CS 11: Introduction to Computer Science

Link List Details

Mark Sheldon
Tufts University
Email: msheldon@cs.tufts.edu
Web: http://www.cs.tufts.edu/~msheldon

Noah Mendelsohn
Tufts University
Email: noah@cs.tufts.edu
Web: http://www.cs.tufts.edu/~noah

Goals for this session

- **Consider again the significance of lists and linked lists**
- **Deep dive on coding link lists**
- **Just for fun: other kinds of linked lists**

Review
What are Lists? Why do we care?

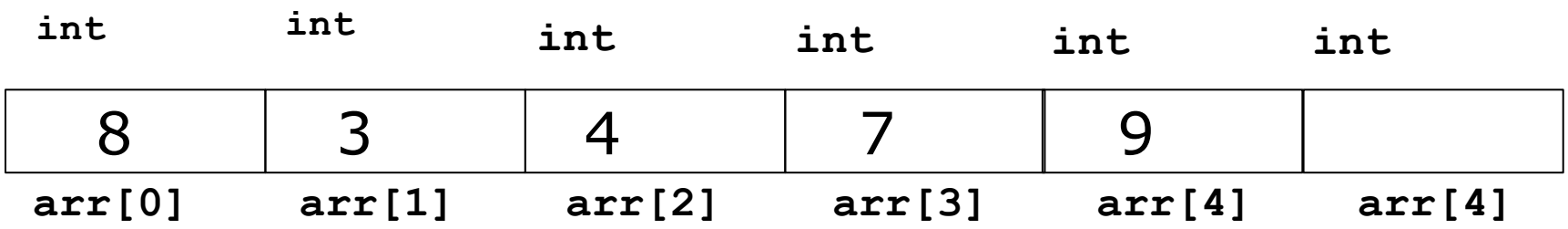
Review – Typical list interfaces let you:

- **Add items to the list at the beginning and/or end**
- **Remove items from the beginning and/or end**
- **Get to all the items by index in the current list**
- **Sometimes:**
  - Remove items from the middle of the list
  - Insert items in the middle of the list
  - Etc.

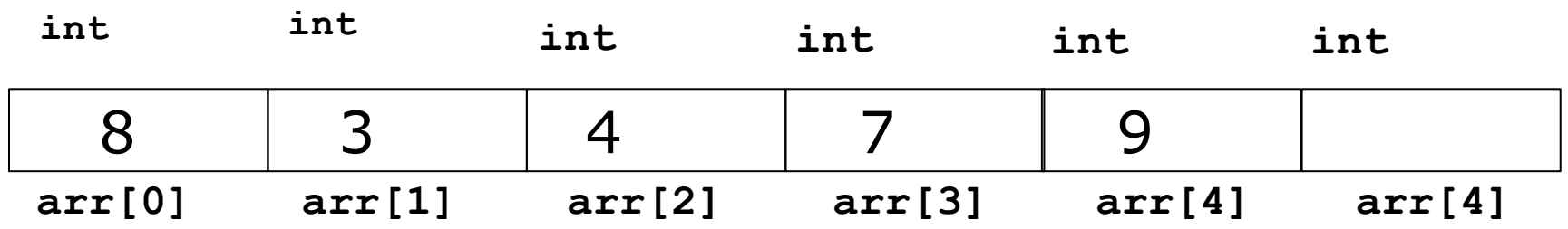
DISCUSSION (Review)

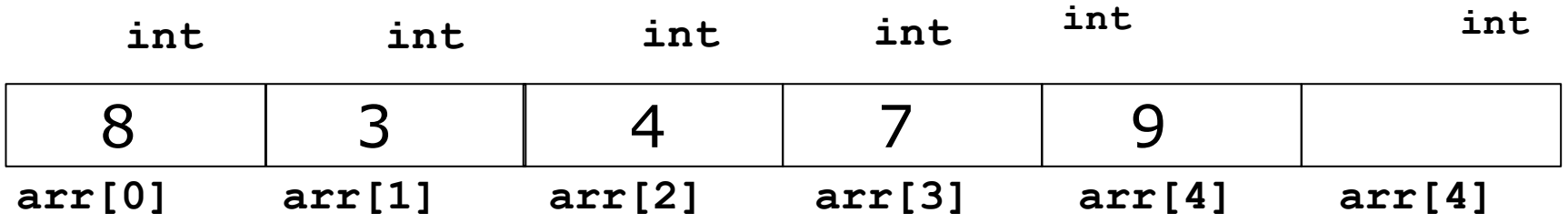
What would happen if we tried to add these new capabilities using our array-based implementation?

Removing from the front of an array-based list



Removing from the front of an array-based list

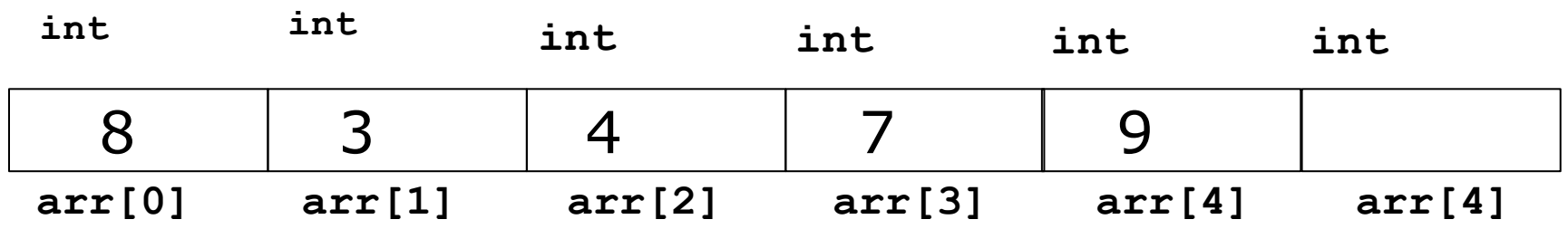


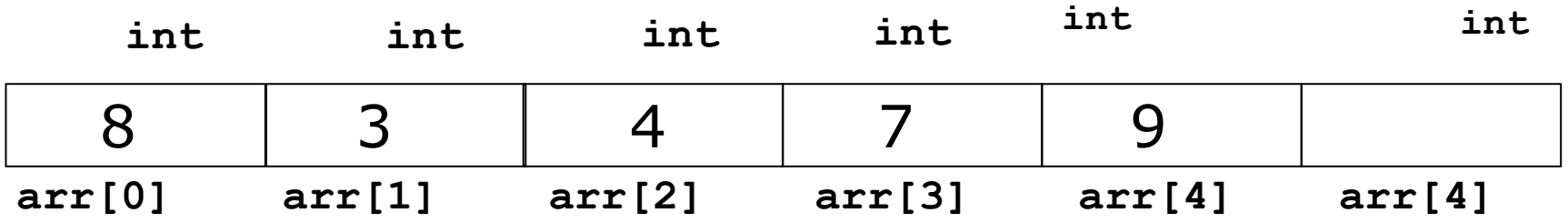


Question: if a list has 10,000 items and we remove them all from the front…how many total moves are done?

