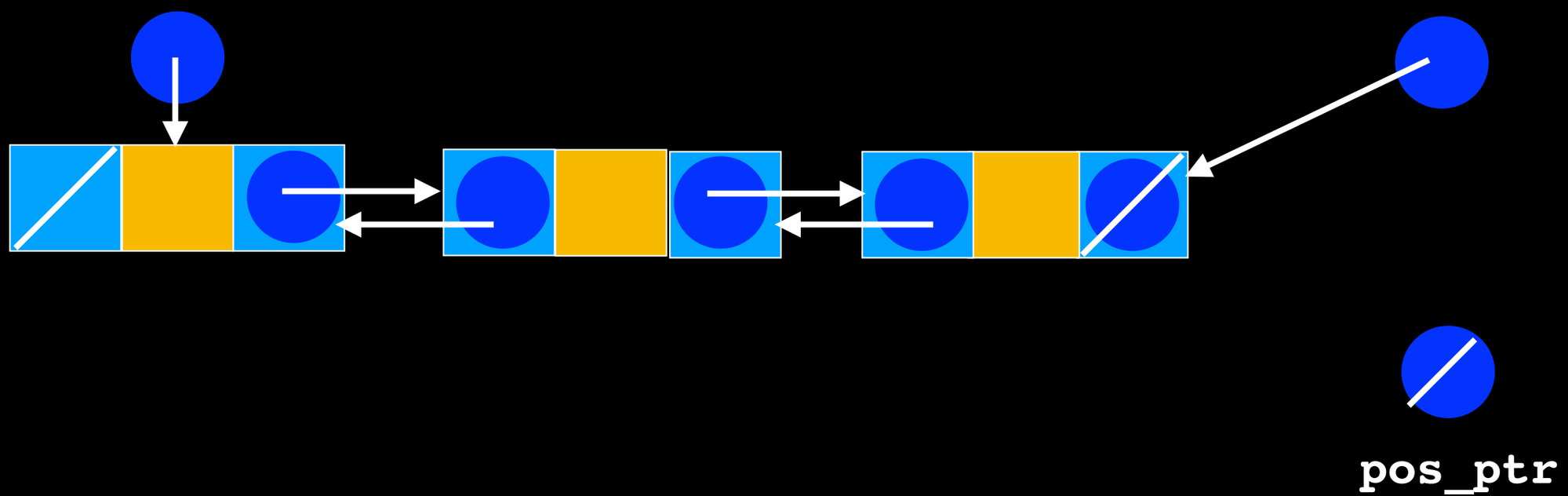
3

```
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;
}
```

