Linked Lists

Part Two

Recursion is Awesome

http://recursivedrawing.com/

Friday Four Square!
4:15PM, Outside Gates

Apply to Section Lead!

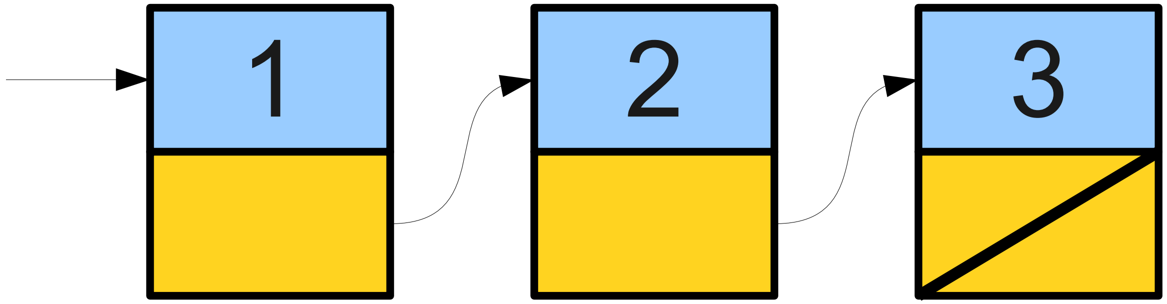
http://cs198.stanford.edu

Announcements

- Assignment 4 due right now.
- Assignment 5 (**Priority Queue**) out today, due Wednesday, May 23
    - Implement a powerful collection class.
    - Master dynamic allocation and linked lists.
    - YEAH hours next Tuesday from 4:15 – 5:45 in 380-380C.

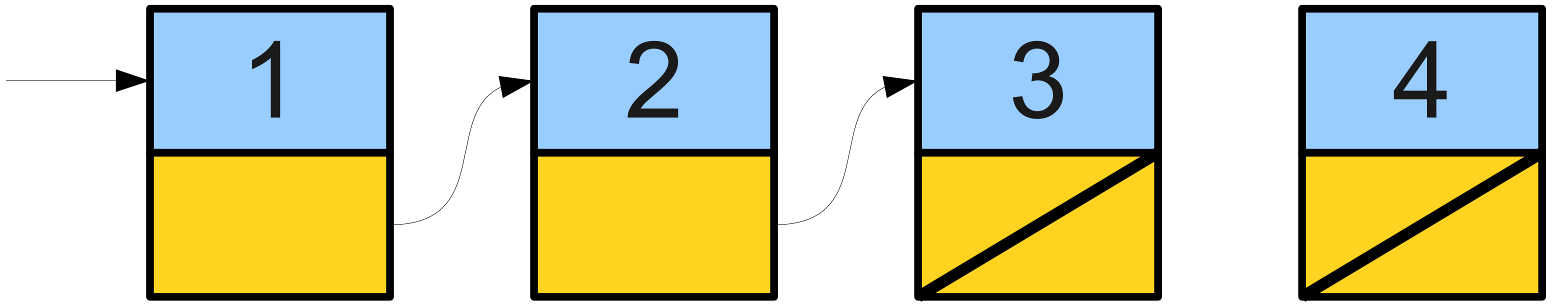
Linked Lists at a Glance

- A **linked list** is a data structure for storing a sequence of elements.
- Each element is stored separately from the rest.
- The elements are then chained together into a sequence.



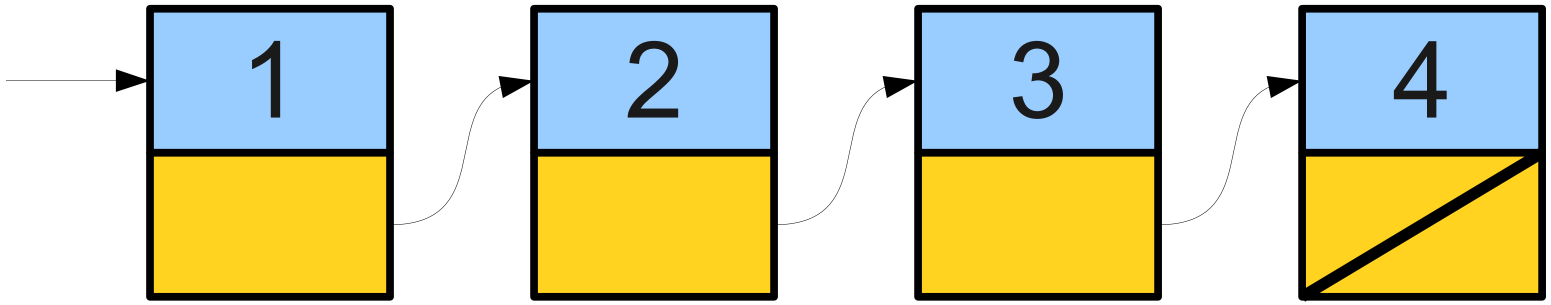
Linked Lists at a Glance

- A **linked list** is a data structure for storing a sequence of elements.
- Each element is stored separately from the rest.
- The elements are then chained together into a sequence.