Answer: 9,999 + 9998 + 9997 + …. + 1 = 49,995,000

Adding at the end of an array-based list





99

Implementing lists with arrays

- **Adding at the end is:**
  - Easy?
  - Hard?
- **Adding at the beginning is:**
  - Easy?
  - Hard?
- **Getting a value by index is:**
  - Easy?
  - Hard?
- **Deleting at the end is:**
  - Easy?
  - Hard?
- **Deleting at the beginning is:**
  - Easy?
  - Hard?
- **Inserting / deleting in the middle is:**
  - Easy?
  - Hard?

Introducing Linked Lists

Linked lists

- **Provide many of the same services as array lists**
- **Use *pointer manipulation* to edit the structure of the list**
- **Typically use dynamic memory allocation (new / delete) for nodes in the list**
- **Compared to array lists:**
  - Tend to be more flexible
  - Have more space overhead for pointers
  - Can't be directly subscripted like C++ arrays

How to represent a linked list?

- **A (linked) list is either:**
  - Empty OR
  - Non-empty, with
    - a first element AND
    - a (linked) list of elements after the first element

How to represent a linked list?

- **A (linked) list is either:**
  - Empty OR
  - Non-empty, with
    - a first element AND
    - a (linked) list of elements after the first element

- **Representation**
  - `nullptr`

How to represent a linked list?

- **A (linked) list is either:**
  - Empty OR
  - Non-empty, with
    - a first element AND
    - a (linked) list of elements after the first element

- **Representation**
  - **`nullptr`** OR
  - pointer to

```
struct Node {
        Node         *next;
        ElementType   data;
};
```

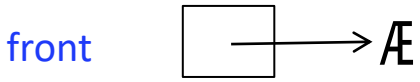
How to represent a linked list?

- **A (linked) list is either:**
  - Empty OR
  - Non-empty, with
    - a first element AND
    - a (linked) list of elements after the first element

- **A (linked) list is either:**
  - nullptr OR
  - pointer to

```
struct Node {
        Node         *next;
        ElementType   data;
};
```

An empty list



front *does NOT contain a node!*
It contains either nullptr OR the *address* of a node

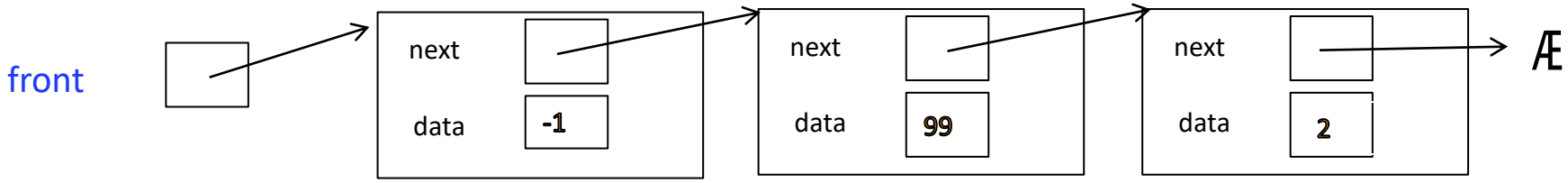
Node structs or classes will typically be on heap

- **A (linked) list is either:**
  - Empty OR
  - Non-empty, with
    - a first element AND
    - a (linked) list of elements after the first element

- **A (linked) list is either:**
  - nullptr OR
  - Pointer to

```
struct Node {
        Node *next;
        int   data;
};
```

A three-element int *list*



front *does NOT contain a node!*
It contains either nullptr OR the *address* of a node

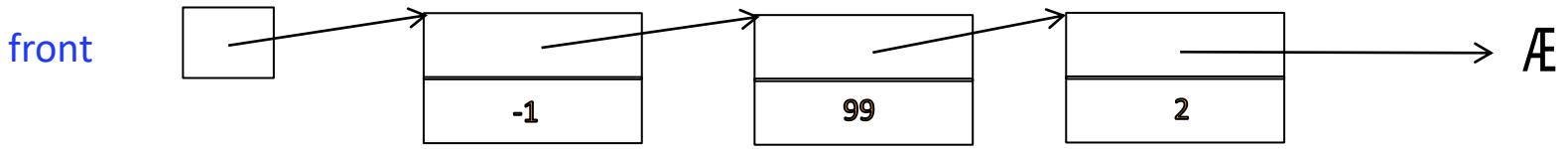
Linked lists: box-and-pointer list diagram

- **A (linked) list is either:**
  - Empty OR
  - Non-empty, with
    - a first element AND
    - a (linked) list of elements after the first element

- **A (linked) list is either:**
  - nullptr OR
  - pointer to

```
struct Node {
Node *next;
int  data;
};
```

A three-element int *list*



front *does NOT contain a node!*
It contains either nullptr OR the *address* of a node

Typical Linked List Interface

Review – Typical list interfaces let you:

- **Add items to the list at the beginning and/or end**
- **Remove items from the beginning and/or end**
- **Get to all the items by index in the current list**
- **Sometimes:**
  - Remove items from the middle of the list
  - Insert items in the middle of the list
  - Etc.

Let's Build a *Linked* List of Integers

The interface to our IntLinkedList class

Create / destroy list

```
 IntLinkedList();
~IntLinkedList();
```

Query size

```
bool isEmpty();
int  size();
```

Work with first element

```
void addToFront(int n);
void removeFirst();
```

Work with *any* element

```
int  getElement(int index);
void setElement(int index, int newValue);
```

