Lists

Tiziana Ligorio

tligorio@hunter.cuny.edu

Today’s Plan

“Get your hands dirty”
Demo

Lists

Announcements and Syllabus Check

Next Tuesday:

- Discuss Project 4
- Midterm Review

Come ready to ask questions!!!

Follow the link for Tentative Schedule from course webpage

Demo

You should do this home!

Lists

What makes a list?

Lists ADT

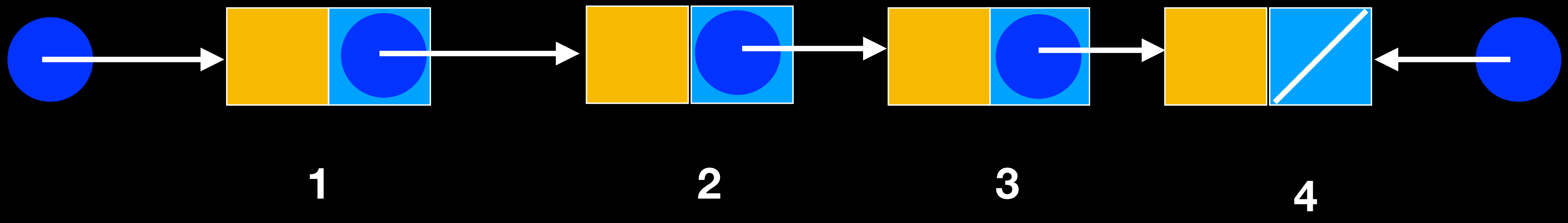
What makes a list?

Play**List**?

Duplicates allowed or not is not a defining factor

What makes a list?

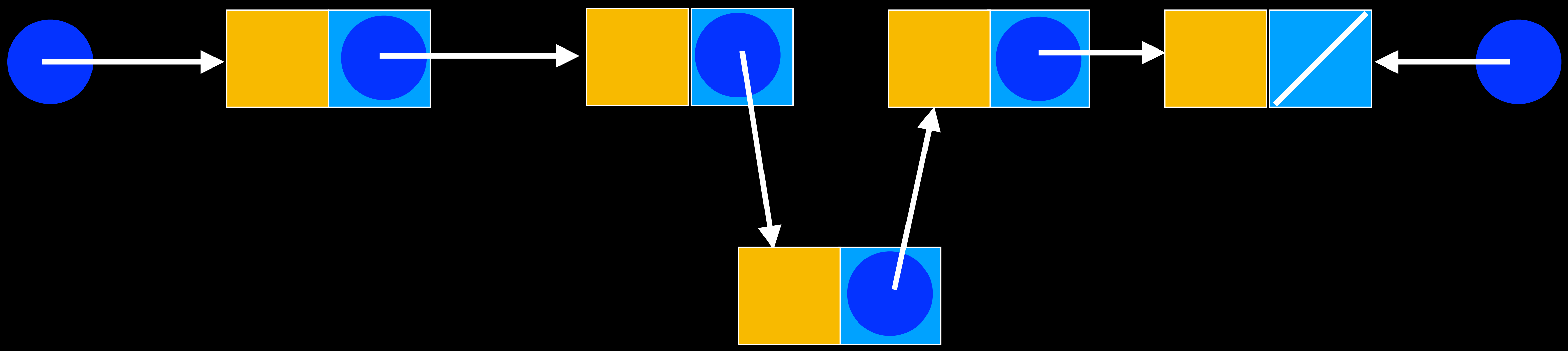
Order is implied



What makes a list?

Order is implied

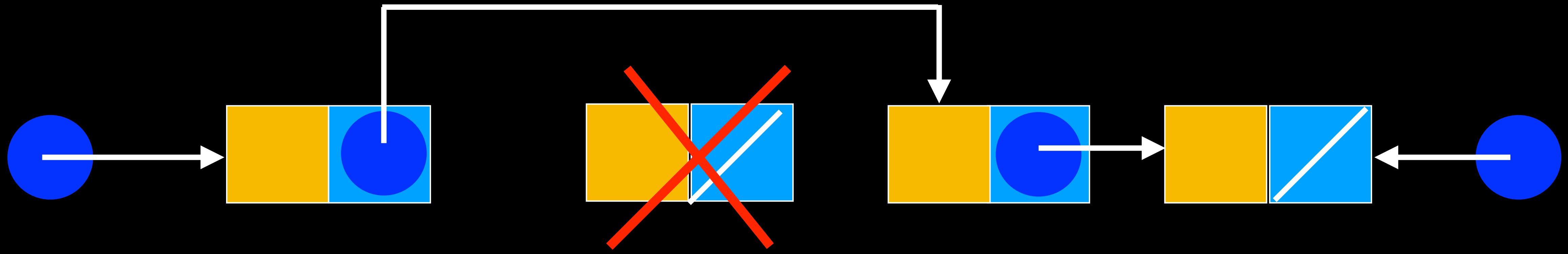
Insertion and removal from middle retains order



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

✓ Low cost of operation, does not depend on # of items

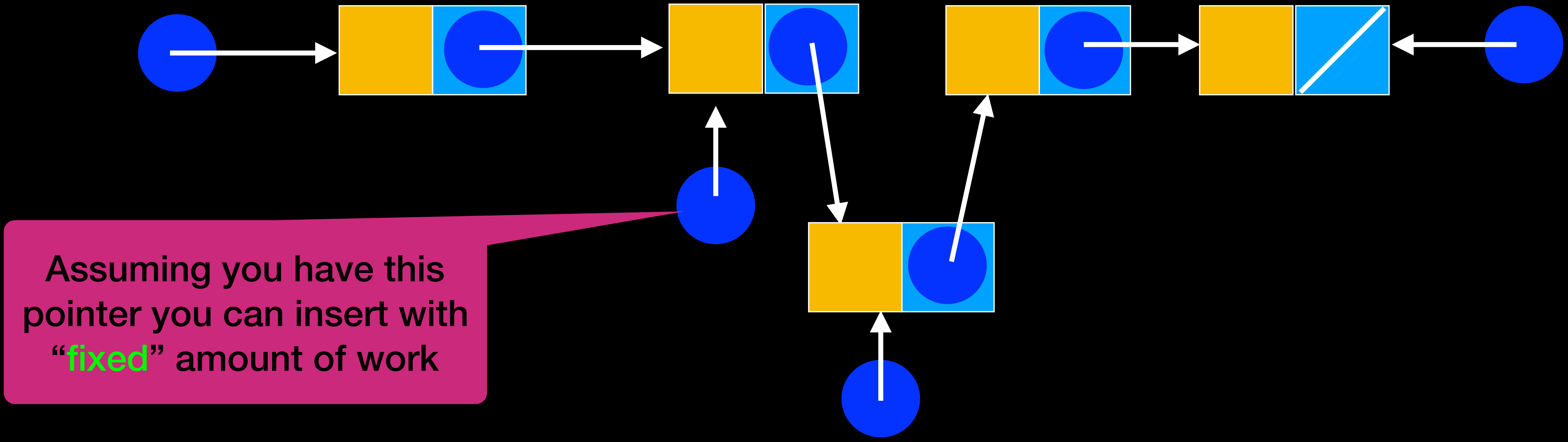
✗ High cost of operation, depends on # of items

	Arrays/Vectors	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	✗	✓ ?

What about the cost of finding the node to remove?

INSERT

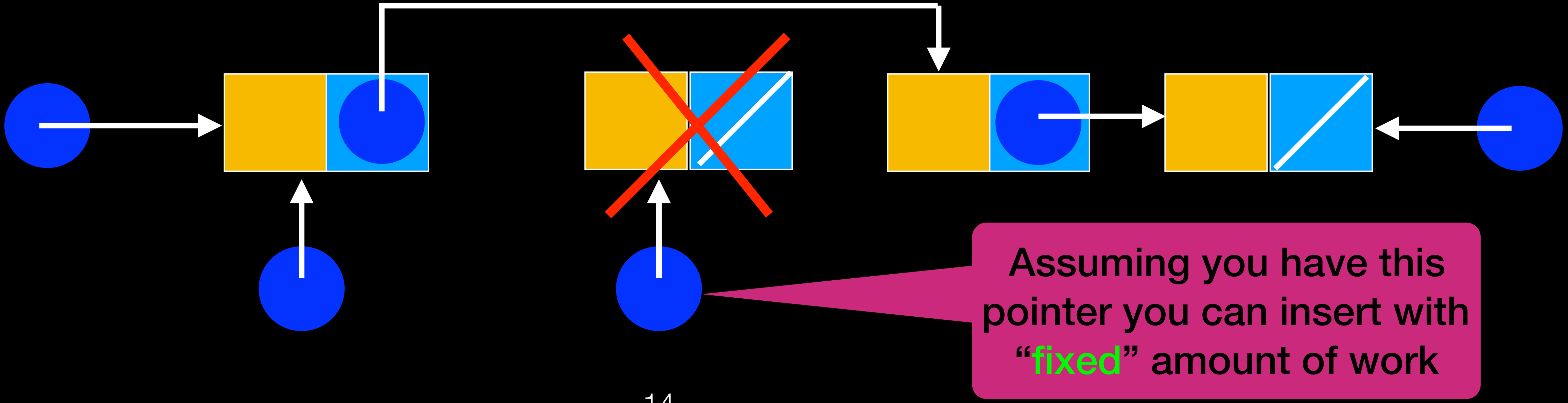
```
void insert(Node<ItemType>* position, ItemType new_element);
```



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

REMOVE

```
void remove(Node<ItemType>* position);
```



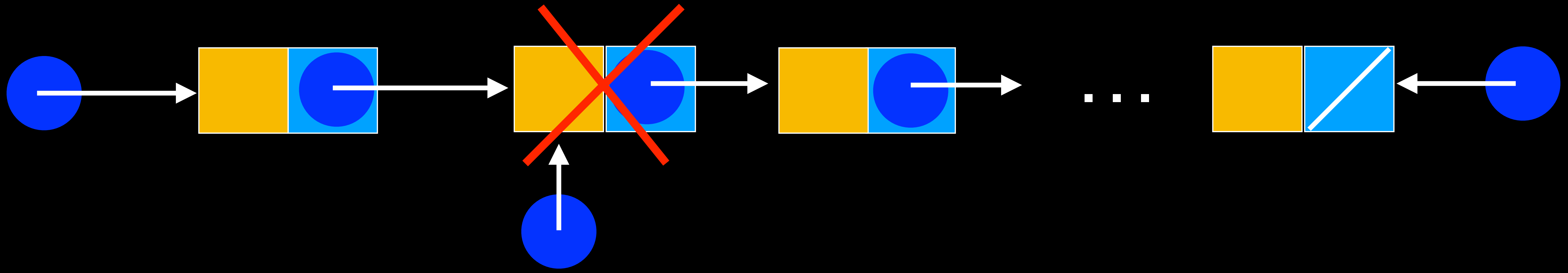
Assuming you have this pointer you can insert with “fixed” amount of work

Caveat

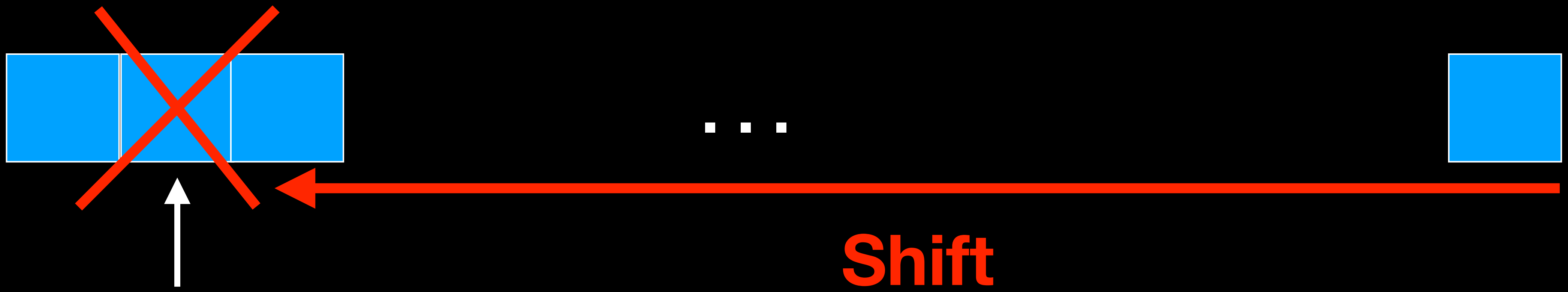
By passing a pointer to insert and remove keep cost of operation “fixed"

Consider that we may need to find the pointer to the node before inserting/removing —> traversal: high cost, *depends on number of elements in list*