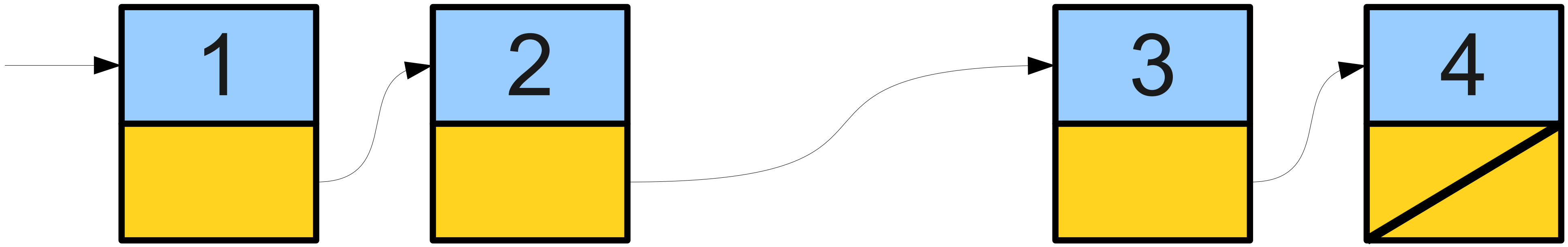
Linked Lists at a Glance

- A **linked list** is a data structure for storing a sequence of elements.
- Each element is stored separately from the rest.
- The elements are then chained together into a sequence.



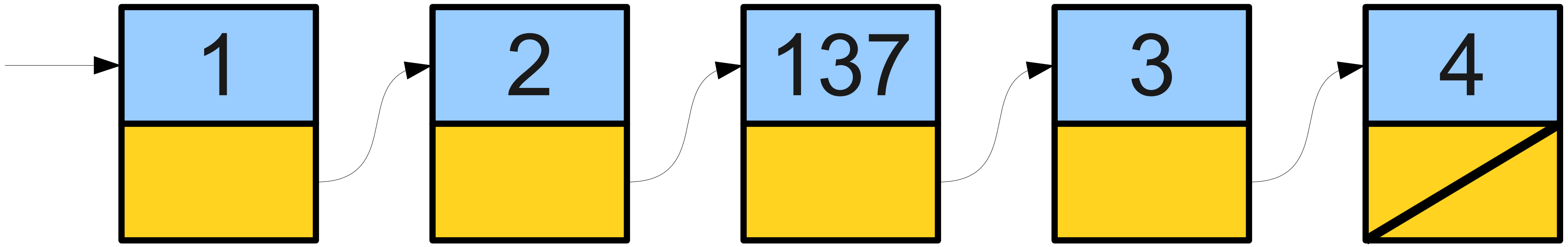
Linked Lists at a Glance

- A **linked list** is a data structure for storing a sequence of elements.
- Each element is stored separately from the rest.
- The elements are then chained together into a sequence.



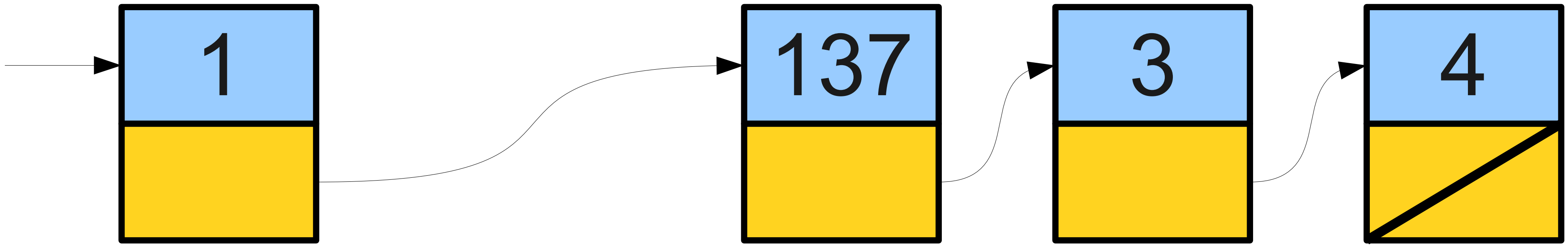
Linked Lists at a Glance

- A **linked list** is a data structure for storing a sequence of elements.
- Each element is stored separately from the rest.
- The elements are then chained together into a sequence.



Linked Lists at a Glance

- A **linked list** is a data structure for storing a sequence of elements.
- Each element is stored separately from the rest.
- The elements are then chained together into a sequence.



Linked List Cells

- A linked list is a chain of **cells**.
- Each cell contains two pieces of information:
    - Some piece of data that is stored in the sequence, and
    - A **link** to the next cell in the list.
- We can traverse the list by starting at the first cell and repeatedly following its link.