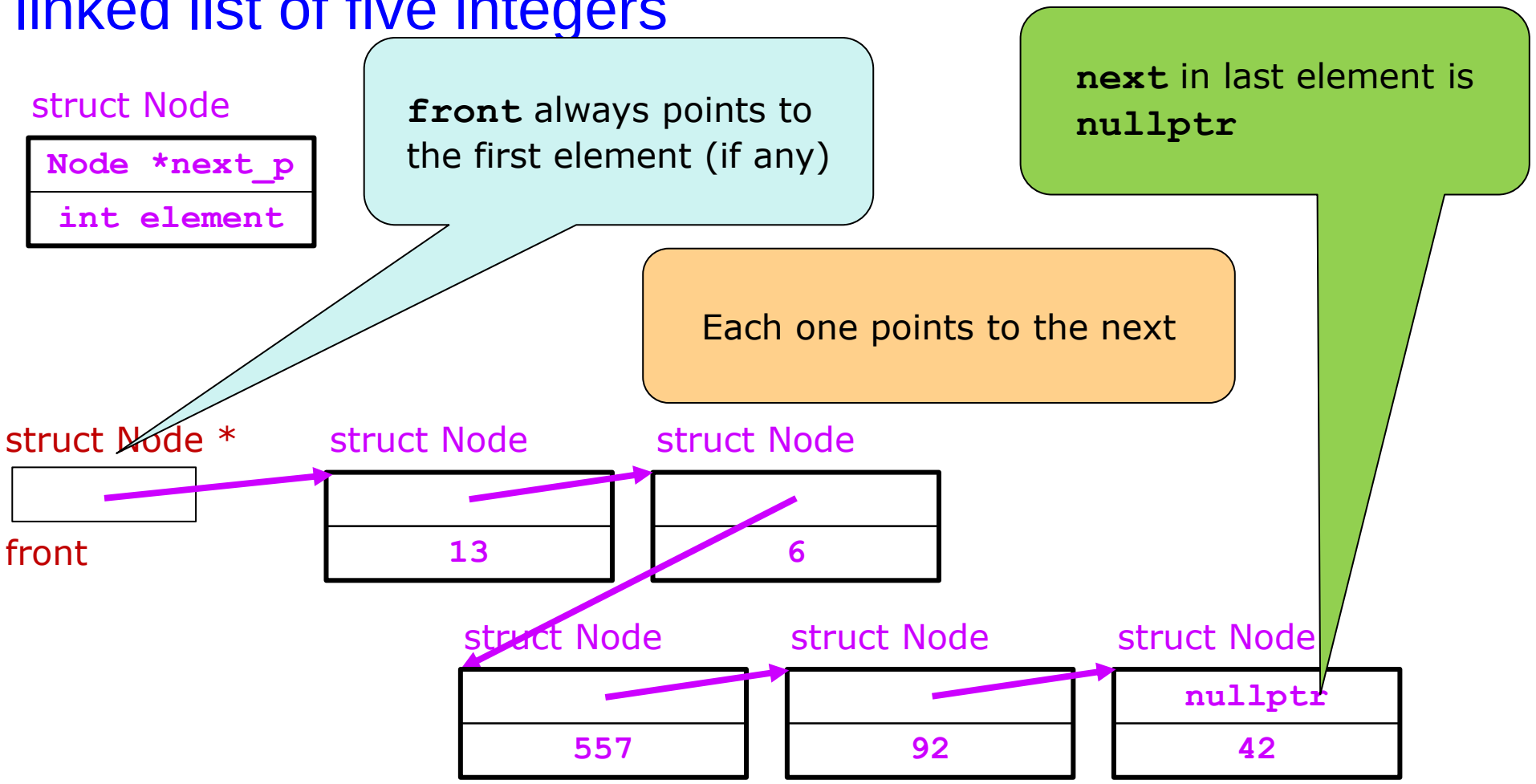
Work with last element

```
void addToBack(int n);
void addAtPosition(int n, int position);
```

…other features sometimes useful

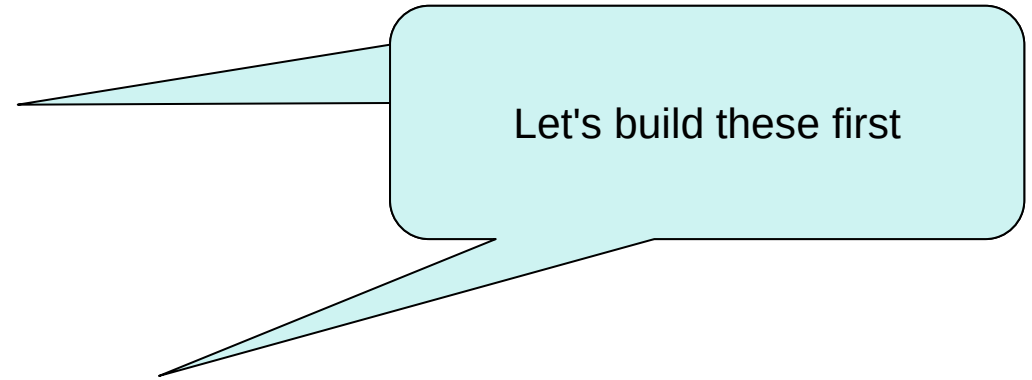
```
void print();
void printAnnotated();
```

A linked list of five integers



The interface to our IntLinkedList class

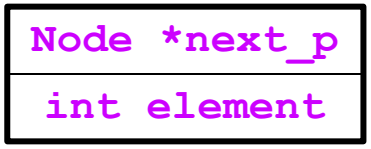
```
 IntLinkedList();
~IntLinkedList();

bool isEmpty();
int  size();

void addToFront(int n);
void removeFirst();

int  getElement(int index);
void setElement(int index, int newValue);

void addToBack(int n);
void addAtPosition(int n, int position);

void print();
void printAnnotated();
```



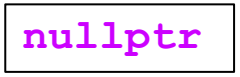
Constructor: IntLinkedList()

What should an empty list look like?

struct Node



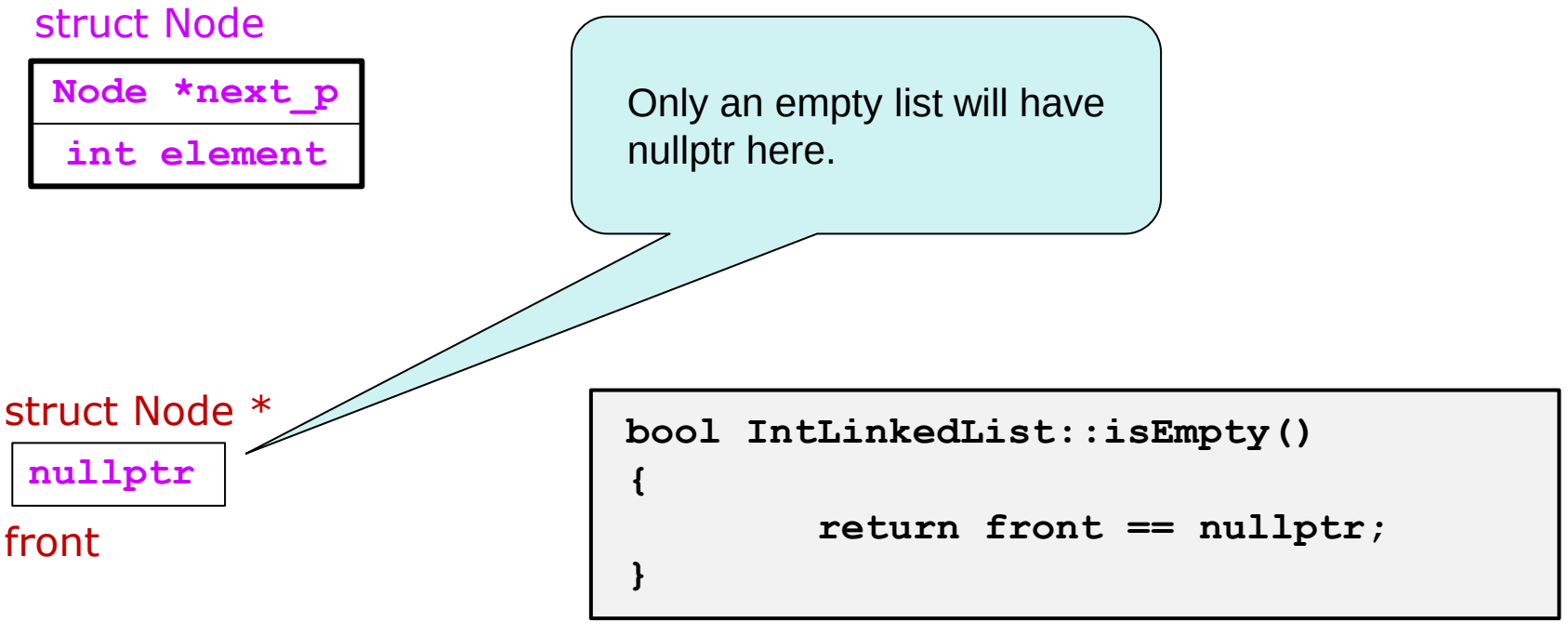
struct Node *



front

```
IntLinkedList::IntLinkedList()
{
        front = nullptr;
}
```

Check if list is empty




```
bool IntLinkedList::isEmpty()
{
        return front == nullptr;
}
```