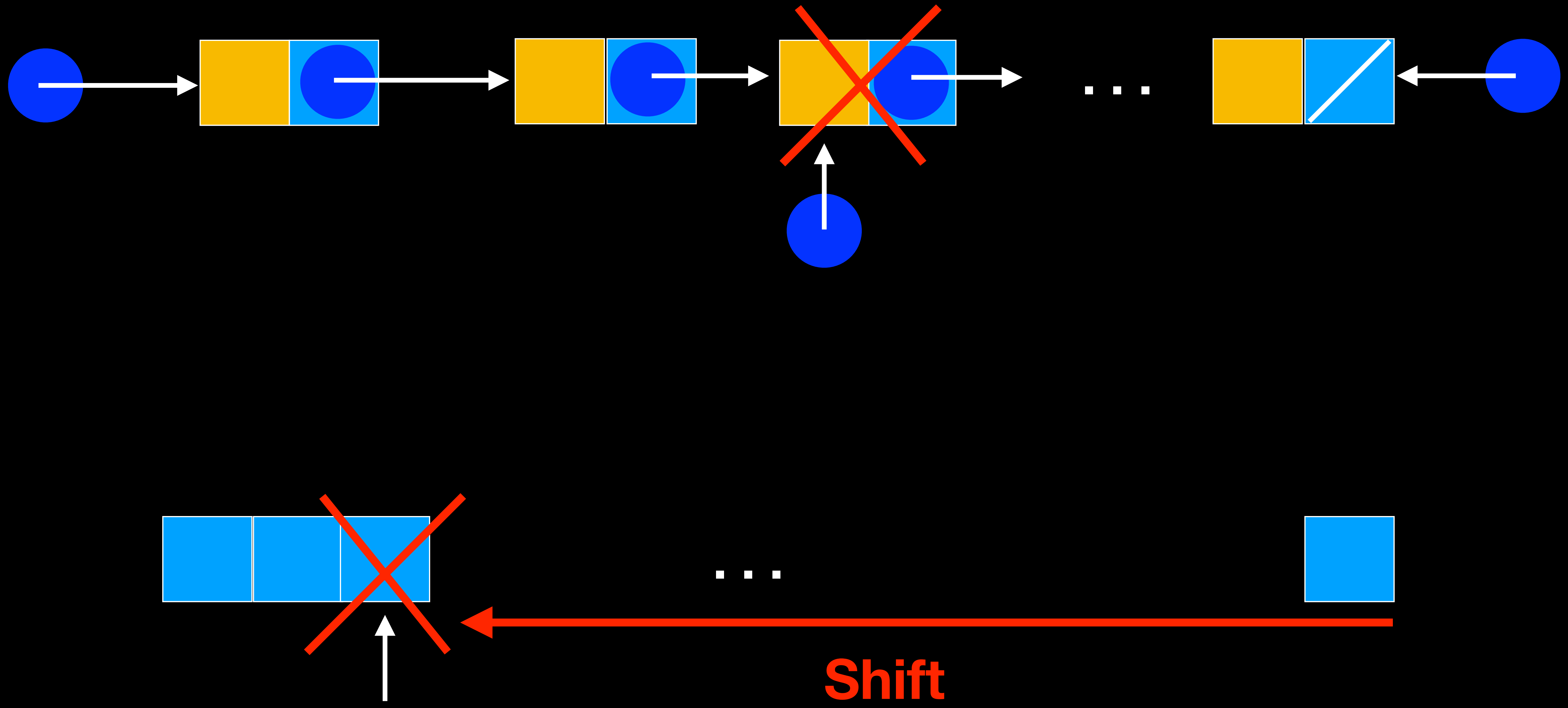
If operations (insertion/deletions) occur on nodes that are close to each other operation cost can stay low



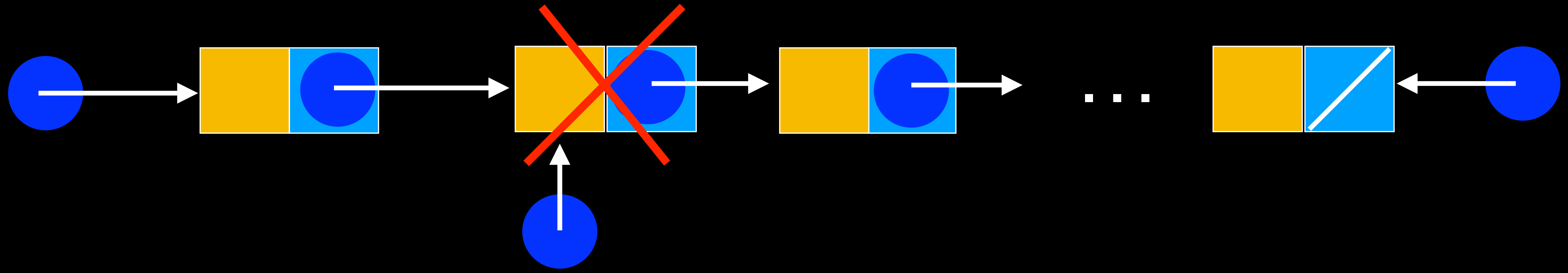
Find

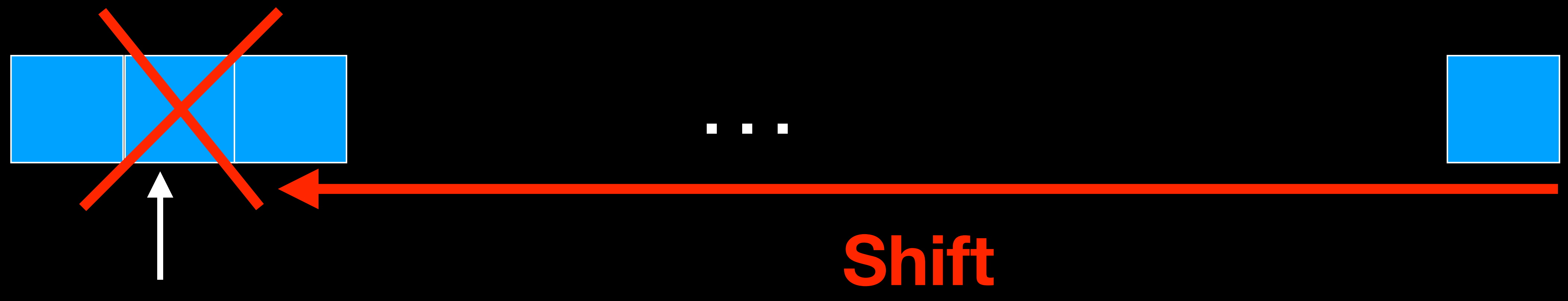


Shift

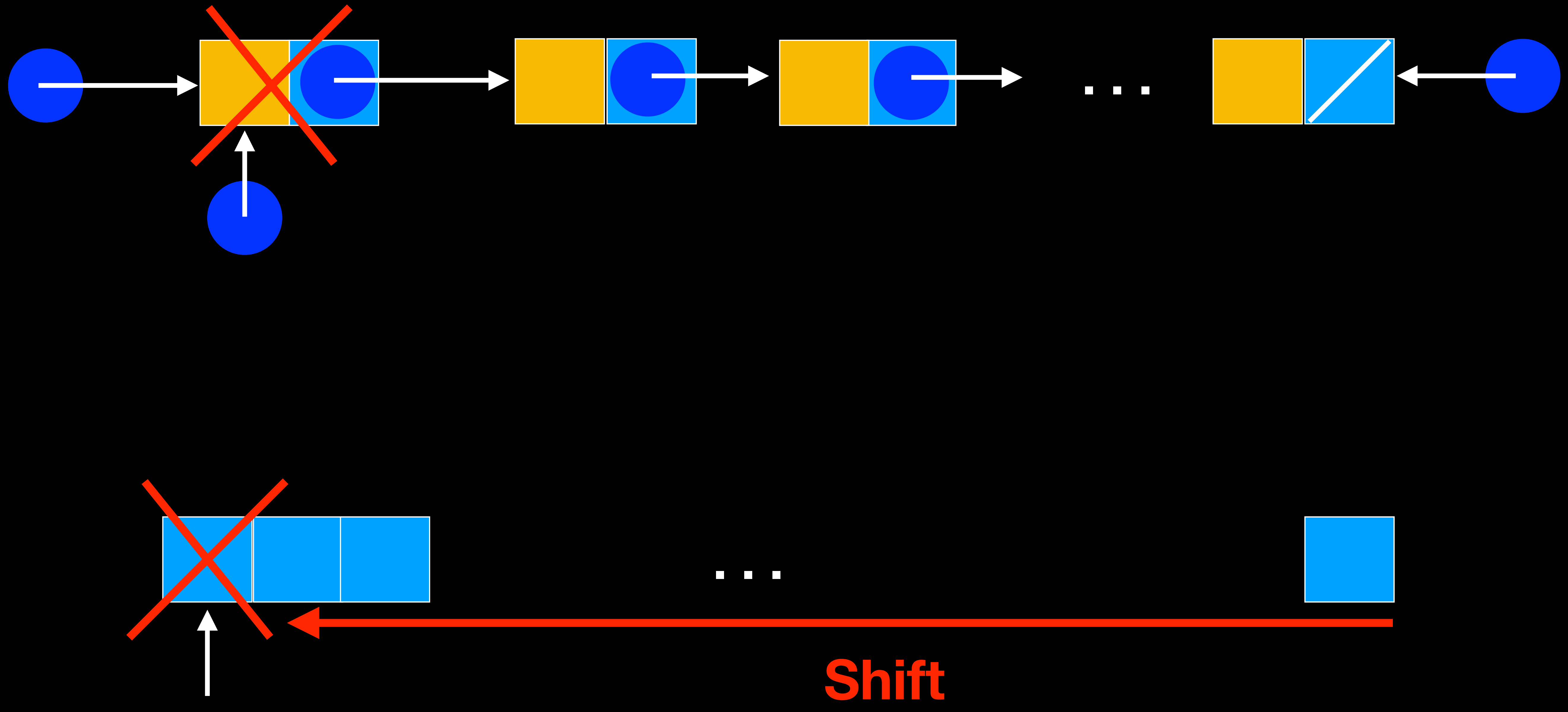


Shift



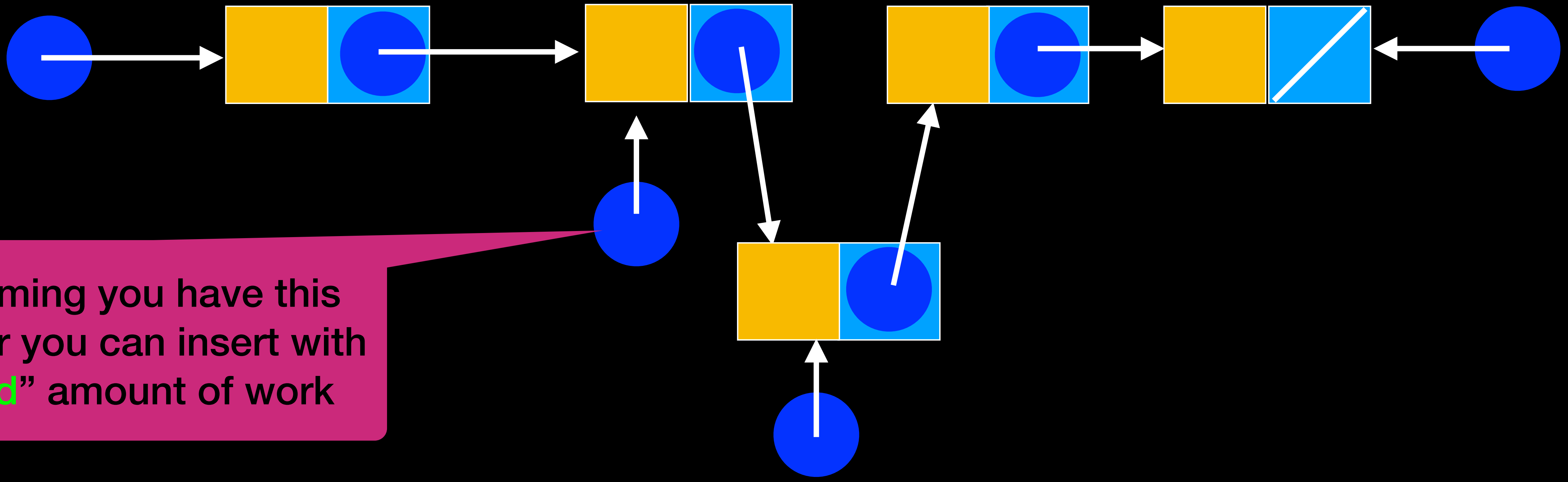


Shift


. . .
. . .
Shift

INSERT

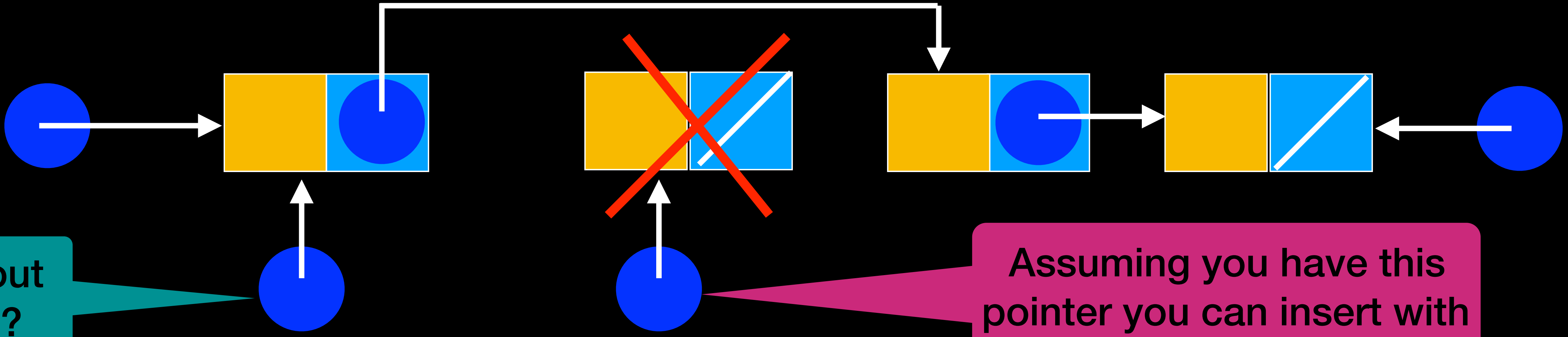
```
void insert(Node<ItemType>* position, ItemType new_element);
```