Representing a Cell

- For simplicity, let's assume we're building a linked list of **`string`**s.
- We can represent a cell in the linked list as a structure:

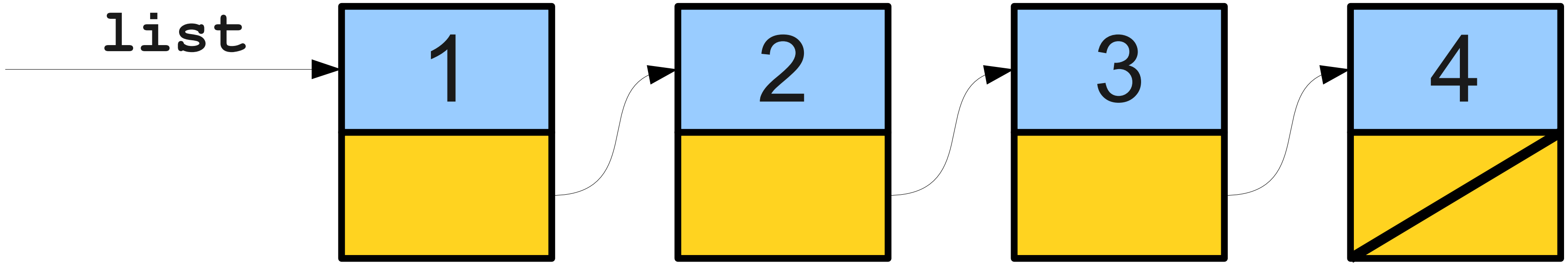
```
struct Cell {
    string value;
    Cell* next;
};
```

- **The structure is defined recursively!**

Traversing a Linked List

- Once we have a linked list, we can traverse it by following the links one at a time.

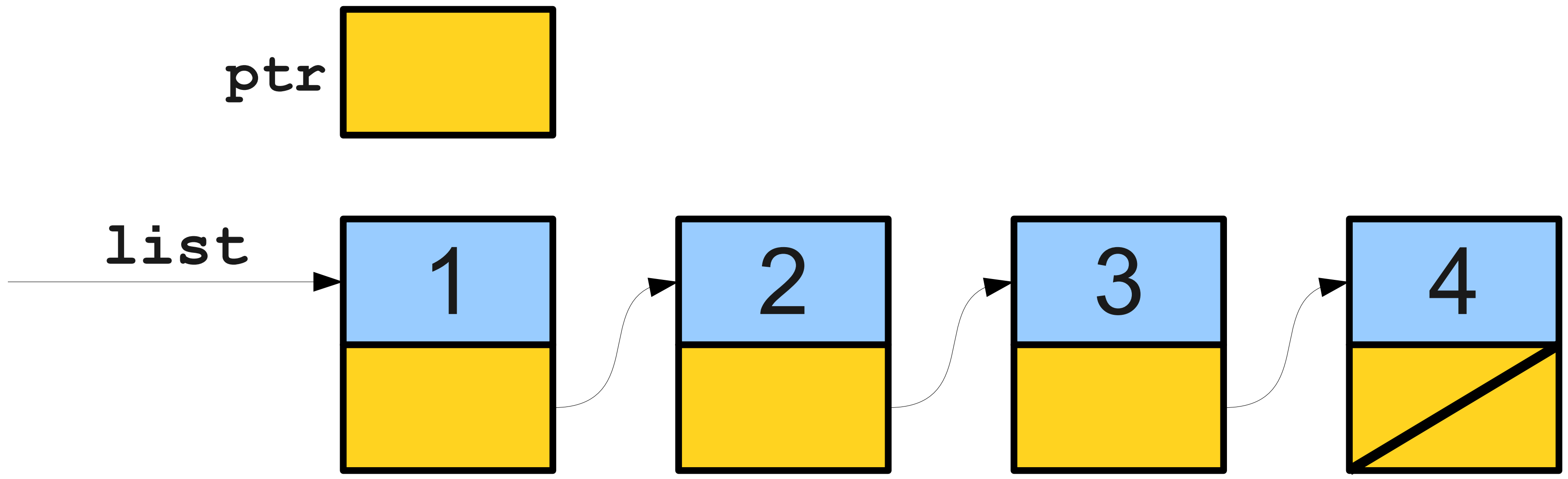
```
for (Cell* ptr = list; ptr != NULL; ptr = ptr->next) {
    /* … use ptr … */
}
```



Traversing a Linked List

- Once we have a linked list, we can traverse it by following the links one at a time.