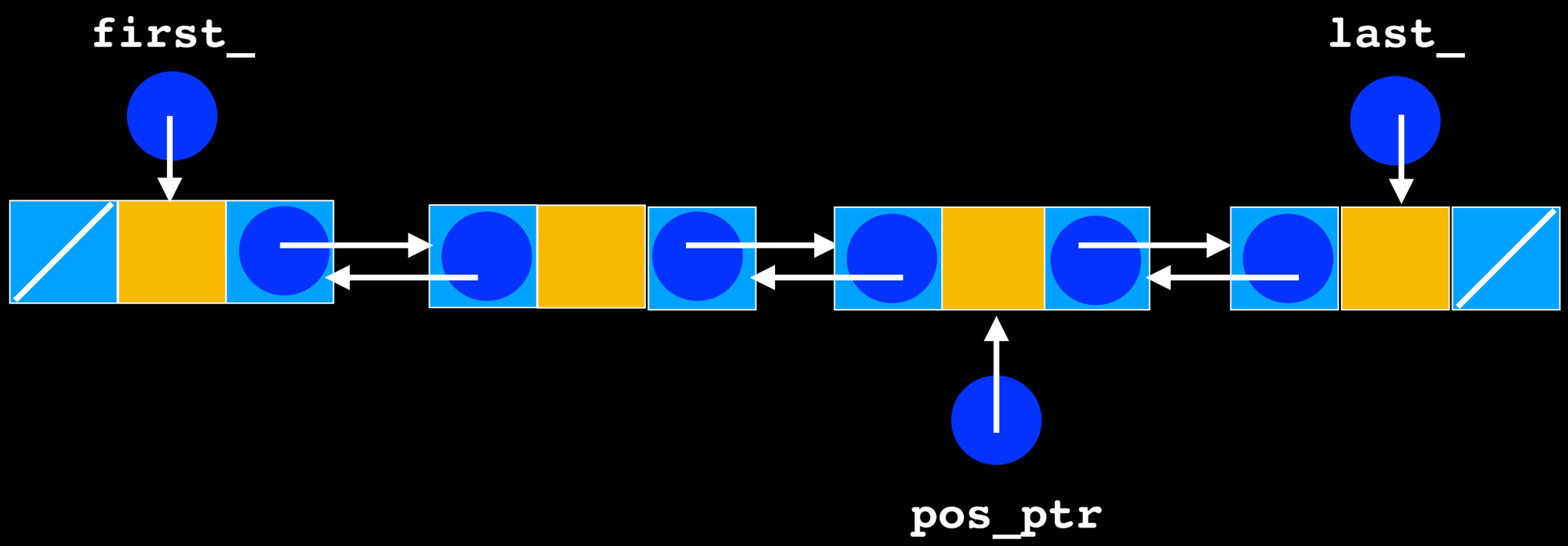


3

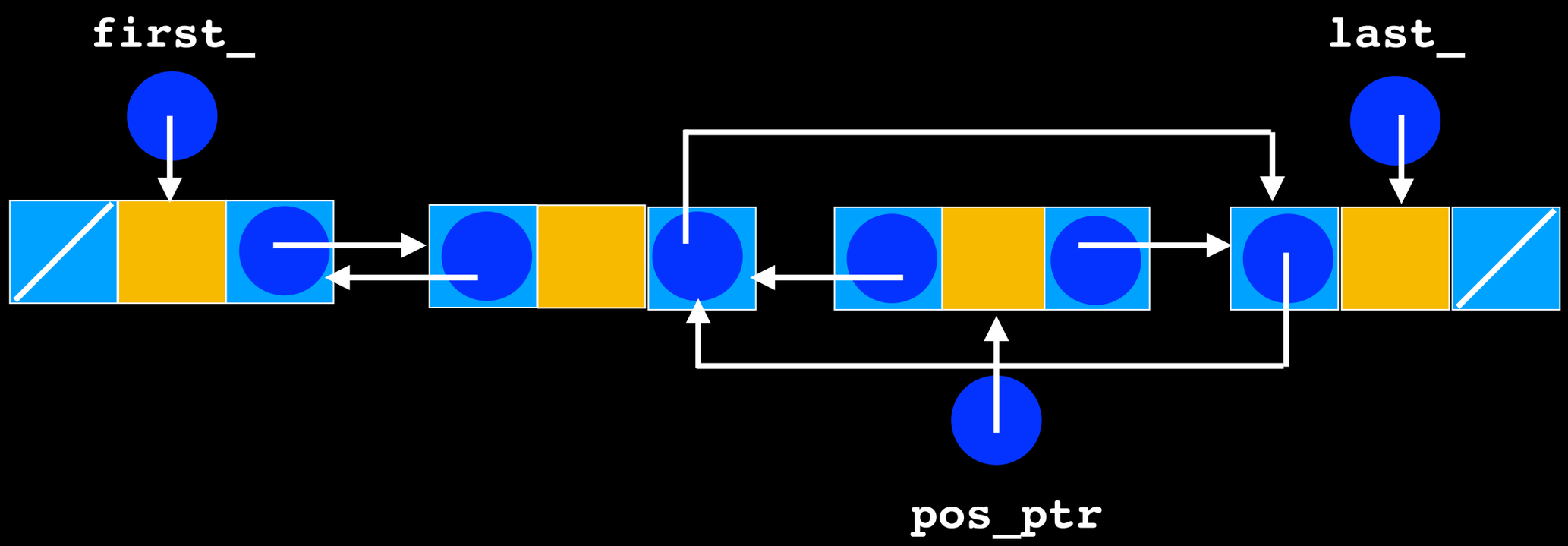
```
else if (pos_ptr == last_ )
{
    //remove last_  node
    last_  = pos_ptr->getPrevious();
    last_ ->setNext(nullptr);

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



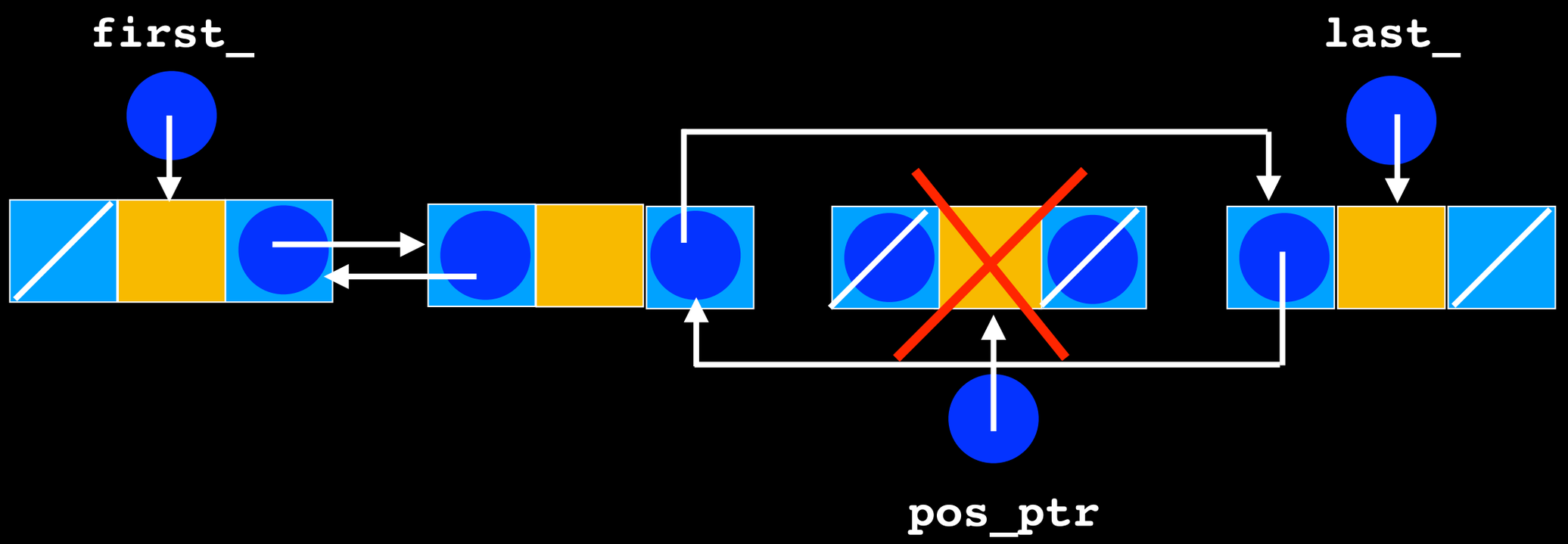
4
```
else if (pos_ptr != nullptr)
{
    //Remove from the middle
    pos_ptr->getPrevious()->setNext(pos_ptr->getNext());
    pos_ptr->getNext()->setPrevious(pos_ptr->getPrevious());