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

REMOVE

```
void remove(Node<ItemType>* position);
```

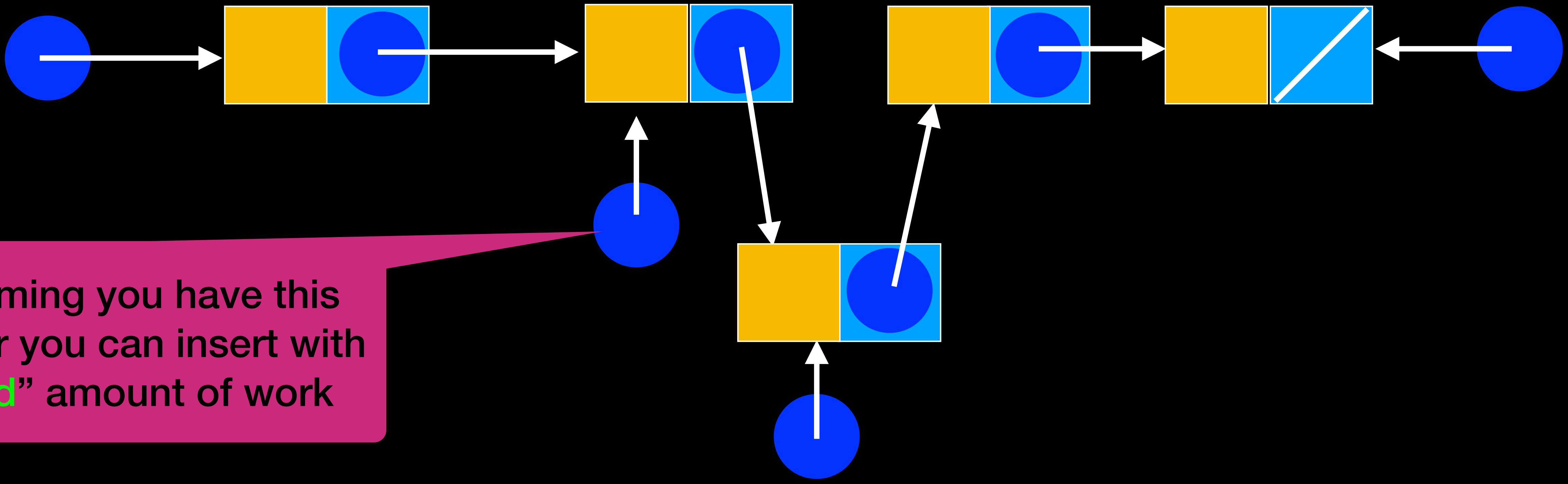


What about this one?

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

INSERT

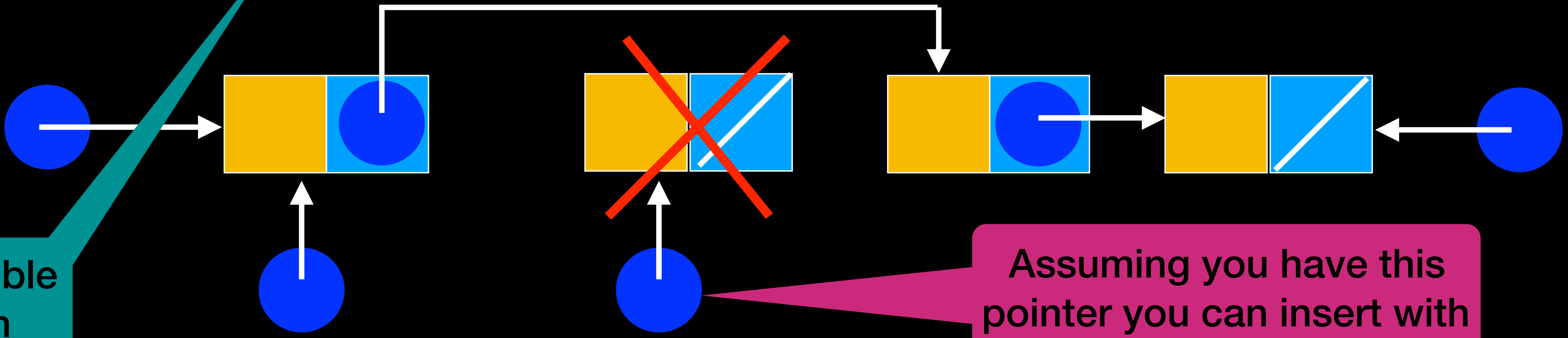
```
void insert(Node<ItemType>* position, ItemType new_element);
```



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

REMOVE

```
void remove(Node<ItemType>* position, Node<ItemType>* previous);
```

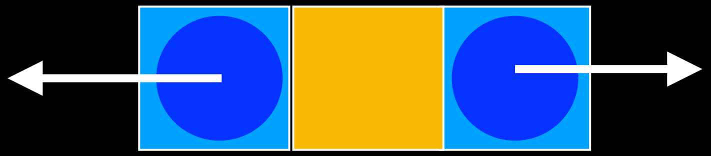


One possible solution

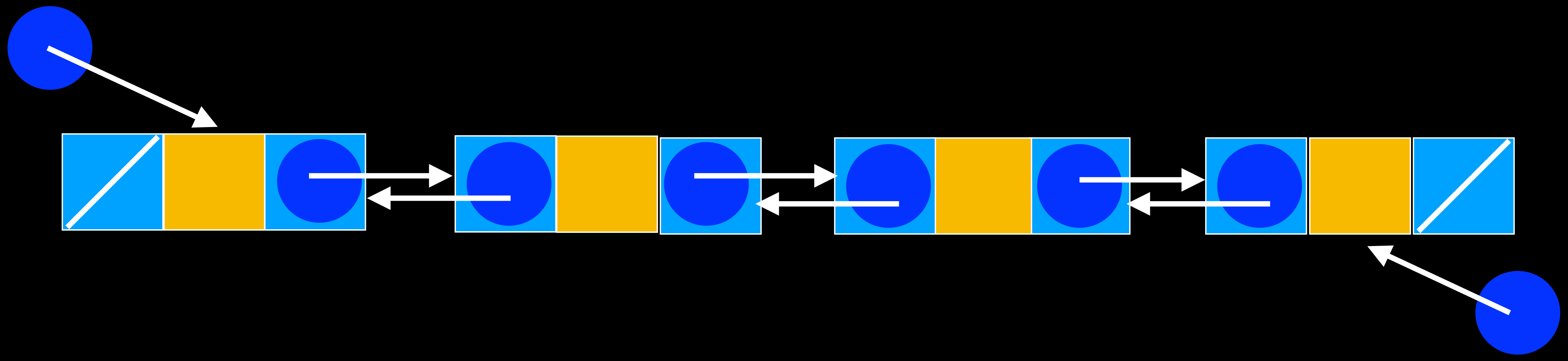
Assuming you have this pointer you can insert with “fixed” amount of work

Another Solution?

```
#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



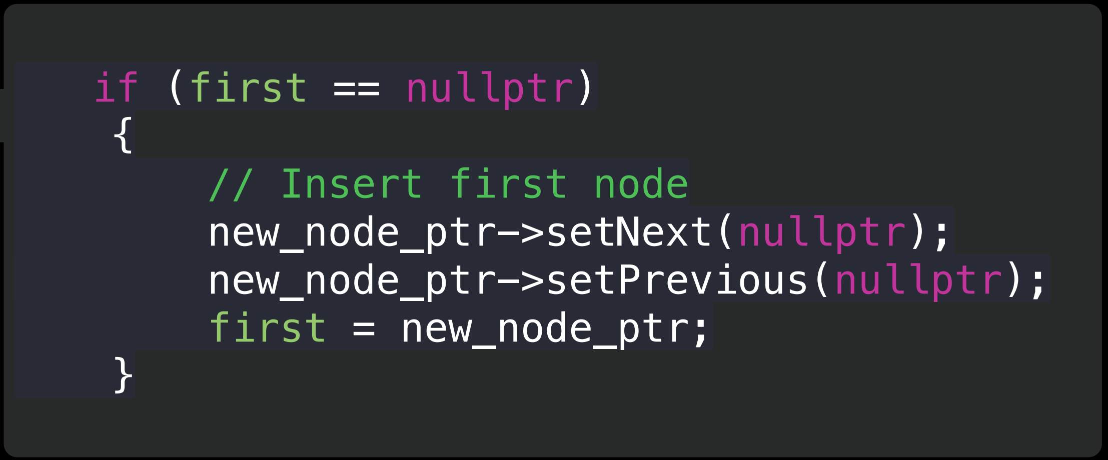
```cpp
#ifndef LIST_H_
#define LIST_H_

template<class ItemType>
class List
{

public:
    List(); // constructor
    List(const List<ItemType>& a_list); // copy constructor
    ~List(); // destructor
    bool isEmpty() const;
    size_t getLength() const;
    void insert(Node<ItemType>* position, const ItemType& new_element);
    void remove(Node<ItemType>* position);
    void clear();

    Node<ItemType>* getPointerTo(size_t position) const throw(std::out_of_range);
    Node<ItemType>* getFirst() const;
    Node<ItemType>* getLast() const;

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
}; // end Node

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

Specify in interface it might throw exception.
Compiler will complain if it tries to throw another type.
Can specify more than one separated by comma

```
#ifndef LIST_H_
#define LIST_H_

template<class ItemType>
class List
{

public:
    List(); // constructor
 ✗  List(const List<ItemType>& a_list); // copy constructor
 ✗  ~List(); // destructor
 ✓  bool isEmpty() const;
 ✓  size_t getLength() const;
 ✓  void insert(Node<ItemType>* position, const ItemType& new_element);
 ✓  void remove(Node<ItemType>* position);
 ✗  void clear();

 ✗  Node<ItemType>* getPointerTo(size_t position) const throw(std::out_of_range);
 ✓  Node<ItemType>* getFirst() const;
 ✓  Node<ItemType>* getLast() const;

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
}; // end Node

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

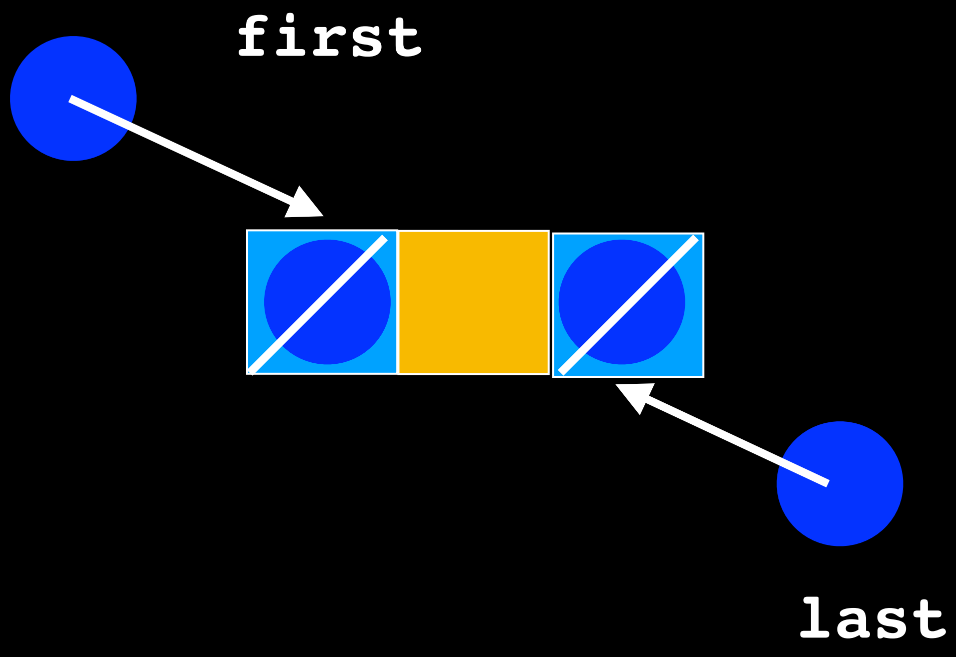
List::insert

```
template<class ItemType>
void List<ItemType>::insert(Node<ItemType>* position, const ItemType& new_element)
{
      // Create a new node containing the new entry
      Node<ItemType>* new_node_ptr = new Node<ItemType>(new_element);

      // Attach new node to chain

      else if (position == 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 (position == 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(position);
         new_node_ptr->setPrevious(position->getPrevious());
         position->getPrevious()->setNext(new_node_ptr);
         position->setPrevious(new_node_ptr);
      }  // end if