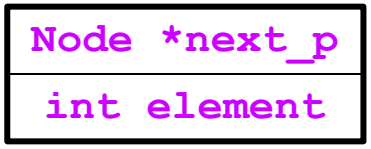
The interface to our IntLinkedList class

```
 IntLinkedList();
~IntLinkedList();

bool isEmpty();
int  size();

void addToFront(int n);
void removeFirst();

int  getElement(int index);
void setElement(int index, int newValue);

void addToBack(int n);
void addAtPosition(int n, int position);

void print();
void printAnnotated();
```

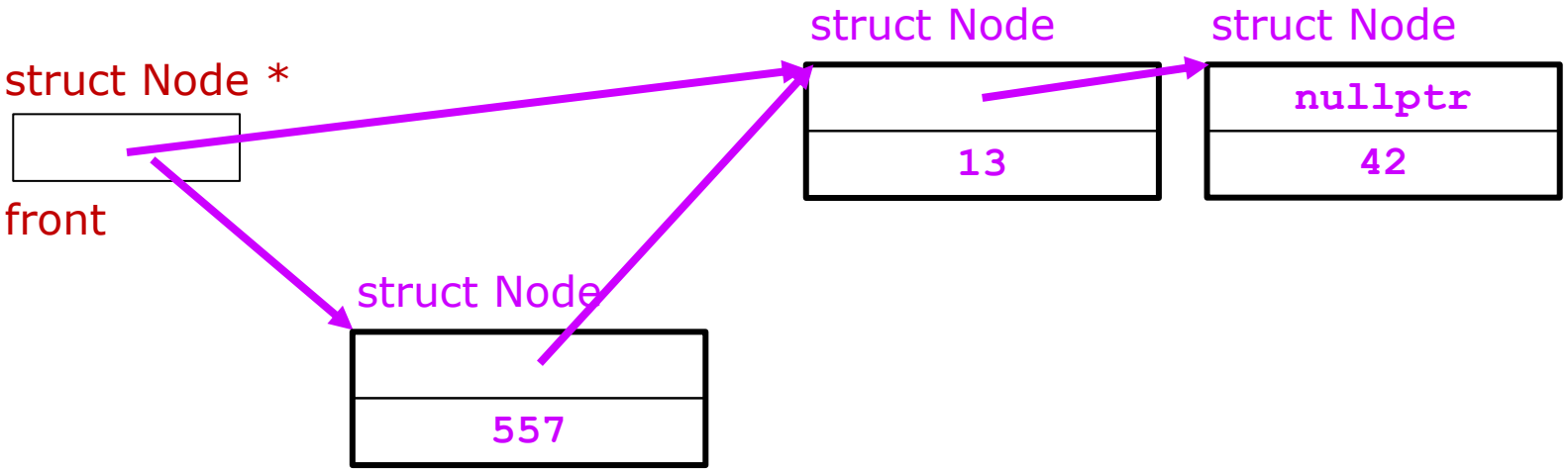
Adding at the front

somelist.addToFront(557)

struct Node




```
void IntLinkedList::addToFront(int n)
{
        front = new Node{front, n};
}
```



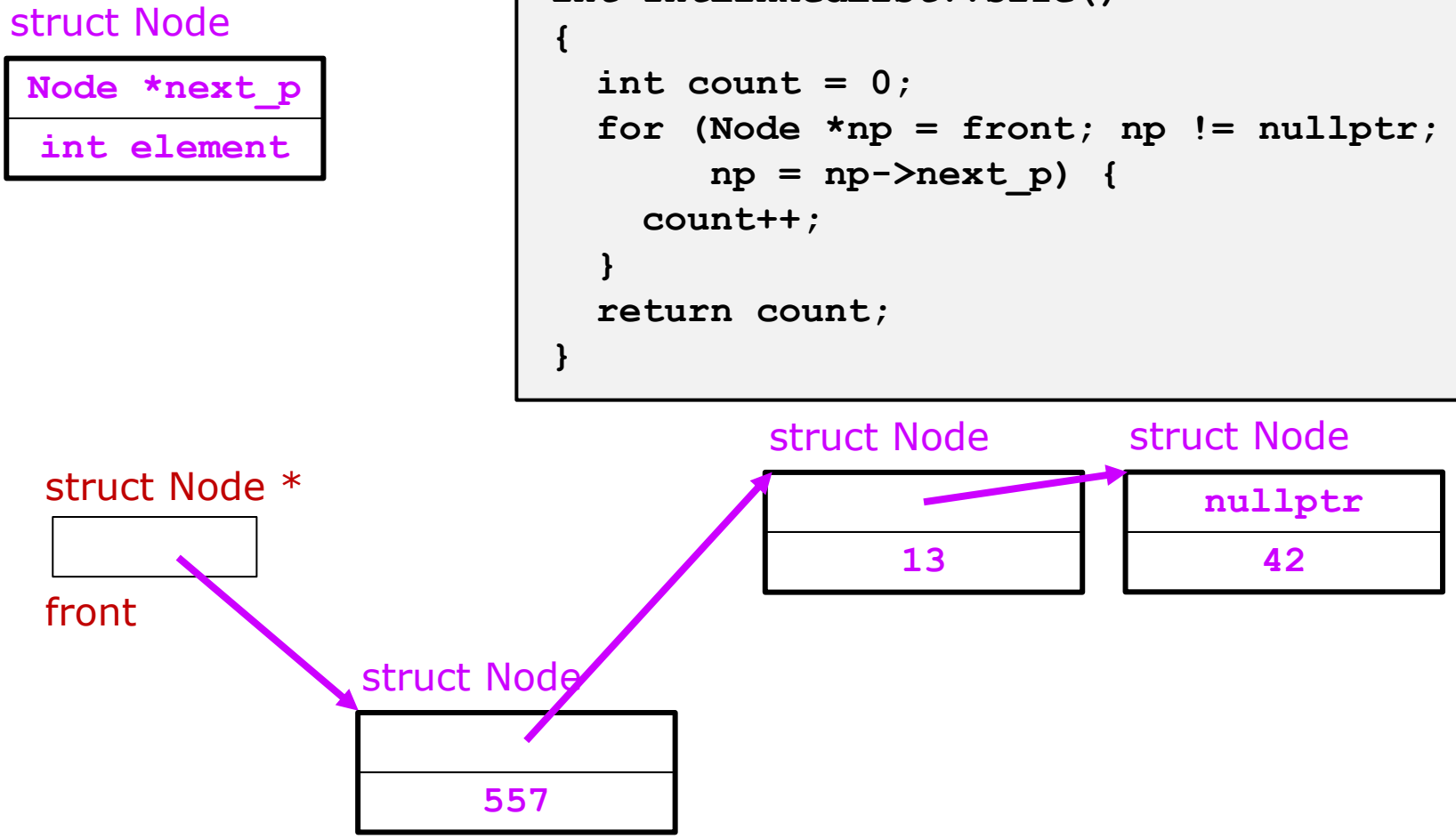
The interface to our IntLinkedList class

```
 IntLinkedList();
~IntLinkedList();

bool isEmpty();
int  size();

void addToFront(int n);
void removeFirst();

int  getElement(int index);
void setElement(int index, int newValue);

void addToBack(int n);
void addAtPosition(int n, int position);

void print();
void printAnnotated();
```