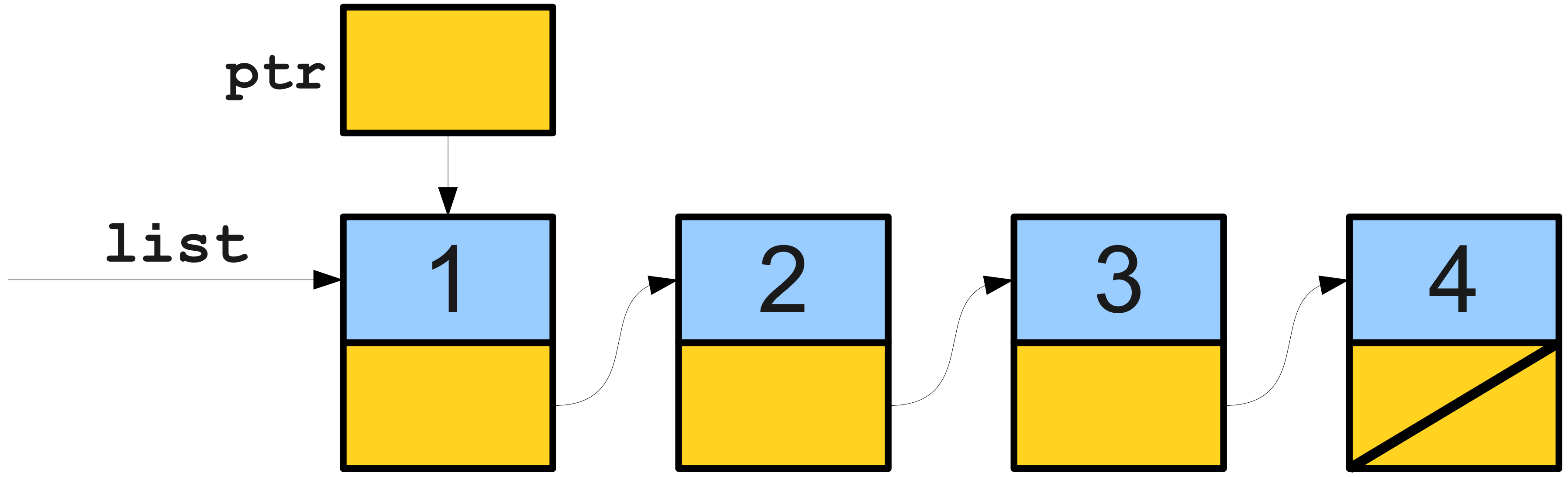
```
for (Cell* ptr = list; ptr != NULL; ptr = ptr->next) {
    /* … use ptr … */
}
```



Traversing a Linked List

- Once we have a linked list, we can traverse it by following the links one at a time.

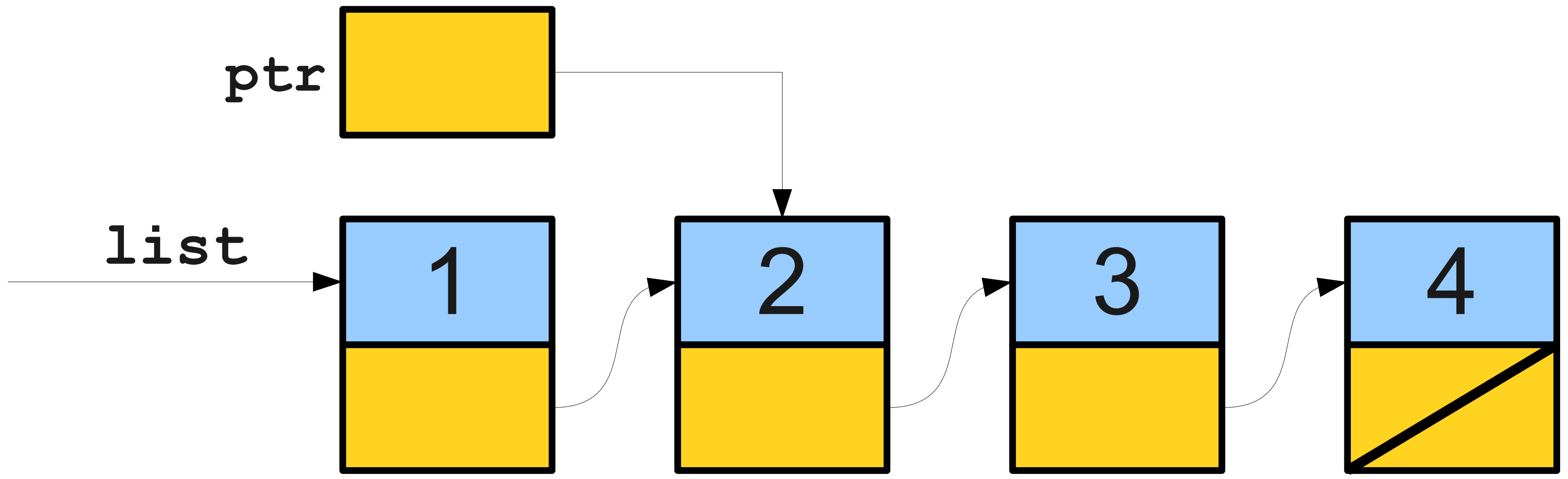
```
for (Cell* ptr = list; ptr != NULL; ptr = ptr->next) {
    /* … use ptr … */
}
```



Traversing a Linked List

- Once we have a linked list, we can traverse it by following the links one at a time.