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

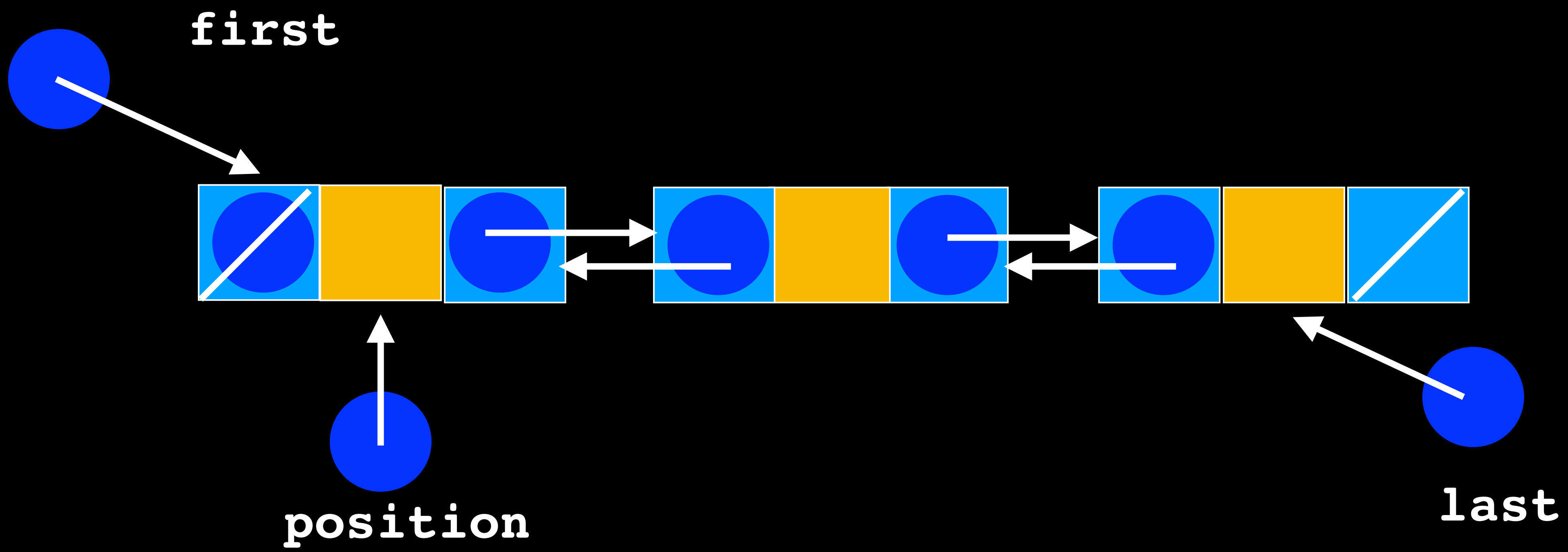


```
if (first == nullptr)
 {
     // Insert first node
     new_node_ptr->setNext(nullptr);
     new_node_ptr->setPrevious(nullptr);
     first = new_node_ptr;
 }
```

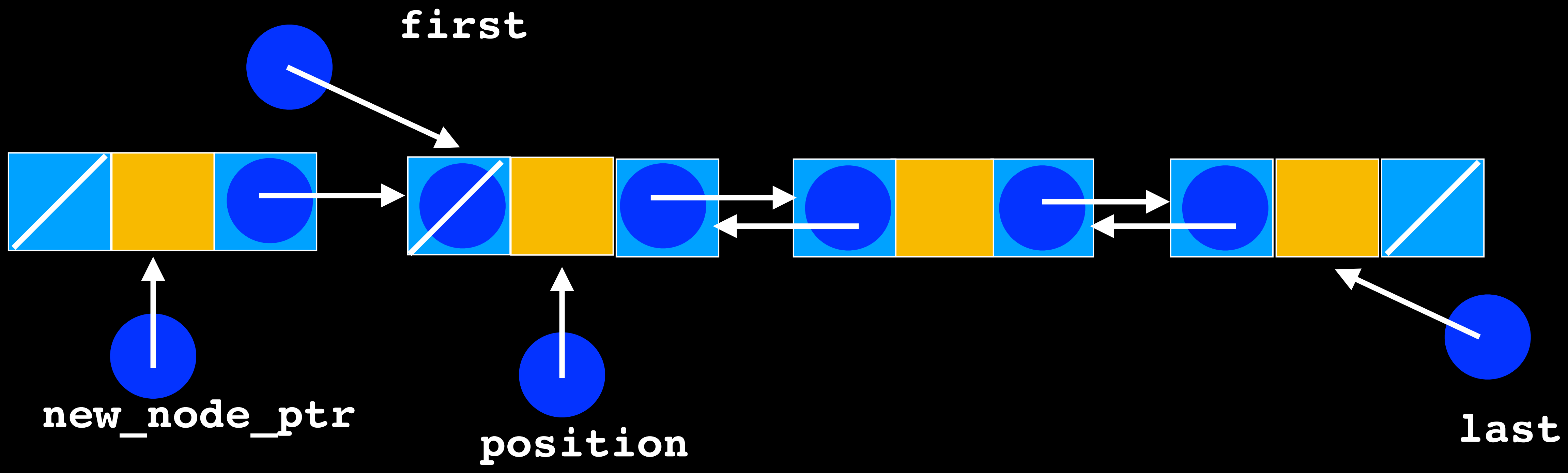
```
if (first == nullptr)
{
    // Insert first node
    new_node_ptr->setNext(nullptr);
    new_node_ptr->setPrevious(nullptr);
    first = new_node_ptr;
    last = new_node_ptr;
}
```



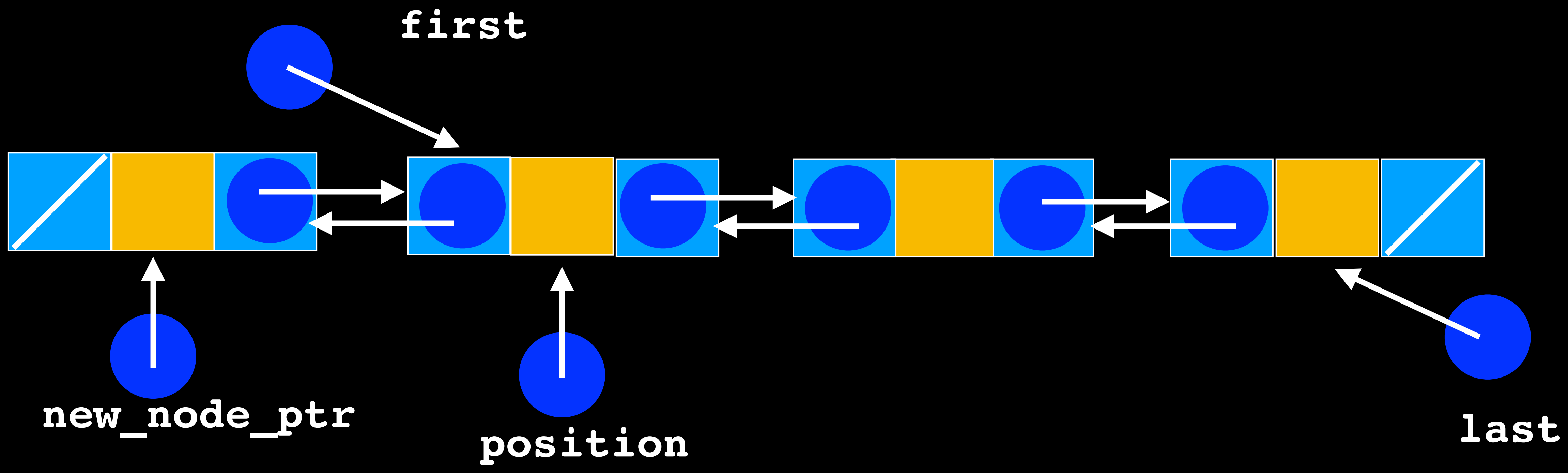
```
else if (position == first)
      {
         // Insert new node at beginning of chain
         new_node_ptr->setNext(first);
         new_node_ptr->setPrevious(nullptr);
         first->setPrevious(new_node_ptr);
         first = new_node_ptr;
      }
```



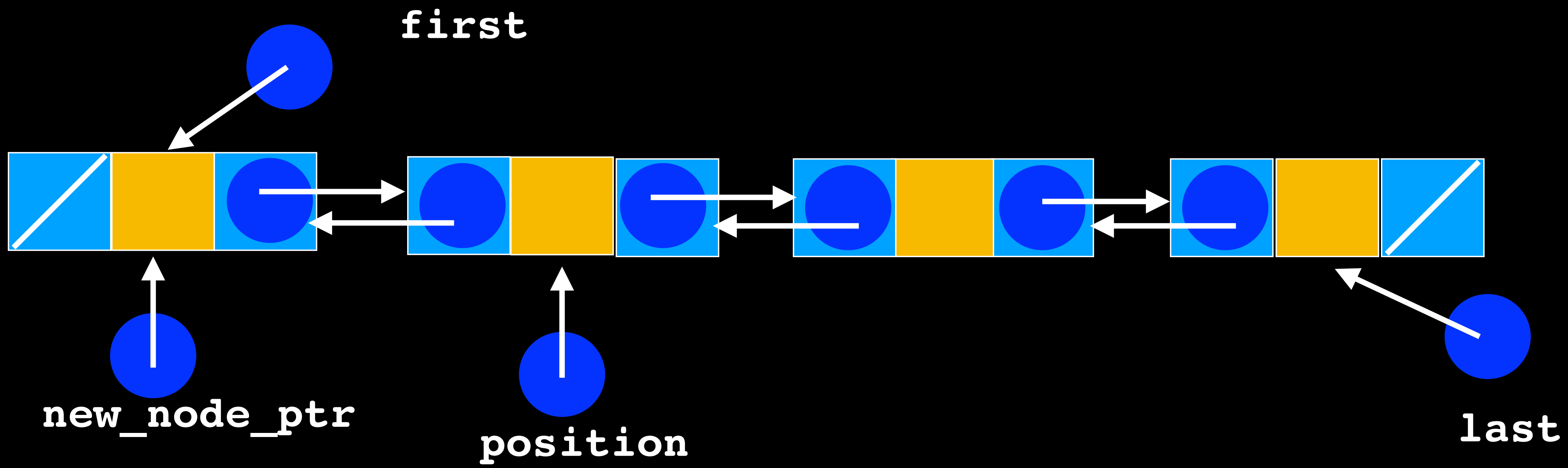
```
if (position == first)
      {
         // Insert new node at beginning of chain
         new_node_ptr->setNext(first);
         new_node_ptr->setPrevious(nullptr);
         first->setPrevious(new_node_ptr);
         first = new_node_ptr;
      }
```



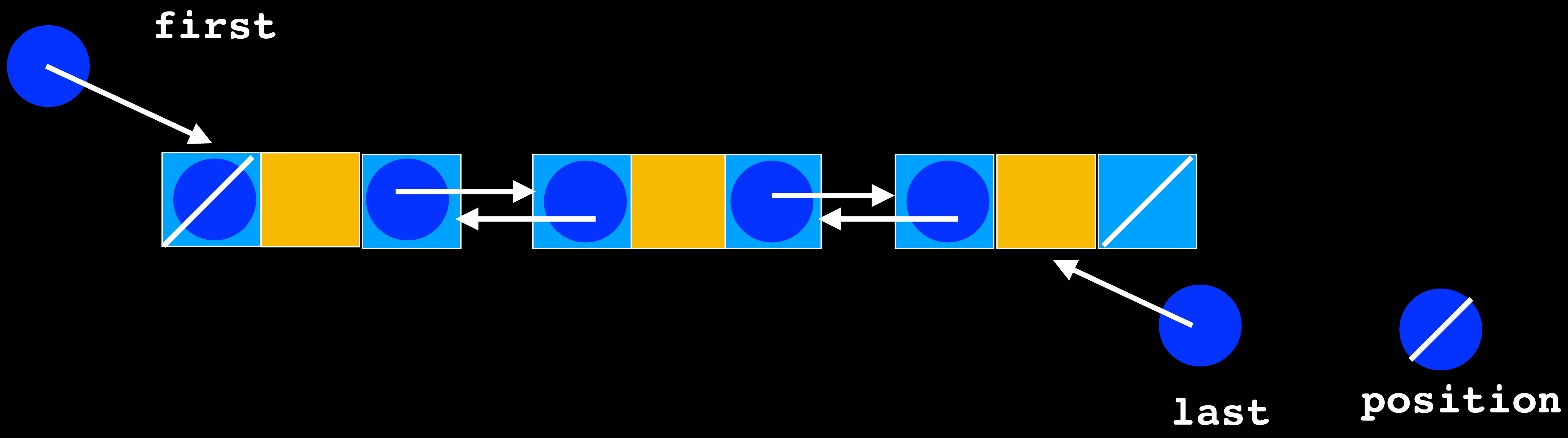
```
if (position == first)
     {
        // Insert new node at beginning of chain
        new_node_ptr->setNext(first);
        new_node_ptr->setPrevious(nullptr);
        first->setPrevious(new_node_ptr);
        first = new_node_ptr;
     }
```



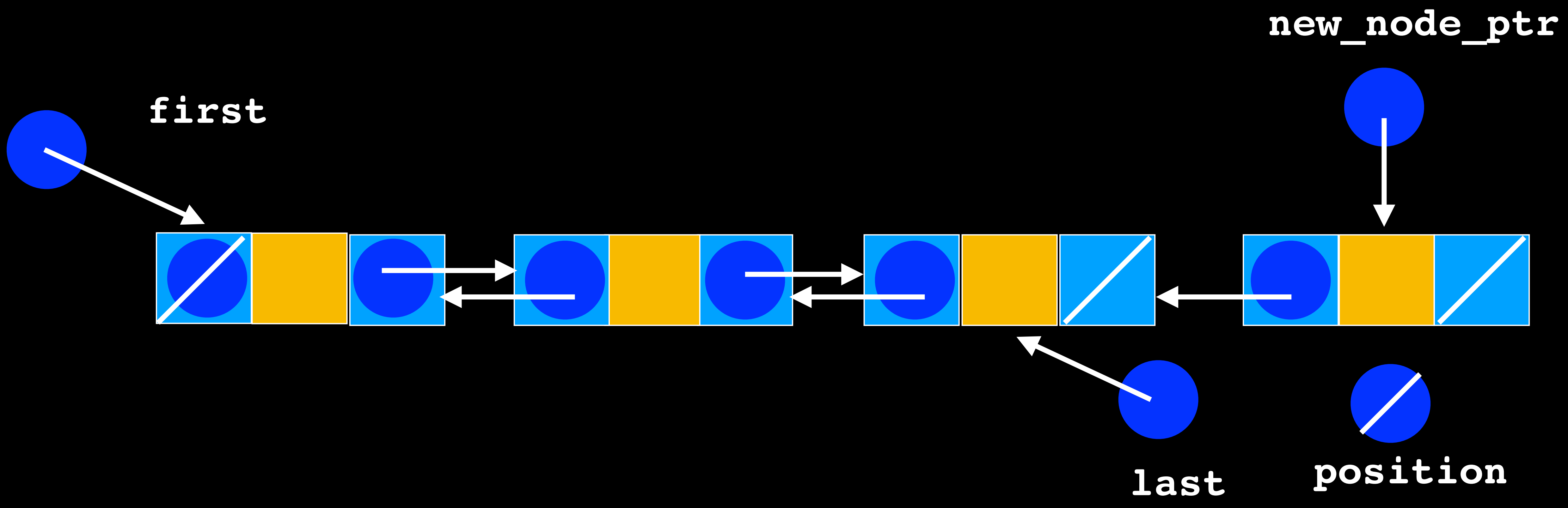
```
if (position == first)
     {
        // Insert new node at beginning of chain
        new_node_ptr->setNext(first);
        new_node_ptr->setPrevious(nullptr);
        first->setPrevious(new_node_ptr);
        first = new_node_ptr;
     }
```