Count the elements

somelist.size()

```
int IntLinkedList::size()
{
  int count = 0;
  for (Node *np = front; np != nullptr;
       np = np->next_p) {
    count++;
  }
  return count;
}
```

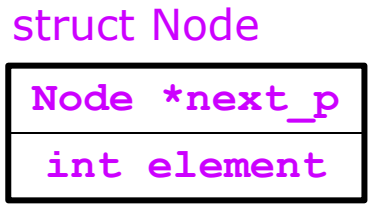


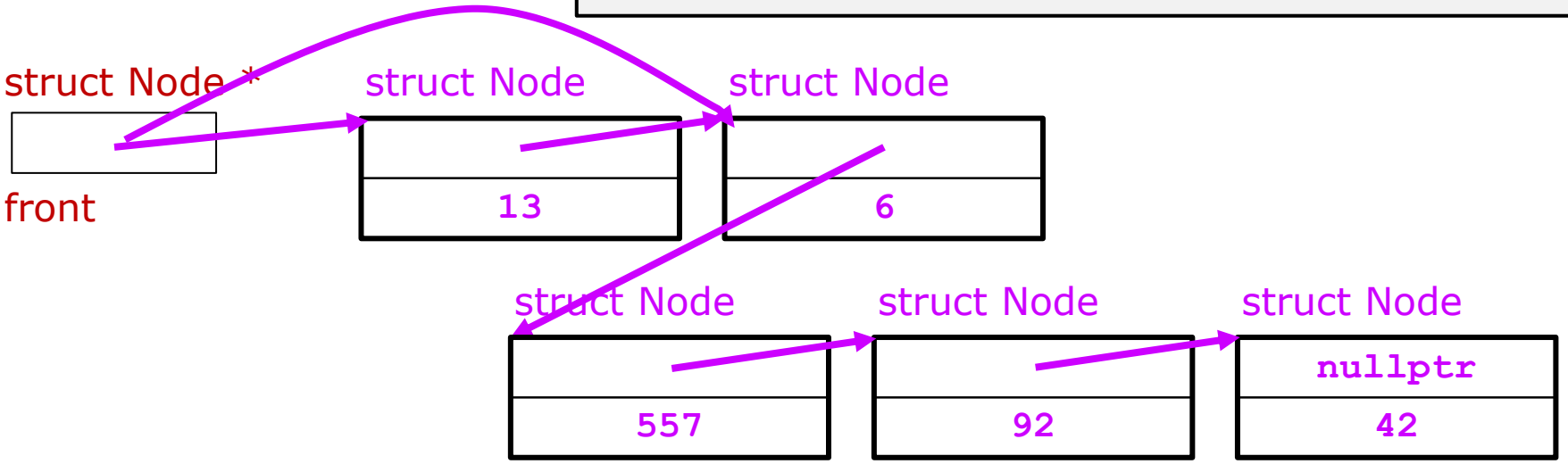
The interface to our IntLinkedList class

```
 IntLinkedList();
~IntLinkedList();

bool isEmpty();
int  size();

void addToFront(int n);
void removeFirst();

int  getElement(int index);
void setElement(int index, int newValue);

void addToBack(int n);
void addAtPosition(int n, int position);

void print();
void printAnnotated();
```

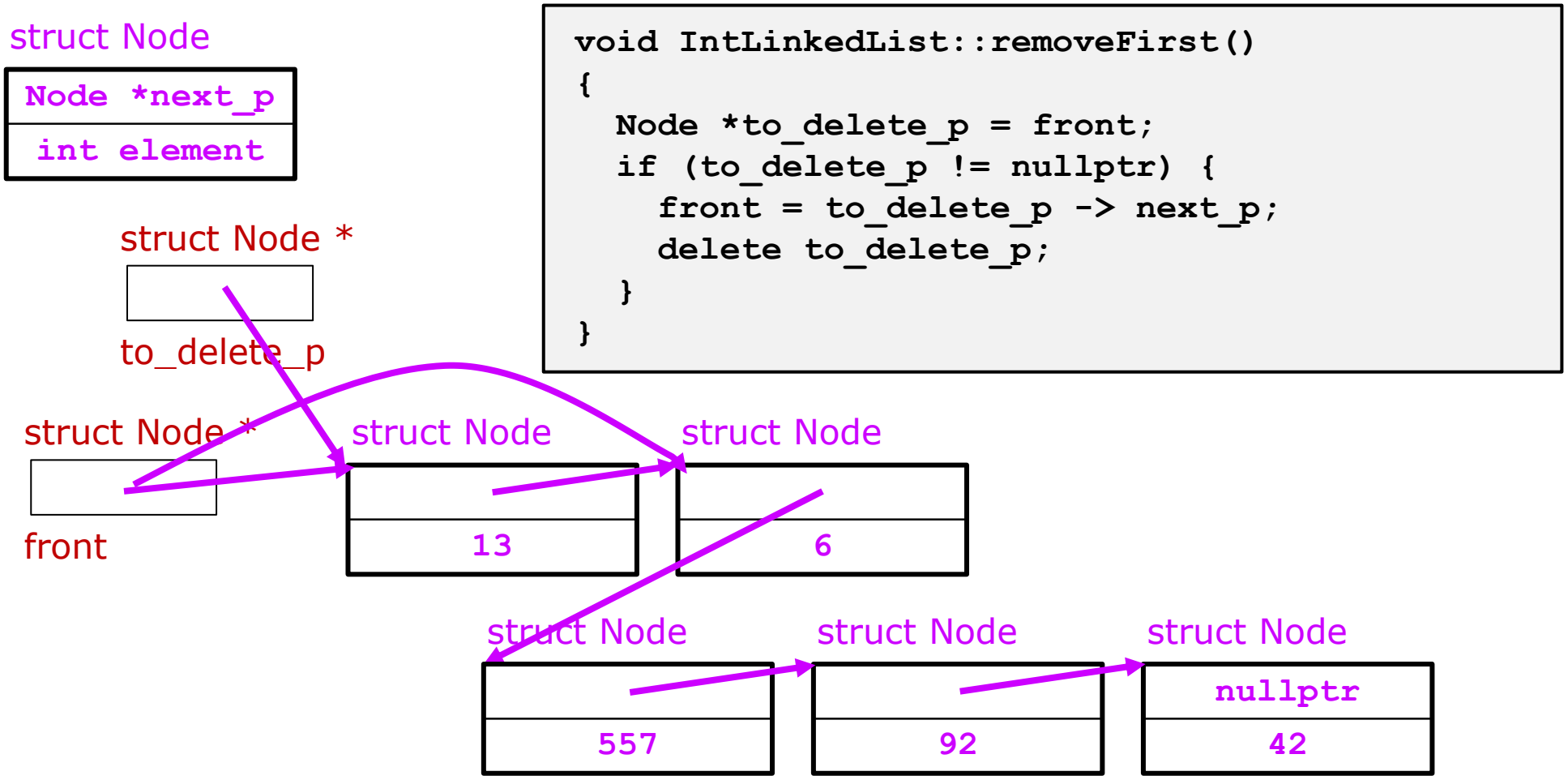
Removing the first element

```
void IntLinkedList::removeFirst()
{
  Node *to_delete_p = front;
  if (to_delete_p != nullptr) {
    front = to_delete_p -> next_p;
    delete to_delete_p;
  }
}
```