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



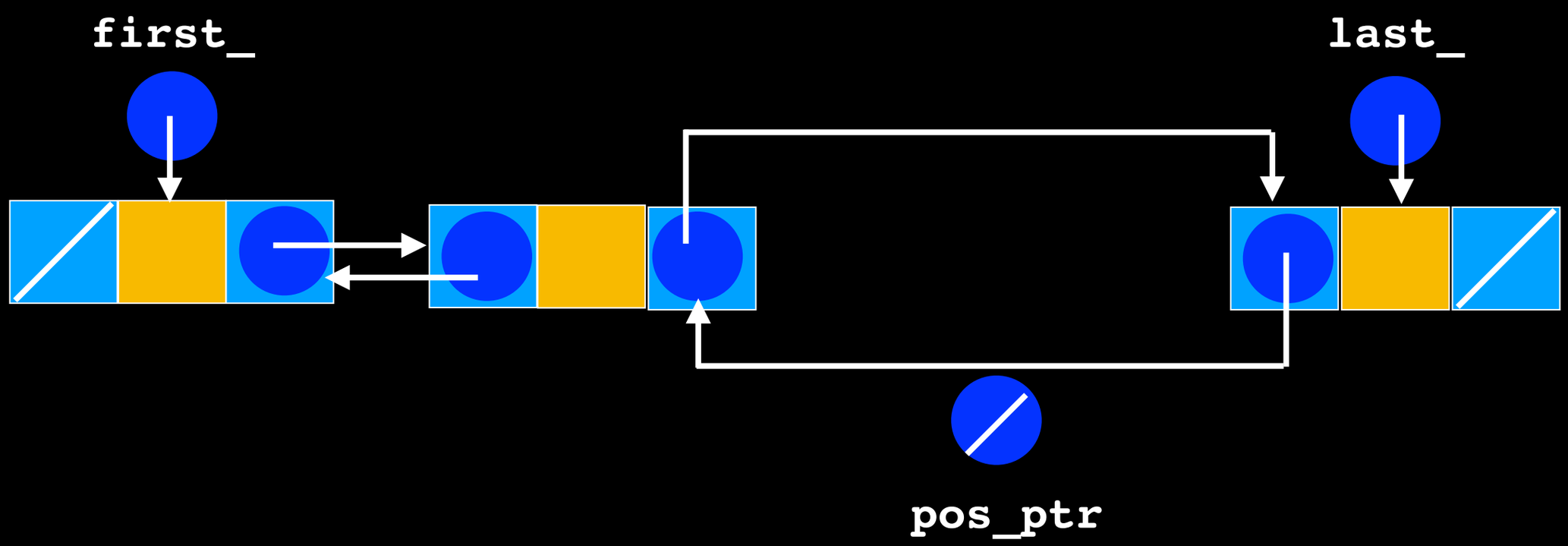
4
```
else if (pos_ptr != nullptr)
{
    //Remove from the middle
    pos_ptr->getPrevious()->setNext(pos_ptr->getNext());
    pos_ptr->getNext()->setPrevious(pos_ptr->getPrevious());

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



4
```
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
```



4
```
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
```

How could you optimize traversal TO POSITION given this implementation?

```
if(position < item_count/2)
    {
        find_ptr = first_;
         . . .
else
{
        find_ptr = last_;
         . . .
```

Still O(n)!!!

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 value to indicate uninitialized object

Next time

Exception Handling