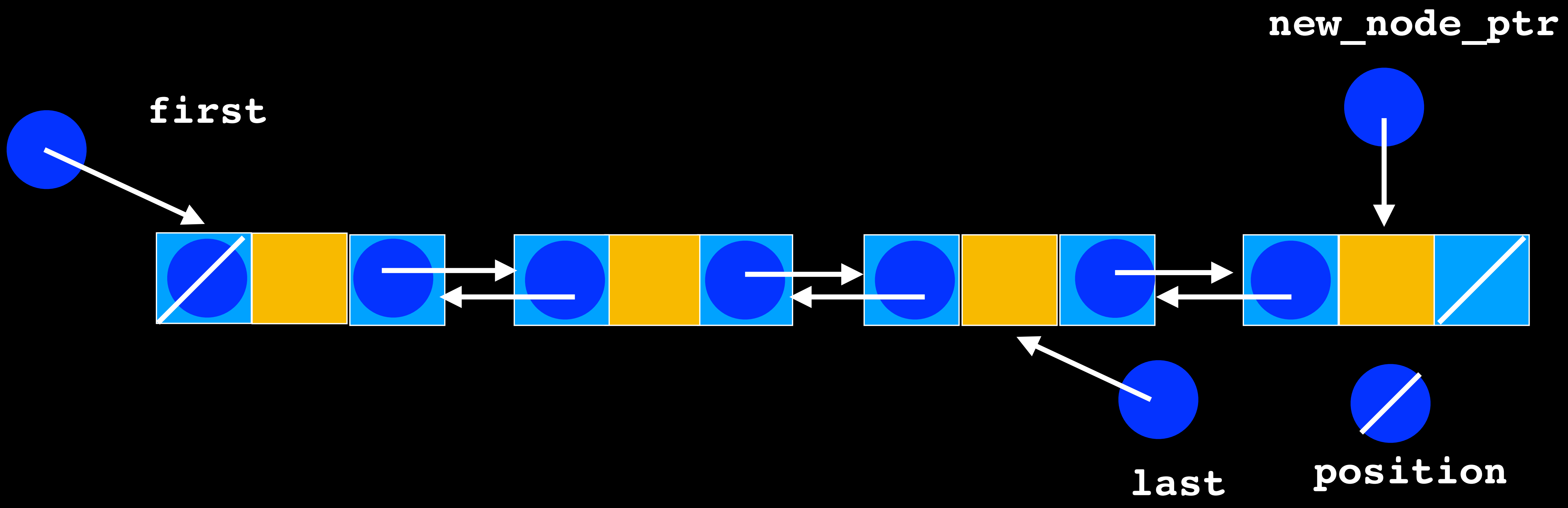
```
else if (position == nullptr)
{
    //insert at end of list
    new_node_ptr->setNext(nullptr);
    new_node_ptr->setPrevious(last);
    last->setNext(new_node_ptr);
    last = new_node_ptr;
}
```



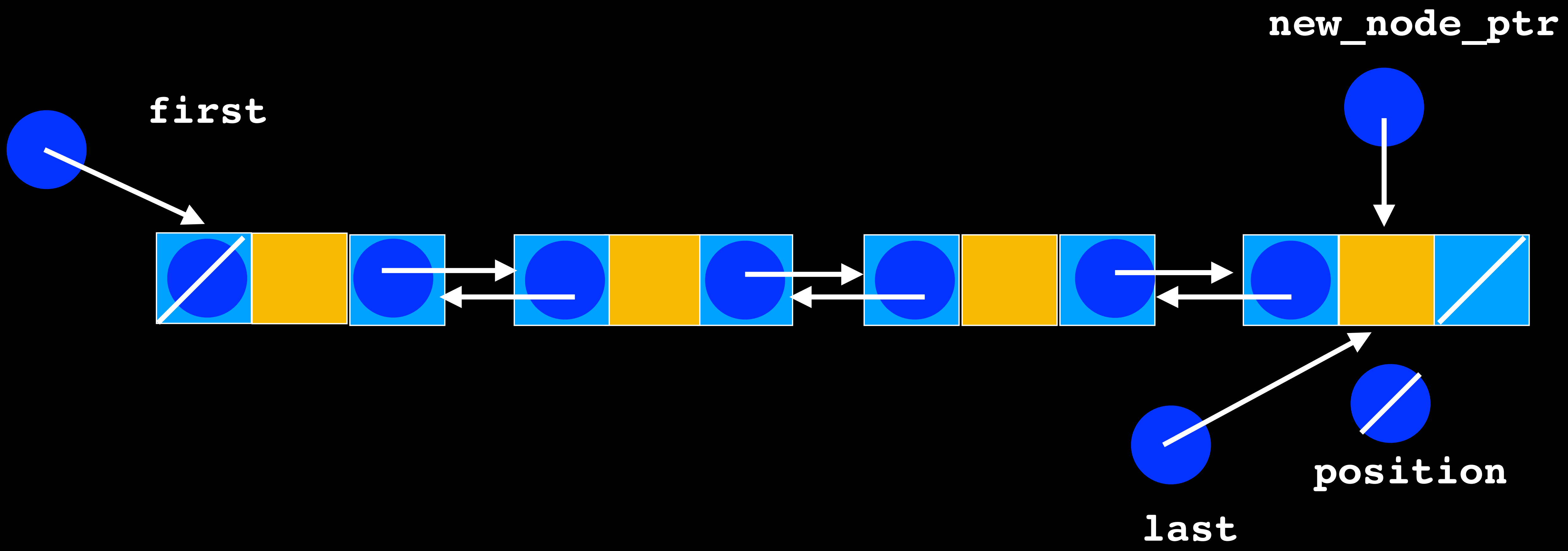
```
else if (position == nullptr)
{
    //insert at end of list
    new_node_ptr->setNext(nullptr);
    new_node_ptr->setPrevious(last);
    last->setNext(new_node_ptr);
    last = new_node_ptr;
}
```



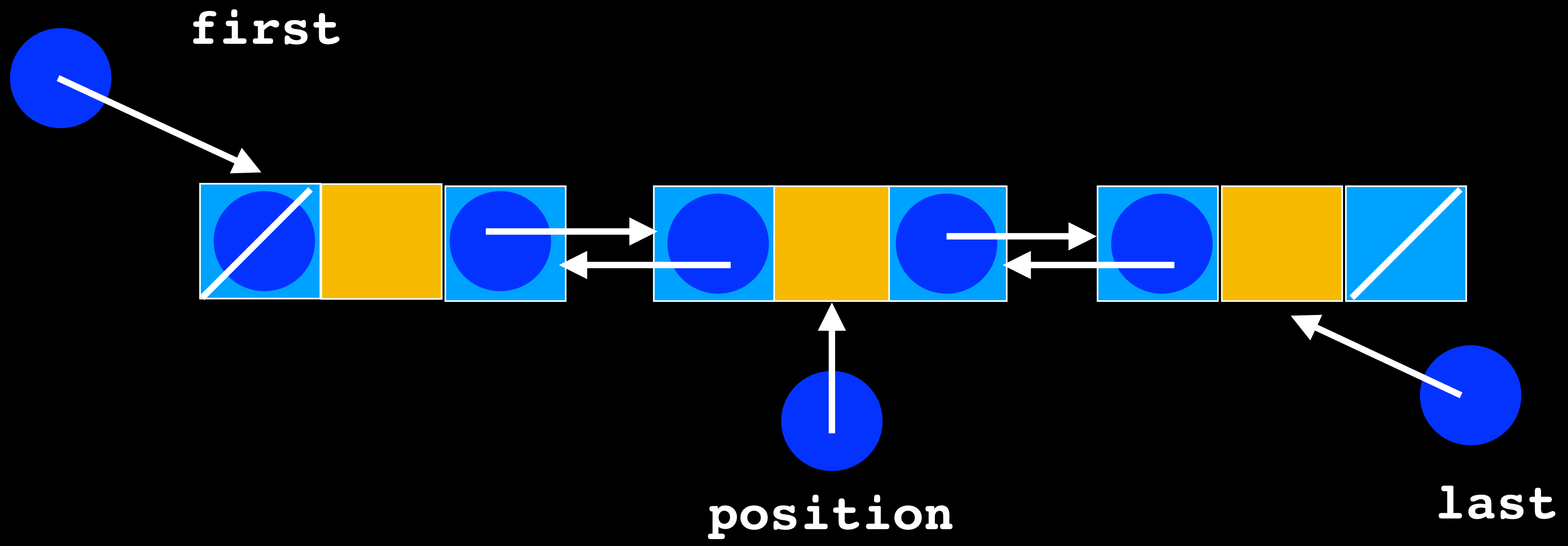
```
else if (position == nullptr)
{
    //insert at end of list
    new_node_ptr->setNext(nullptr);
    new_node_ptr->setPrevious(last);
    last->setNext(new_node_ptr);
    last = new_node_ptr;
}
```



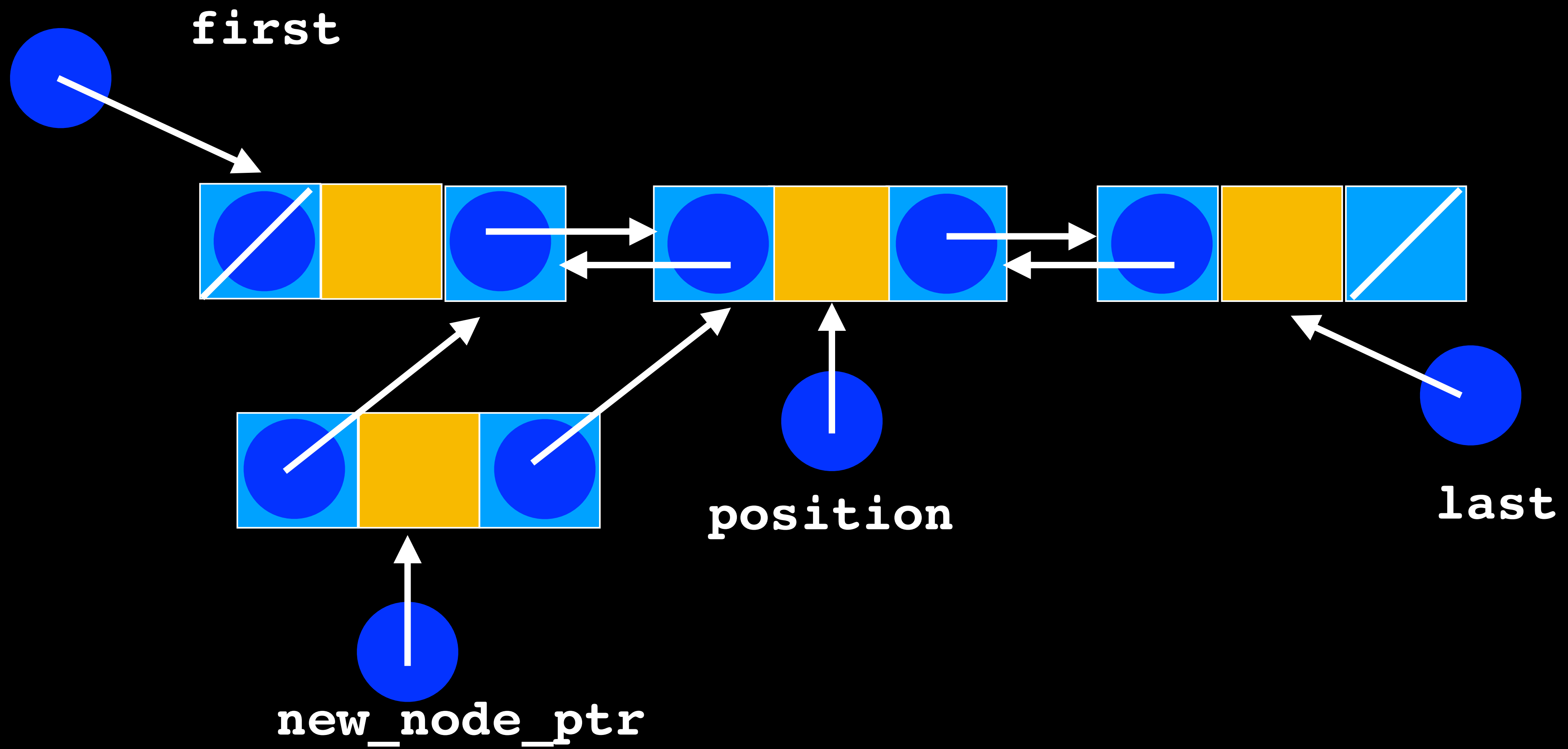
```
else if (position == nullptr)
{
    //insert at end of list
    new_node_ptr->setNext(nullptr);
    new_node_ptr->setPrevious(last);
    last->setNext(new_node_ptr);
    last = new_node_ptr;
}
```