Removing the first element

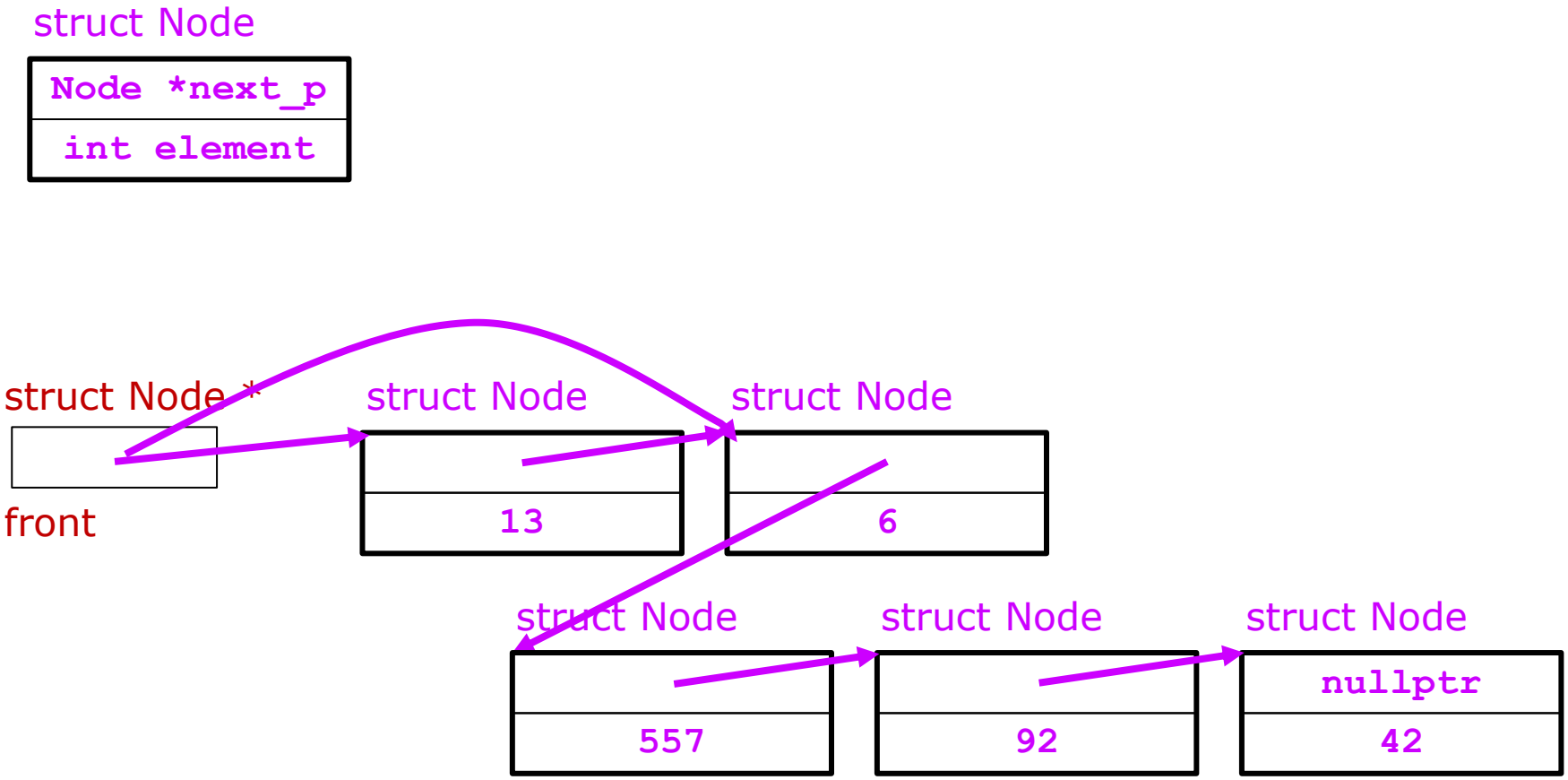



```
void IntLinkedList::removeFirst()
{
  Node *to_delete_p = front;
  if (to_delete_p != nullptr) {
    front = to_delete_p -> next_p;
    delete to_delete_p;
  }
}
```