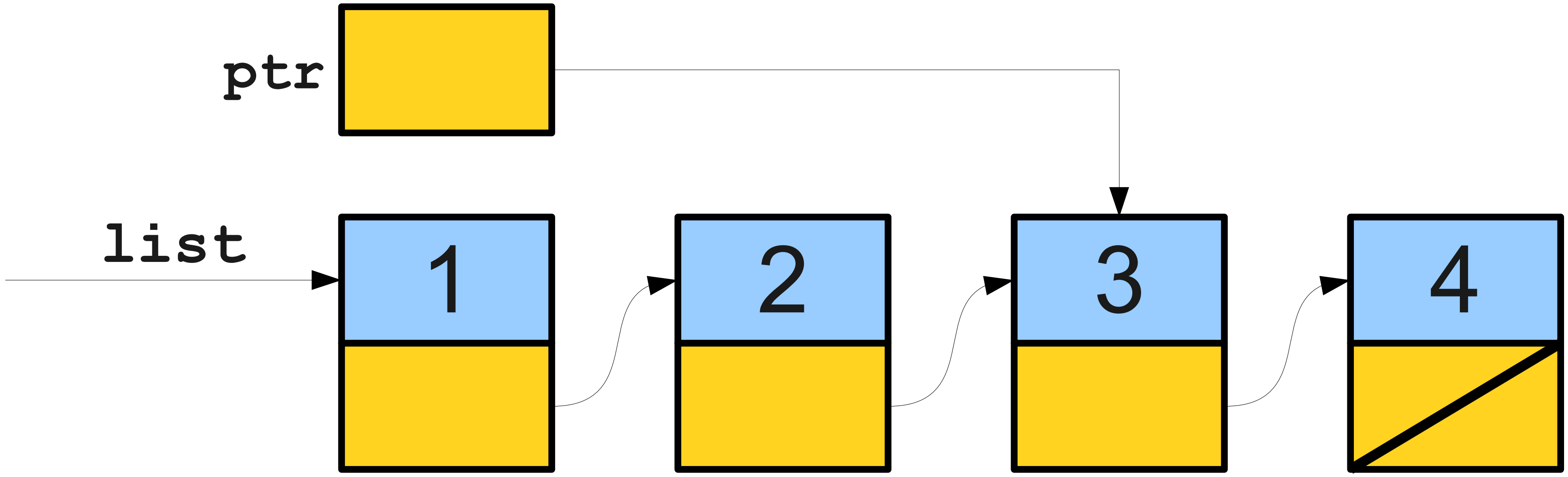
```
for (Cell* ptr = list; ptr != NULL; ptr = ptr->next) {
    /* … use ptr … */
}
```



Traversing a Linked List

- Once we have a linked list, we can traverse it by following the links one at a time.

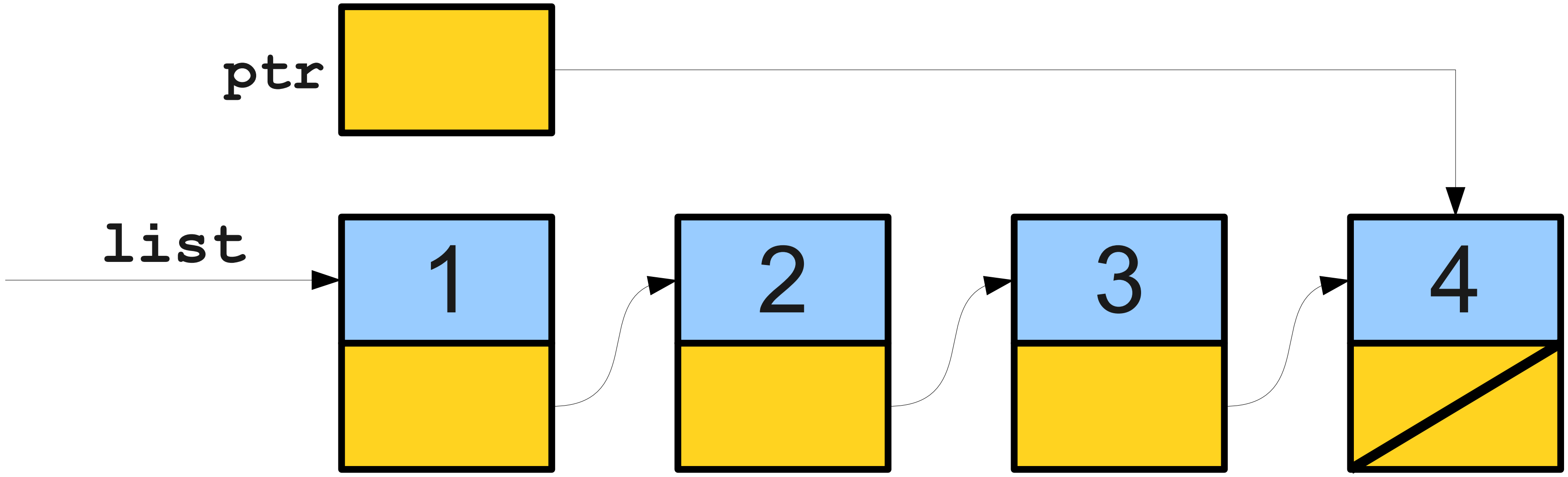
```
for (Cell* ptr = list; ptr != NULL; ptr = ptr->next) {
    /* … use ptr … */
}
```



Traversing a Linked List

- Once we have a linked list, we can traverse it by following the links one at a time.