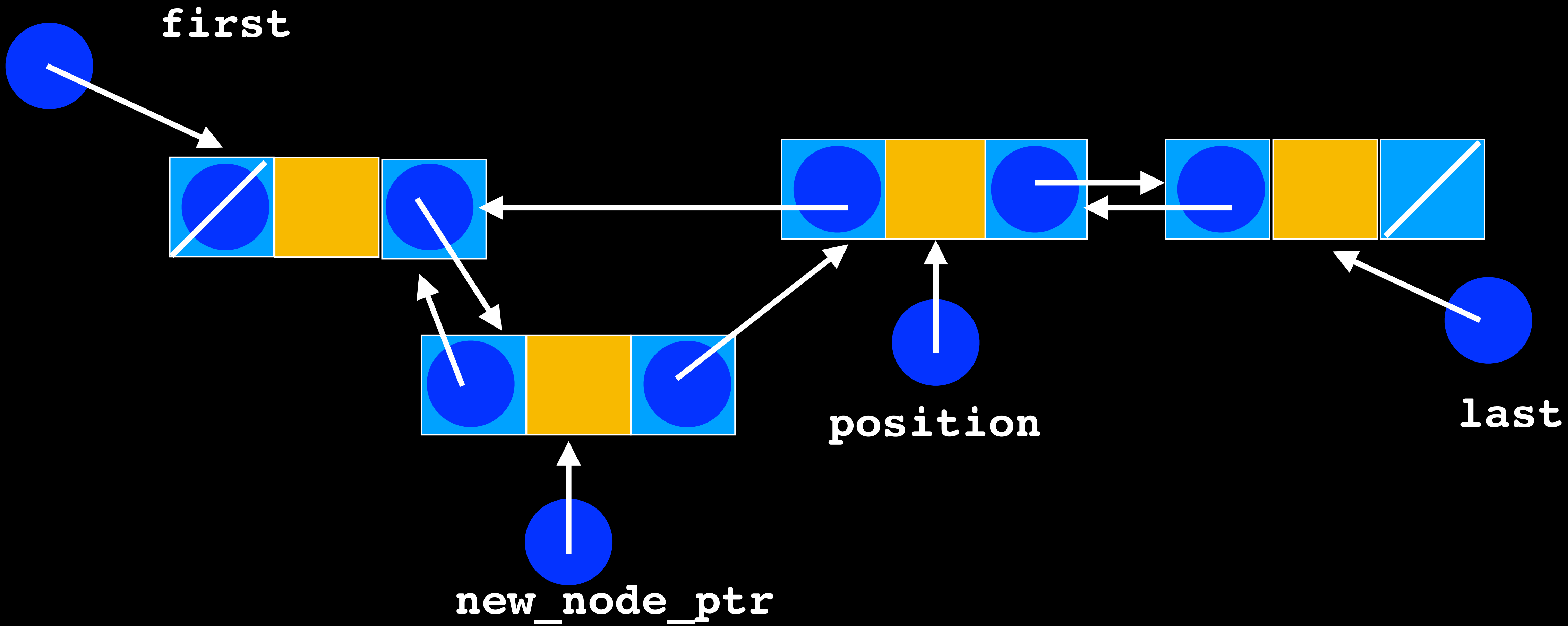
```
else
    {
       // Insert new node before node to which position points
       new_node_ptr->setNext(position);
       new_node_ptr->setPrevious(position->getPrevious());
       position->getPrevious()->setNext(new_node_ptr);
       position->setPrevious(new_node_ptr);
    }  // end if
```



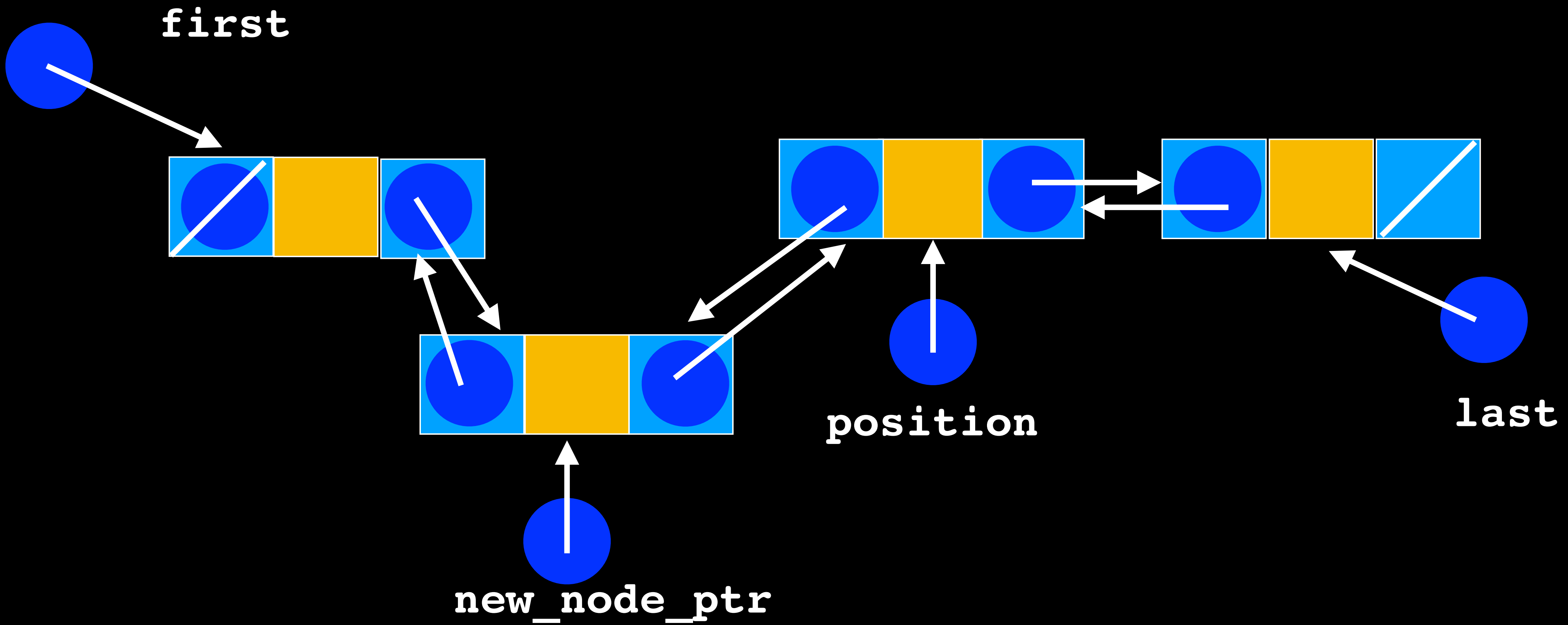
```
else
    {
       // Insert new node before node to which position points
       new_node_ptr->setNext(position);
       new_node_ptr->setPrevious(position->getPrevious());
       position->getPrevious()->setNext(new_node_ptr);
       position->setPrevious(new_node_ptr);
    }  // end if
```



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



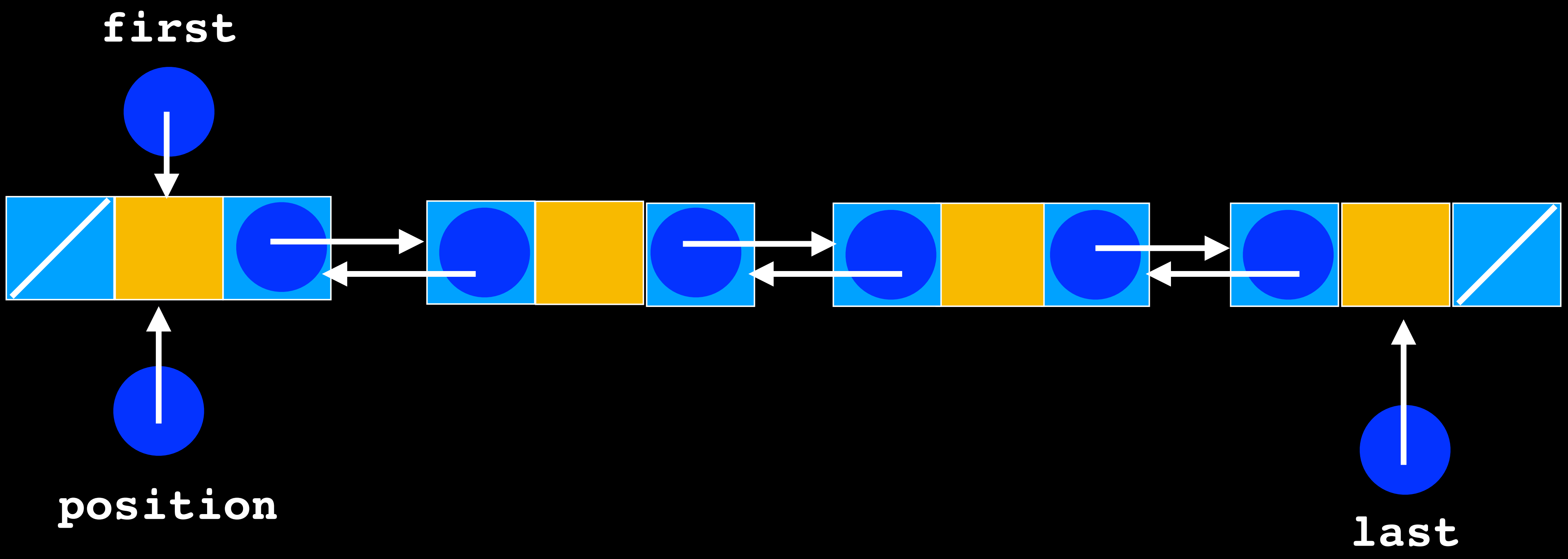
```
else
    {
       // Insert new node before node to which position points
       new_node_ptr->setNext(position);
       new_node_ptr->setPrevious(position->getPrevious());
       position->getPrevious()->setNext(new_node_ptr);
       position->setPrevious(new_node_ptr);
    }  // end if
```



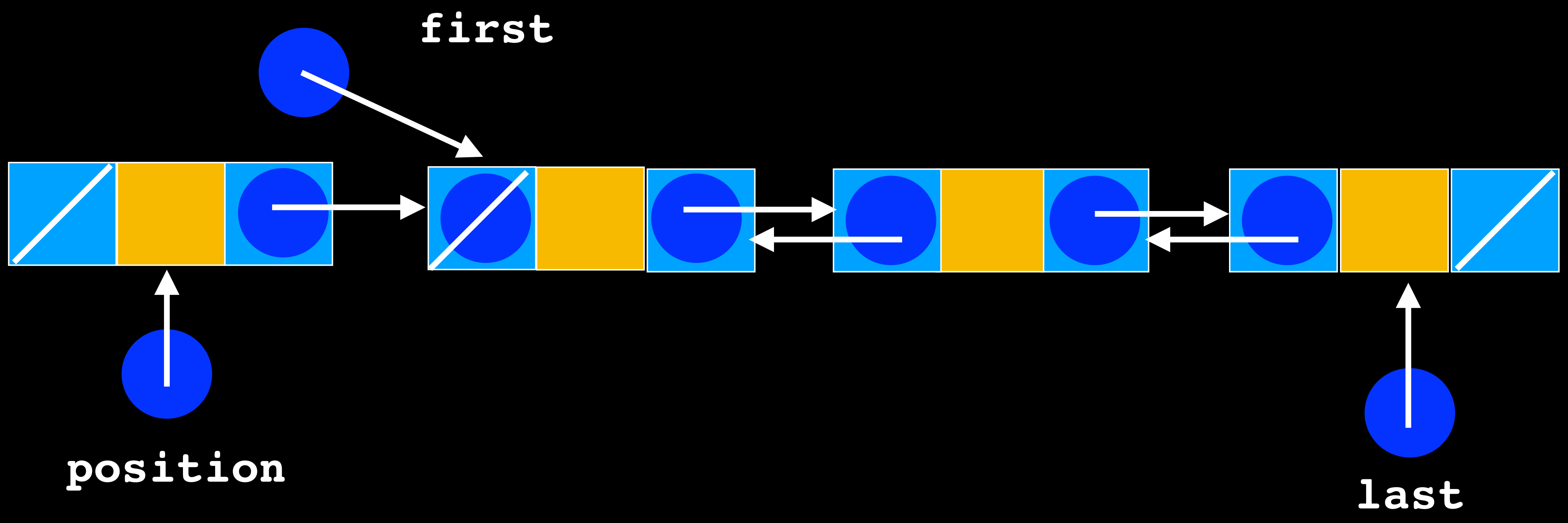
List::Remove

```
template<class ItemType>
void List<ItemType>::remove(Node<ItemType>* position)
{
    // Remove node from chain
    if (position == first)
    {
        // Remove first node
        first = position->getNext();
        first->setPrevious(nullptr);

        // Return node to the system
        position->setNext(nullptr);
        delete position;
        position = nullptr;
    }
    else if (position == last)
    {
        //remove last node
        last = position->getPrevious();
        last->setNext(nullptr);

        // Return node to the system
        position->setPrevious(nullptr);
        delete position;
        position = nullptr;
    }
    else
    {
        //Remove from the middle
        position->getPrevious()->setNext(position->getNext());
        position->getNext()->setPrevious(position->getPrevious());

        // Return node to the system
        position->setNext(nullptr);
        position->setPrevious(nullptr);
        delete position;
        position = nullptr;

    }  // end if

    item_count--;  // decrease count of entries
}  // end remove
```