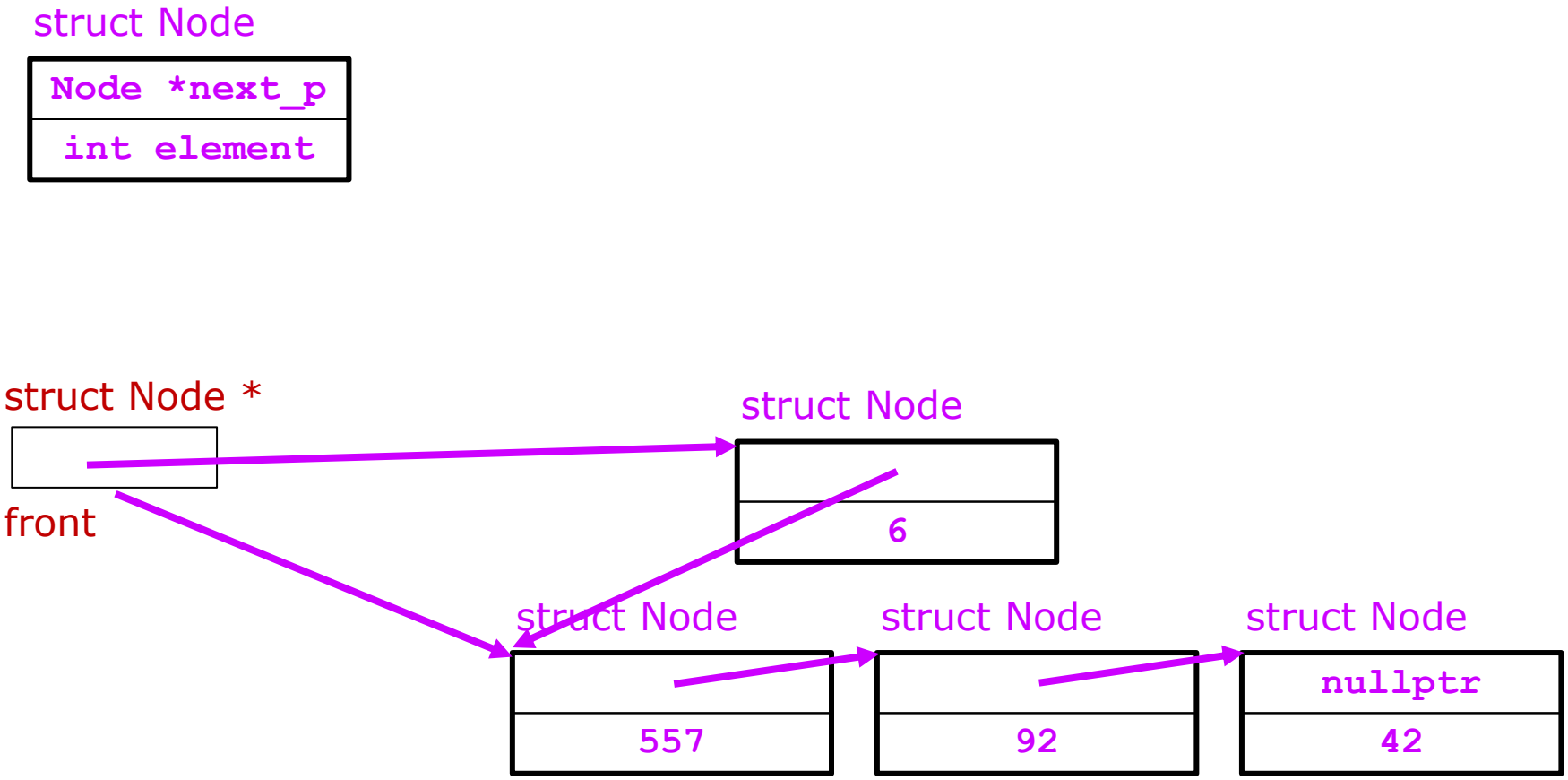
The interface to our IntLinkedList class

```
  IntLinkedList();
 ~IntLinkedList();

bool isEmpty();
int  size();

void addToFront(int n);
void removeFirst();

int  getElement(int index);
void setElement(int index, int newValue);

void addToBack(int n);
void addAtPosition(int n, int position);

void print();
void printAnnotated();
```

Destructor: remove all the elements



Destructor: remove all the elements

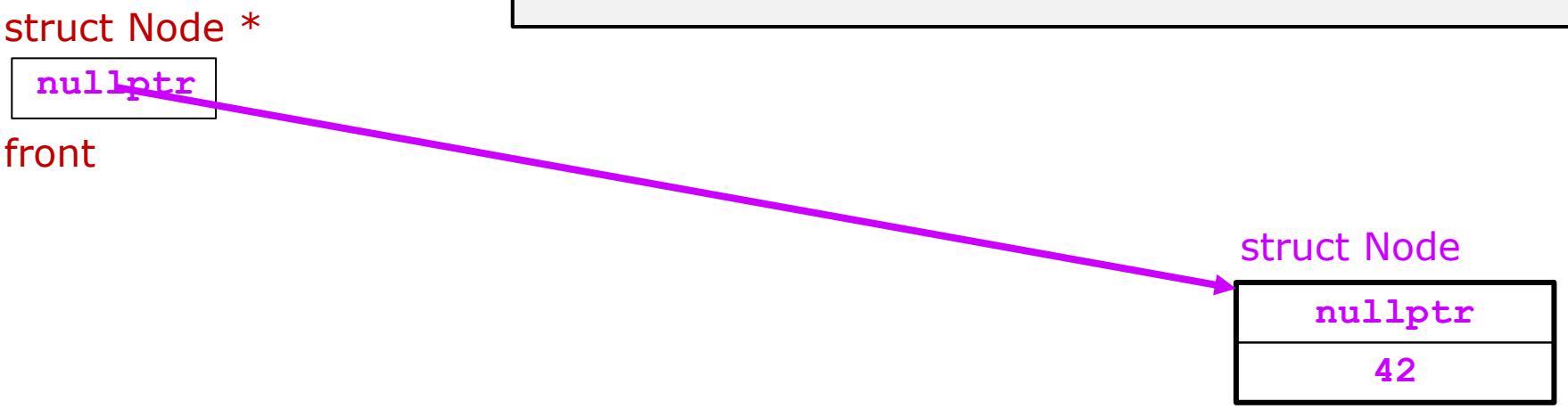


Destructor: remove all the elements

struct Node

Node *next_p
int element

```
IntLinkedList::~IntLinkedList()
{
  while (front != nullptr) {
    removeFirst();
  }
}
```

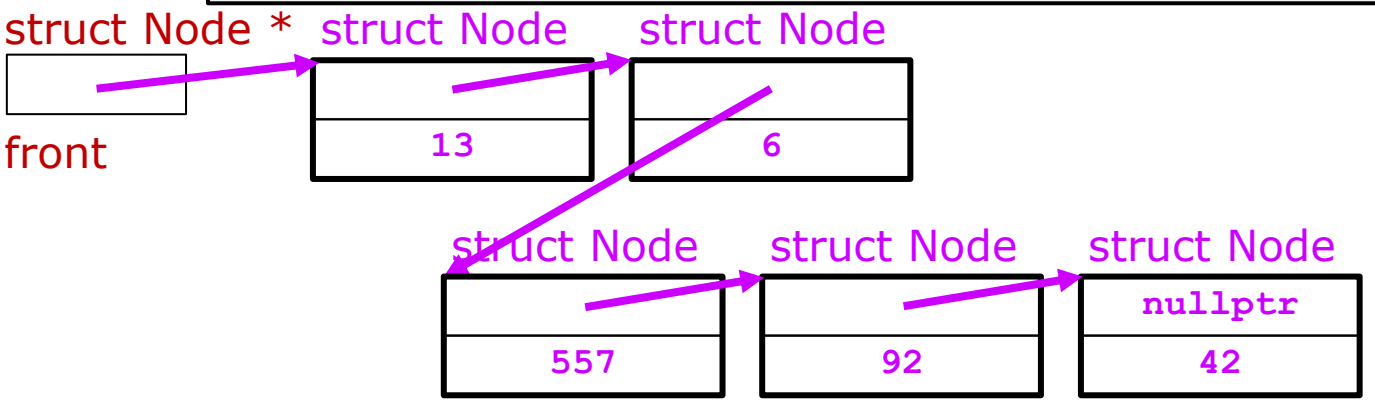


The interface to our IntLinkedList class

```
 IntLinkedList();
~IntLinkedList();

bool isEmpty();
int  size();

void addToFront(int n);
void removeFirst();

int  getElement(int index);
void setElement(int index, int newValue);

void addToBack(int n);
void addAtPosition(int n, int position);

void print();
void printAnnotated();
```