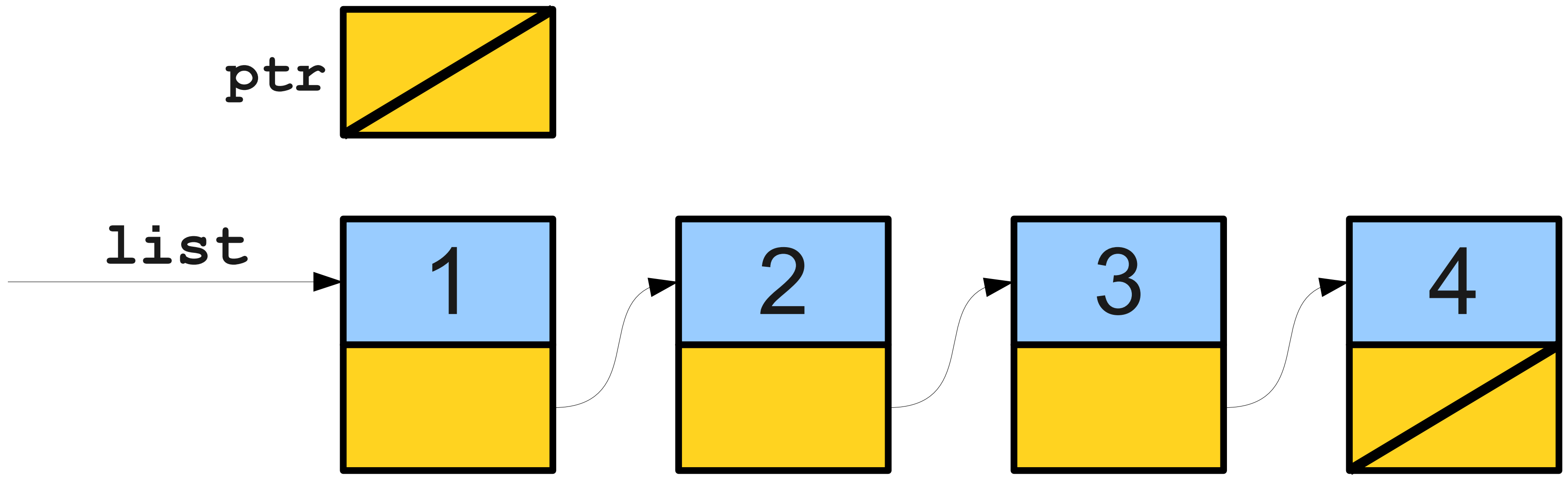
```
for (Cell* ptr = list; ptr != NULL; ptr = ptr->next) {
    /* … use ptr … */
}
```



Traversing a Linked List

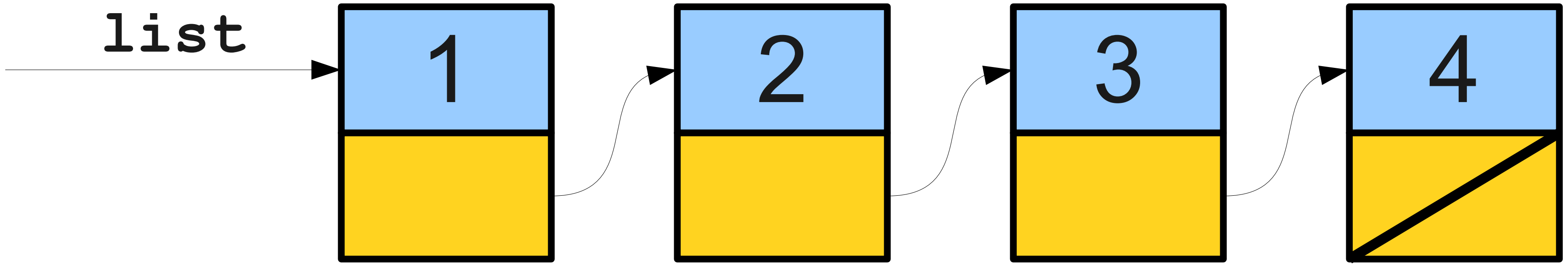
- Once we have a linked list, we can traverse it by following the links one at a time.

```
for (Cell* ptr = list; ptr != NULL; ptr = ptr->next) {
    /* … use ptr … */
}
```



A Recursive View of Linked Lists

- We can think of linked lists **recursively**.
- The empty list of no cells (represented by a **`NULL`** pointer) is a linked list.
- A linked list cell followed by a linked list is a linked list.



Once More With Recursion

- Linked lists are defined recursively, and we can traverse them using recursion!

```
void recursiveTraverse(Cell* list) {
    if (list == NULL) return;
    /* … do something with list … */
    recursiveTraverse(list->next);
}
```

Freeing a Linked List

- All good things must come to an end, and we eventually need to reclaim the memory for a linked list.
- The following is an Extremely Bad Idea:

```
for (Cell* ptr = list; ptr != NULL; ptr = ptr->next) {
    delete ptr;
}
```

Freeing a Linked List

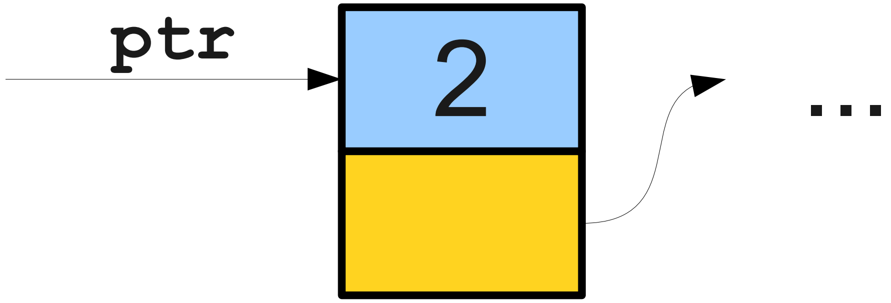
- All good things must come to an end, and we eventually need to reclaim the memory for a linked list.
- The following is an Extremely Bad Idea:

```
for (Cell* ptr = list; ptr != NULL; ptr = ptr->next) {
    delete ptr;
}
```