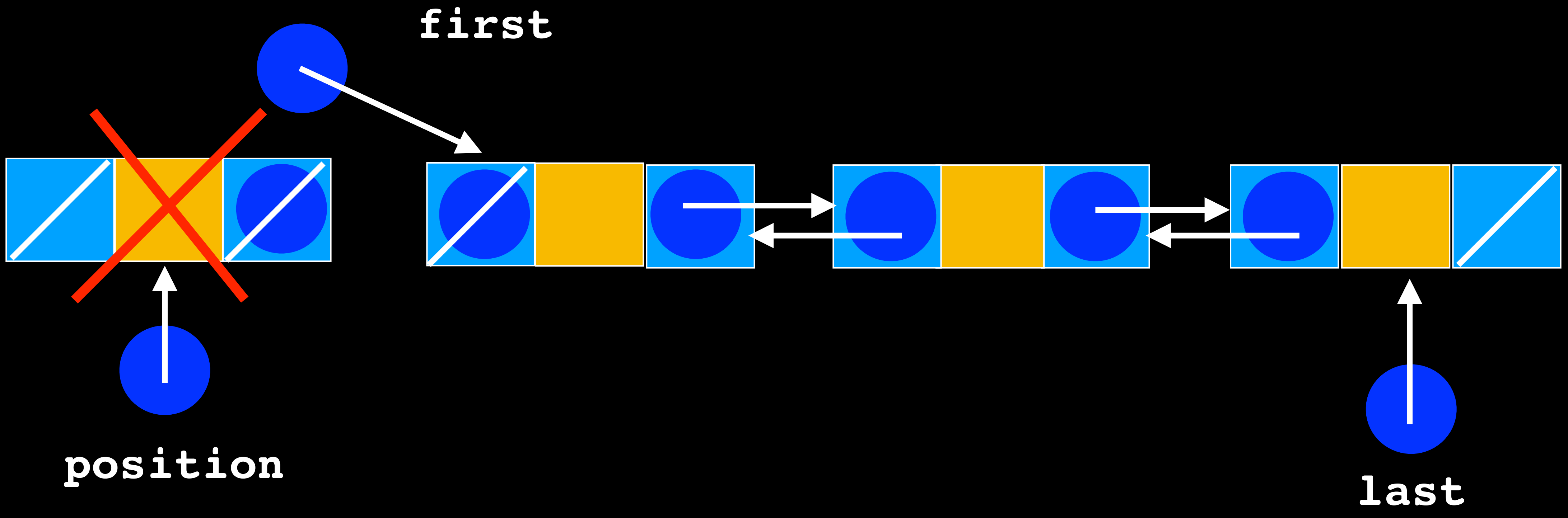
```
// Remove node from chain
if (position == first)
{
    // Remove first node
    first = position->getNext();
    first->setPrevious(nullptr);

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



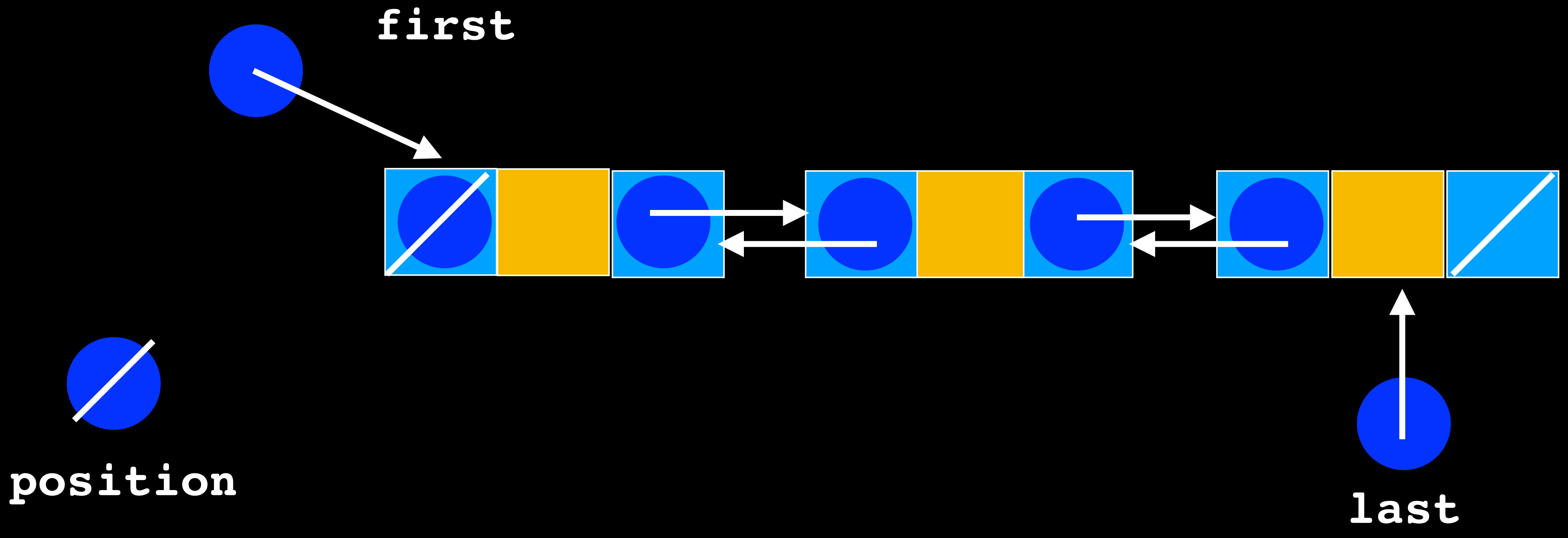
```
// Remove node from chain
if (position == first)
{
    // Remove first node
    first = position->getNext();
    first->setPrevious(nullptr);

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



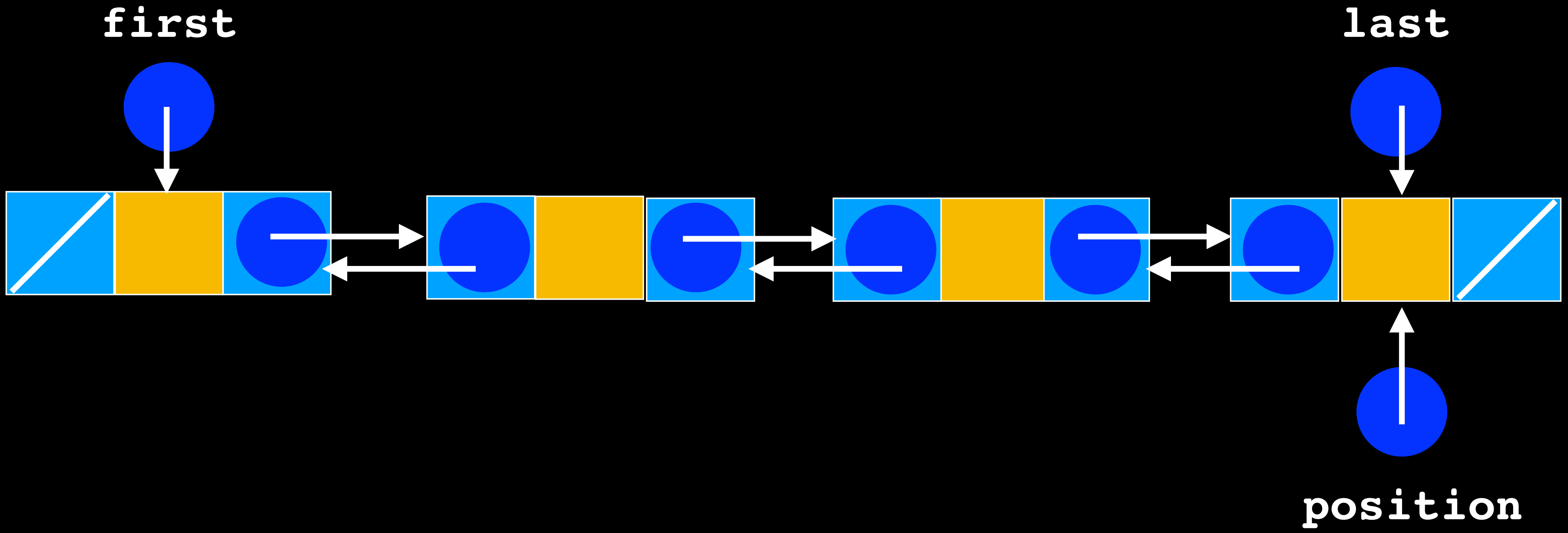
```
// Remove node from chain
if (position == first)
{
    // Remove first node
    first = position->getNext();
    first->setPrevious(nullptr);

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



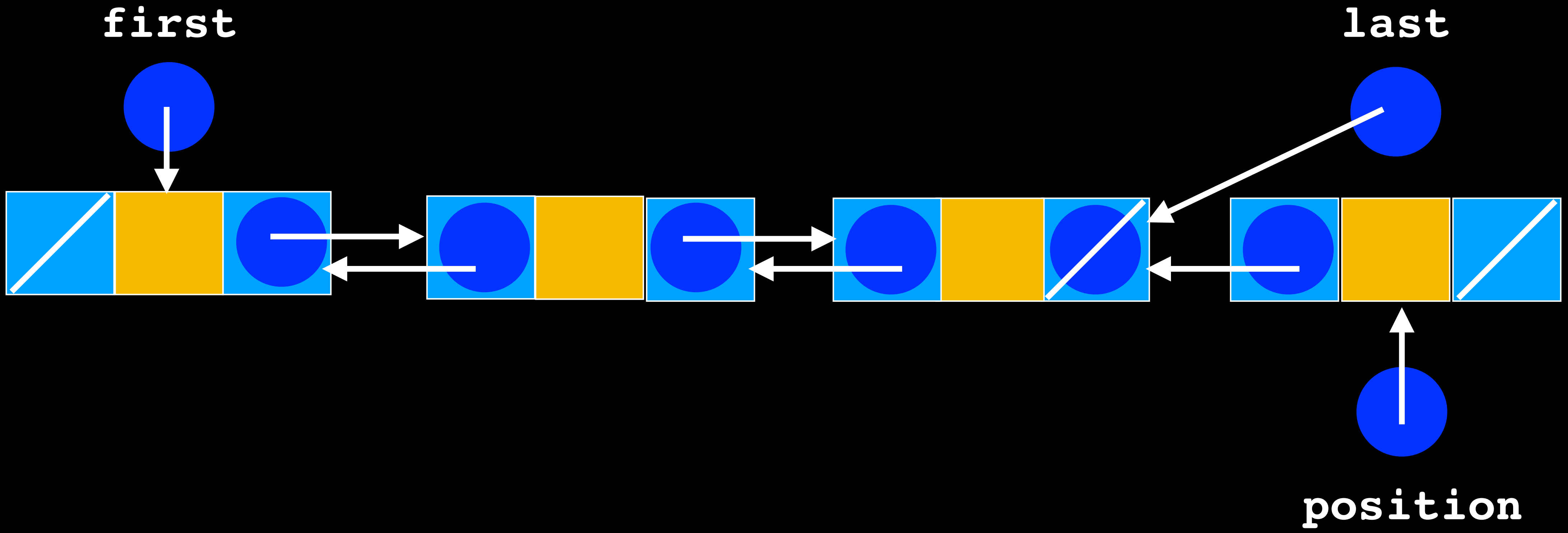
```
// Remove node from chain
if (position == first)
{
    // Remove first node
    first = position->getNext();
    first->setPrevious(nullptr);