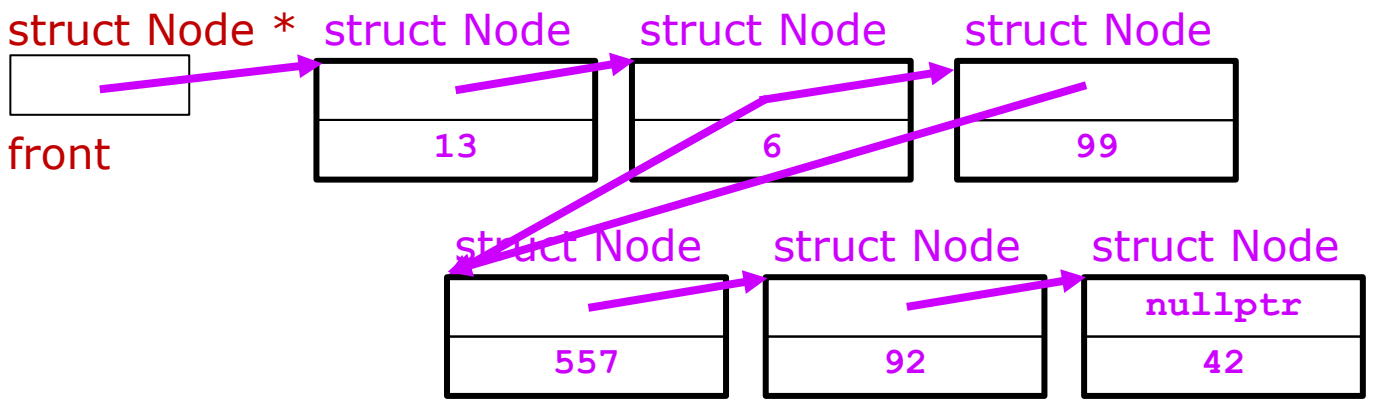
Finding list entry at a given index

```
int  IntLinkedList::getElement(int index)
{
  Node *np;
  int count = 0;
  for (np = front;
       (np != nullptr) and (count < index);
        np = np->next_p) {
                 count++;
  }
  // Here, np points to the node we want
  if (np == nullptr) {
    cerr << "index out of range" << endl; return -1;
  }
  return np->element;
}
```



Adding at a given position

somelist.addAtPosition(99, 2)



The interface to our IntLinkedList class

```
 IntLinkedList();
~IntLinkedList();

bool isEmpty();
int  size();

void addToFront(int n);
void removeFirst();

int  getElement(int index);
void setElement(int index, int newValue);

void addToBack(int n);
void addAtPosition(int n, int position);

void print();
void printAnnotated();
```

Adding at a given position

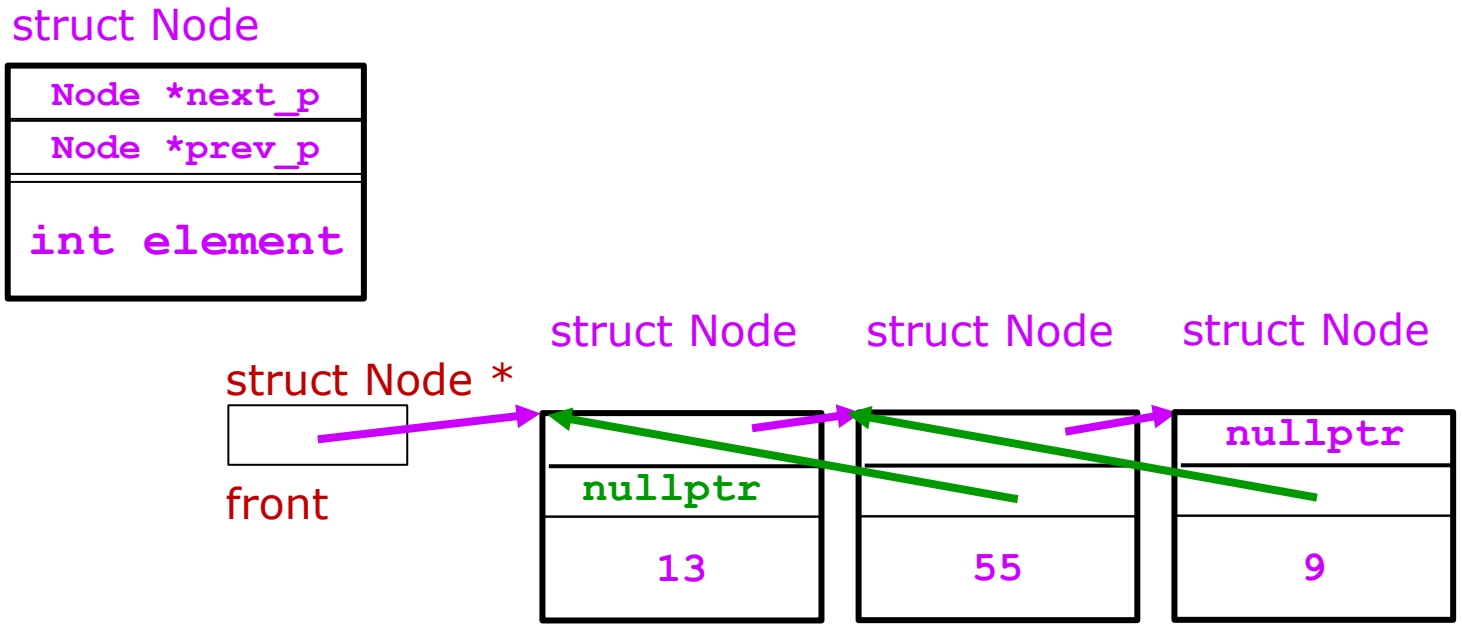
```
void IntLinkedList::addAtPosition(int n, int position)
{
        if (position == 0) {
                addToFront(n);
        } else {
                Node *prev;
                int count = 0;
                for (prev = front;
                     (prev != nullptr) and (count < (position-1));
                     prev = prev->next_p) {
                     count++;
                }
                if (prev == nullptr) {
                        cerr << "index out of range" << endl;
                } else {
                        Node *newNode_p = new Node{prev->next_p, n};
                        prev->next_p = newNode_p;
                }
        }
        return;
}
```

Implementing lists using linked nodes

- **Adding at the end is:**
  - Easy?
  - Hard?
- **Adding at the beginning is:**
  - Easy?
  - Hard?
- **Getting a value by index is:**
  - Easy?
  - Hard?
- **Deleting at the end is:**
  - Easy?
  - Hard?
- **Deleting at the beginning is:**
  - Easy?
  - Hard?
- **Inserting / deleting in the middle is:**
  - Easy?
  - Hard?

Just For Fun
Fancier Lists

Doubly linked lists: forward and backward pointers



Summary