Freeing a Linked List

- All good things must come to an end, and we eventually need to reclaim the memory for a linked list.
- The following is an Extremely Bad Idea:

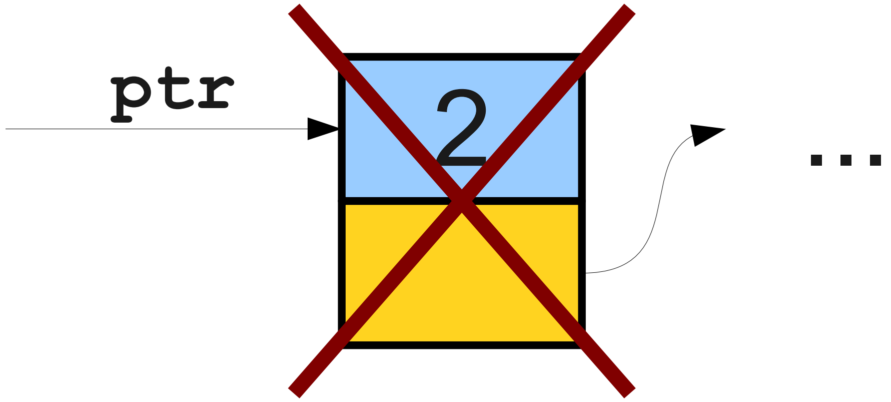
```
for (Cell* ptr = list; ptr != NULL; ptr = ptr->next) {
    delete ptr;
}
```



Freeing a Linked List

- All good things must come to an end, and we eventually need to reclaim the memory for a linked list.
- The following is an Extremely Bad Idea:

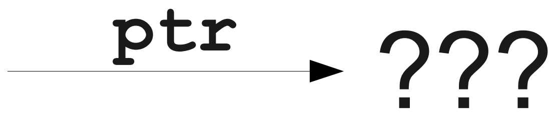
```
for (Cell* ptr = list; ptr != NULL; ptr = ptr->next) {
    delete ptr;
}
```



Freeing a Linked List

- All good things must come to an end, and we eventually need to reclaim the memory for a linked list.
- The following is an Extremely Bad Idea:

```
for (Cell* ptr = list; ptr != NULL; ptr = ptr->next) {
    delete ptr;
}
```



Freeing a Linked List Properly

- To properly free a linked list, we have to be able to
  - Destroy a cell, and
  - Advance to the cell after it.
- How might we accomplish this?

Once More With Recursion

- We can also deallocate lists recursively!
- **Base Case:**
  - There is nothing to free in an empty list.
- **Recursive Case:**
  - Deallocate all cells after the current cell.
  - Deallocate the current cell.

Linked Lists: The Tricky Parts

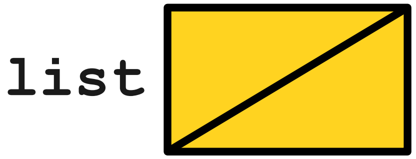
- Suppose that we want to write a function that will add an element to a linked list.
- What might this function look like?

What went wrong?

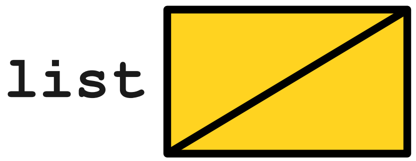
```
int main() {
    Cell* list = NULL;
    ListInsert(list, 137);
    ListInsert(list, 42);
    ListInsert(list, 271);
}
```

```
int main() {
    Cell* list = NULL;
    ListInsert(list, 137);
    ListInsert(list, 42);
    ListInsert(list, 271);
}
```

```
int main() {
    Cell* list = NULL;
    ListInsert(list, 137);
    ListInsert(list, 42);
    ListInsert(list, 271);
}
```

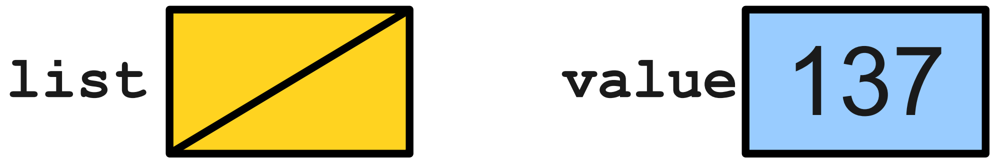


```
int main() {
    Cell* list = NULL;
    ListInsert(list, 137);
    ListInsert(list, 42);
    ListInsert(list, 271);
}
```