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



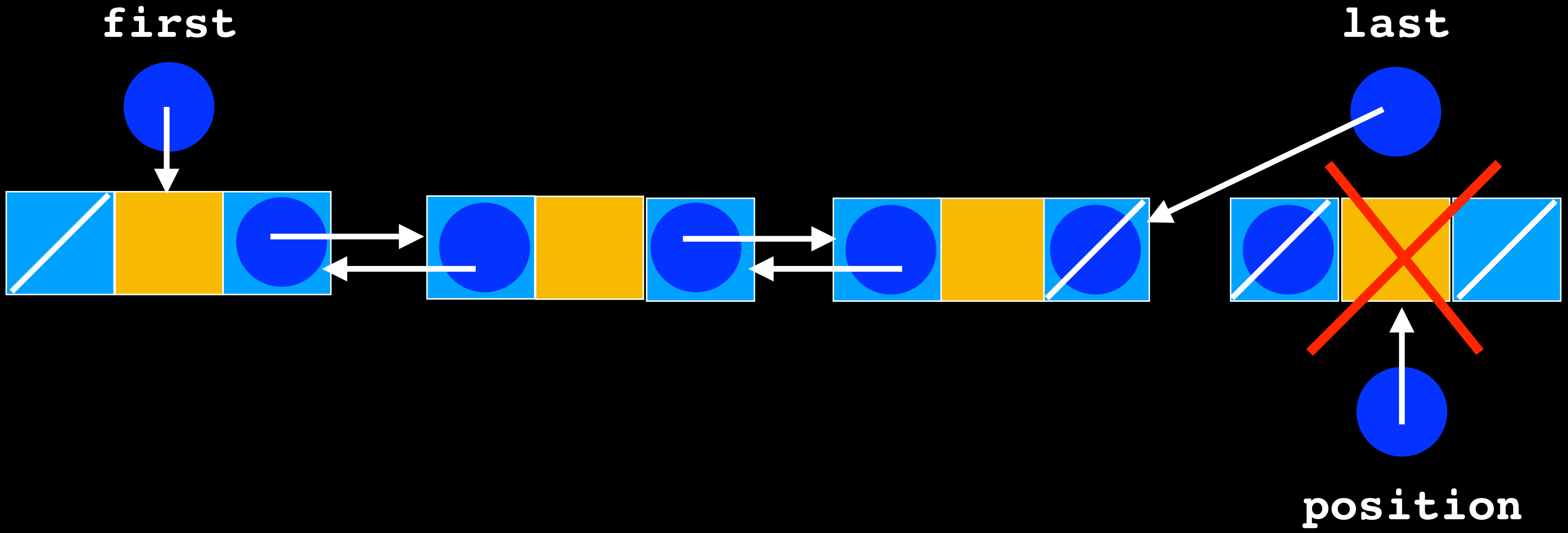
```
else if (position == last)
{
    //remove last node
    last = position->getPrevious();
    last->setNext(nullptr);

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



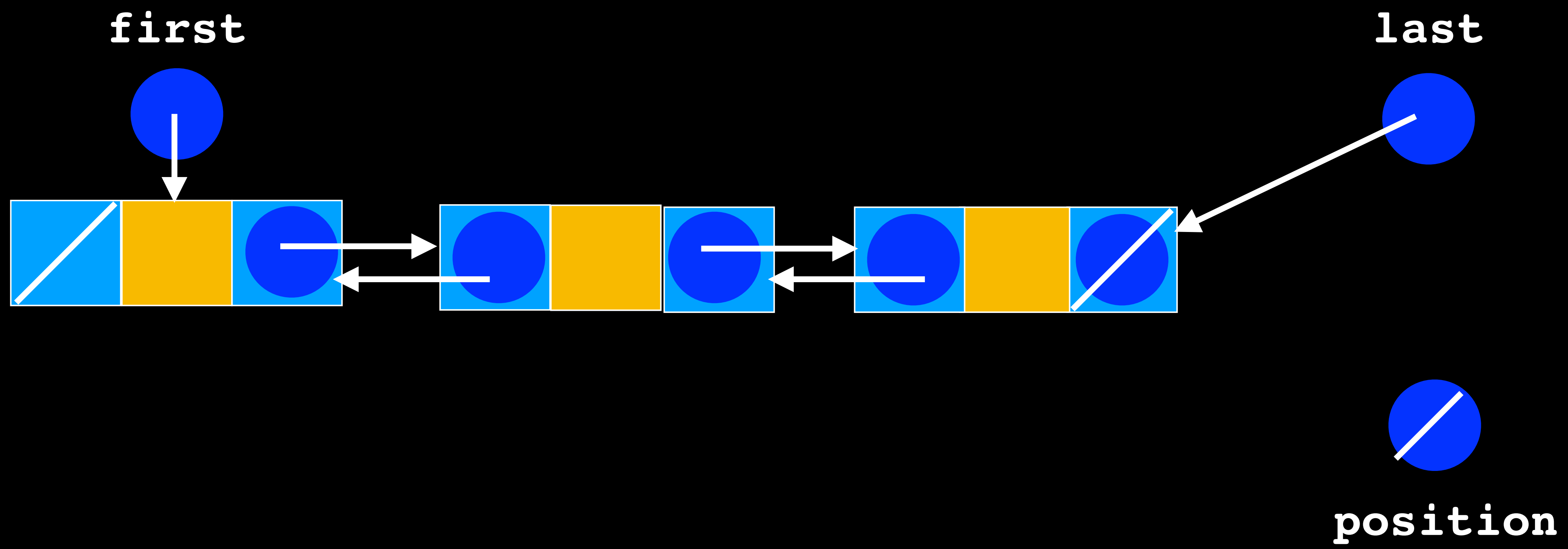
```
else if (position == last)
{
    //remove last node
    last = position->getPrevious();
    last->setNext(nullptr);

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



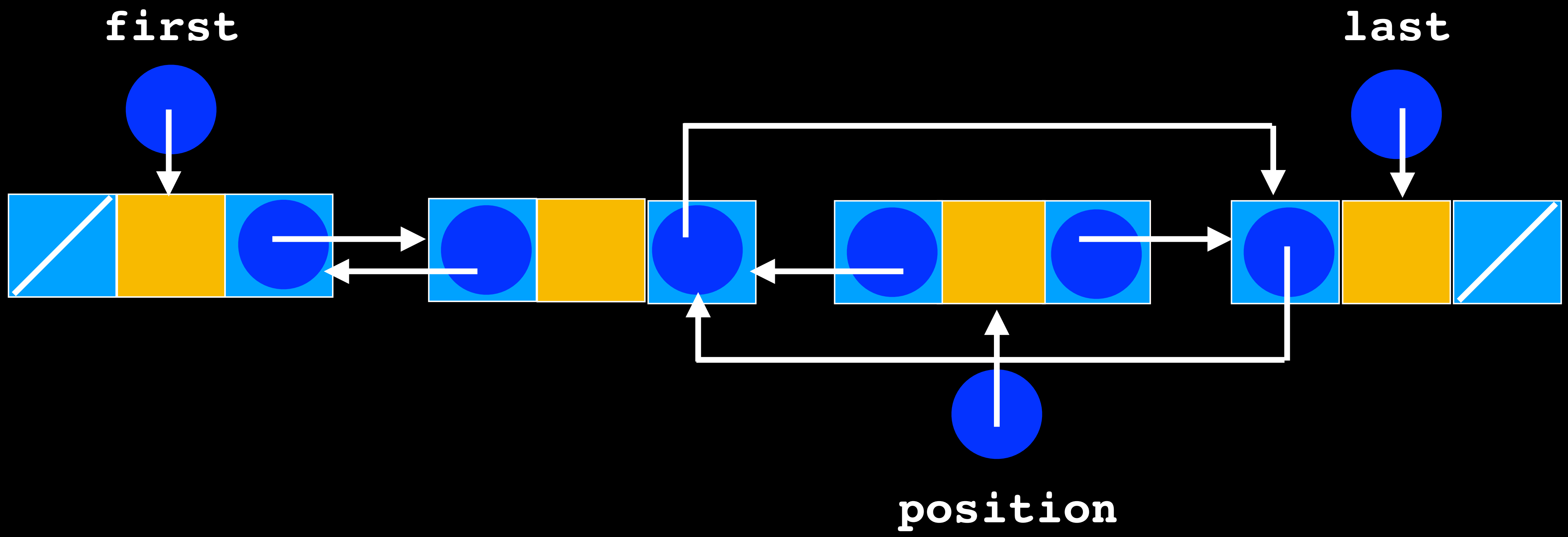
```
else if (position == last)
{
     //remove last node
     last = position->getPrevious();
     last->setNext(nullptr);

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



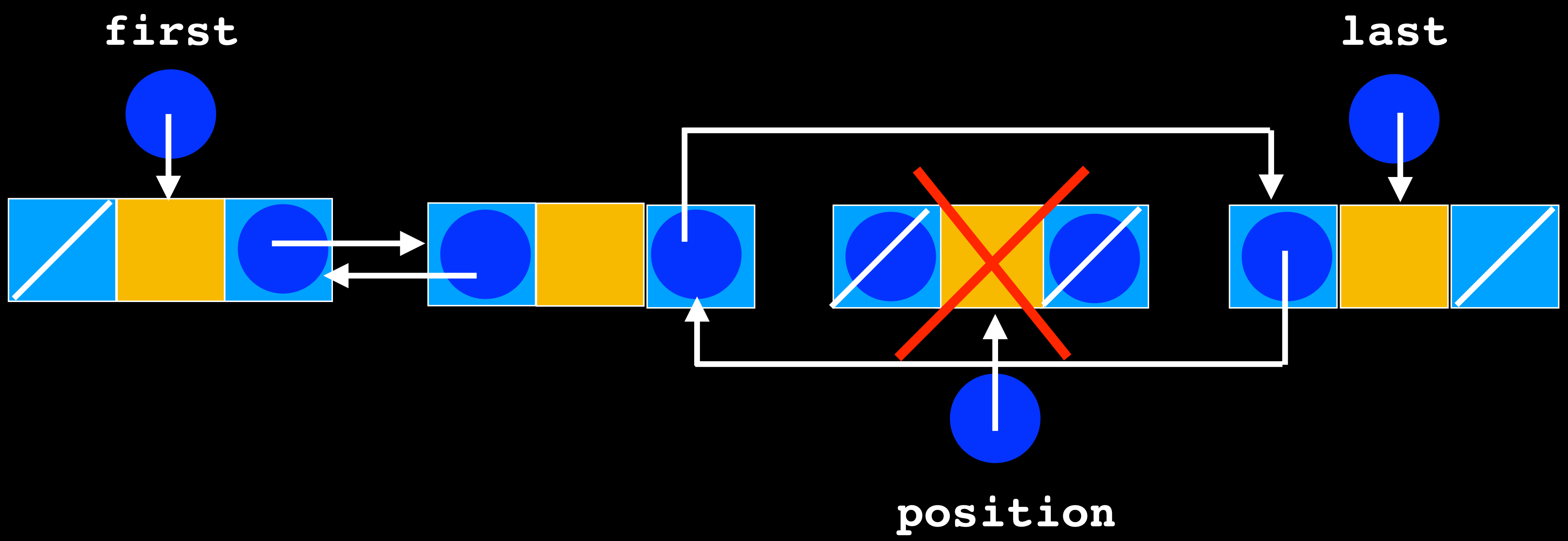
```
else if (position == last)
{
    //remove last node
    last = position->getPrevious();
    last->setNext(nullptr);

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



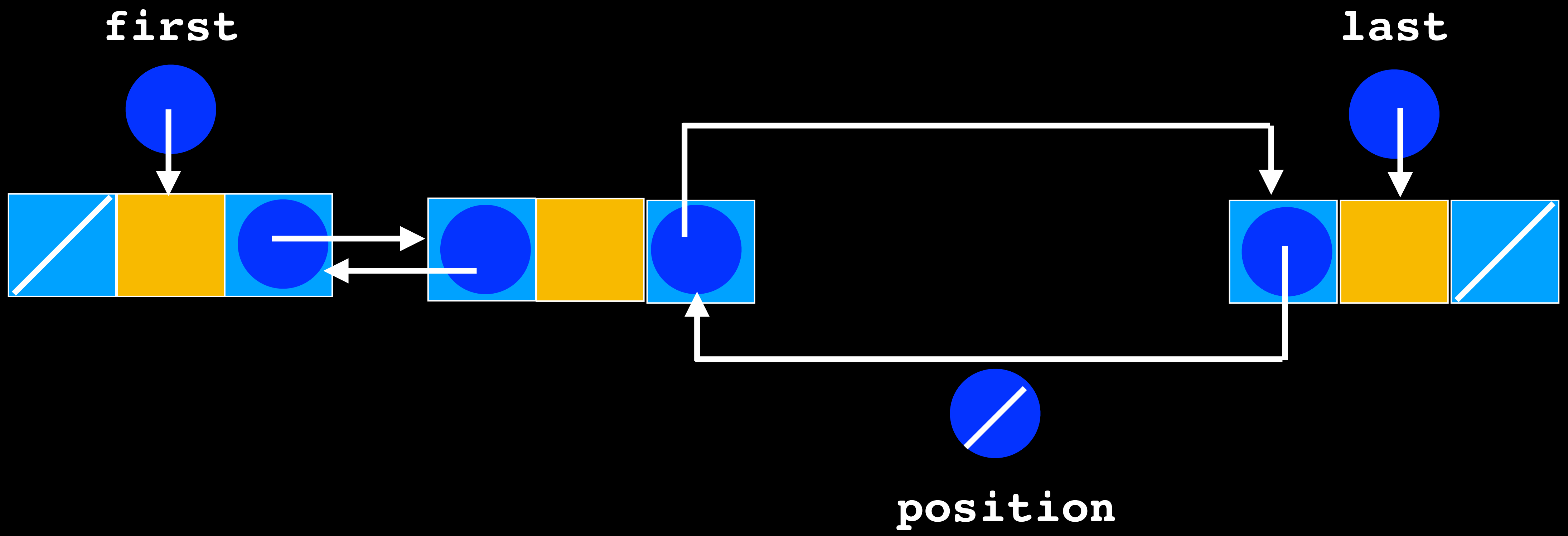
```
else
{
    //Remove from the middle
    position->getPrevious()->setNext(position->getNext());
    position->getNext()->setPrevious(position->getPrevious());

    // Return node to the system
    position->setNext(nullptr);
    position->setPrevious(nullptr);
    delete position;
    position = nullptr;

}  // end if
```



```
else
{
    //Remove from the middle
    position->getPrevious()->setNext(position->getNext());
    position->getNext()->setPrevious(position->getPrevious());

    // Return node to the system
    position->setNext(nullptr);
    position->setPrevious(nullptr);
    delete position;
    position = nullptr;

}  // end if
```



```
else
{
    //Remove from the middle
    position->getPrevious()->setNext(position->getNext());
    position->getNext()->setPrevious(position->getPrevious());

    // Return node to the system
    position->setNext(nullptr);
    position->setPrevious(nullptr);
    delete position;
    position = nullptr;

}  // end if
```



List::getPointerTo

```
template<class ItemType>
Node<ItemType>* List<ItemType>::getPointerTo(std::size_t position) const
throw(std::out_of_range)
{

    Node<ItemType>* find = nullptr;
    if(position >= item_count)
        throw std::out_of_range("position is larger than the current size
of the list.");
    else
    {
        find = first;
        for(size_t i = 0; i < position; ++i)
        {
            find = find->getNext();
        }
    }

    return find;
}//end getPointerTo
```