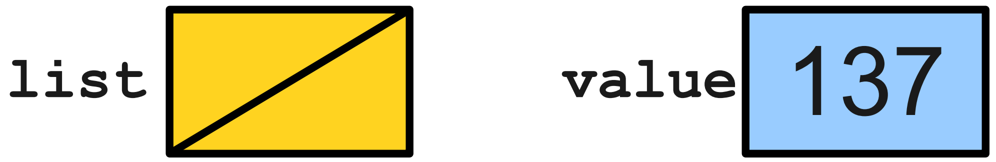
```
int main() {

}
```

```
void listInsert(Cell* list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```



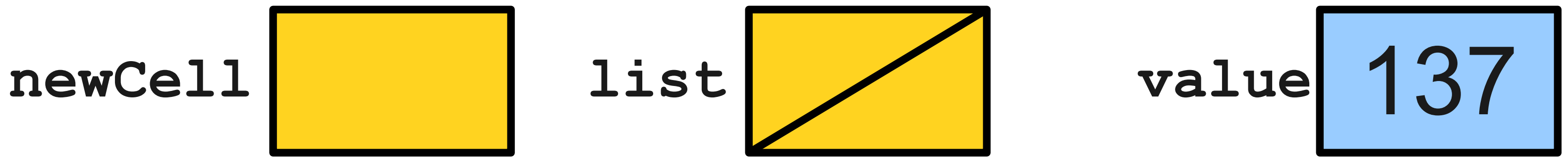
```
int main() {

}
```

```
void listInsert(Cell* list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```



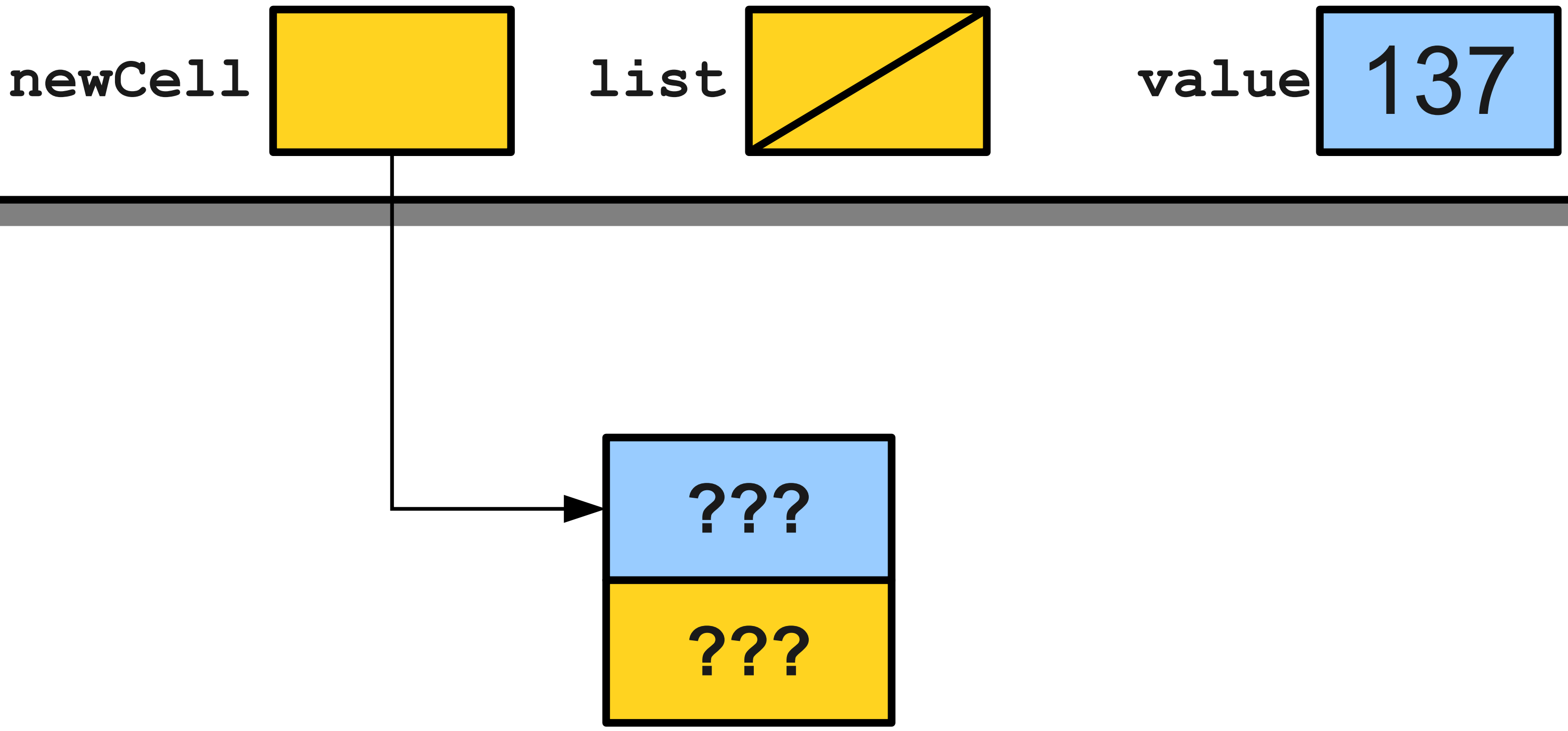
```
int main() {

}
```

```
void listInsert(Cell* list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```



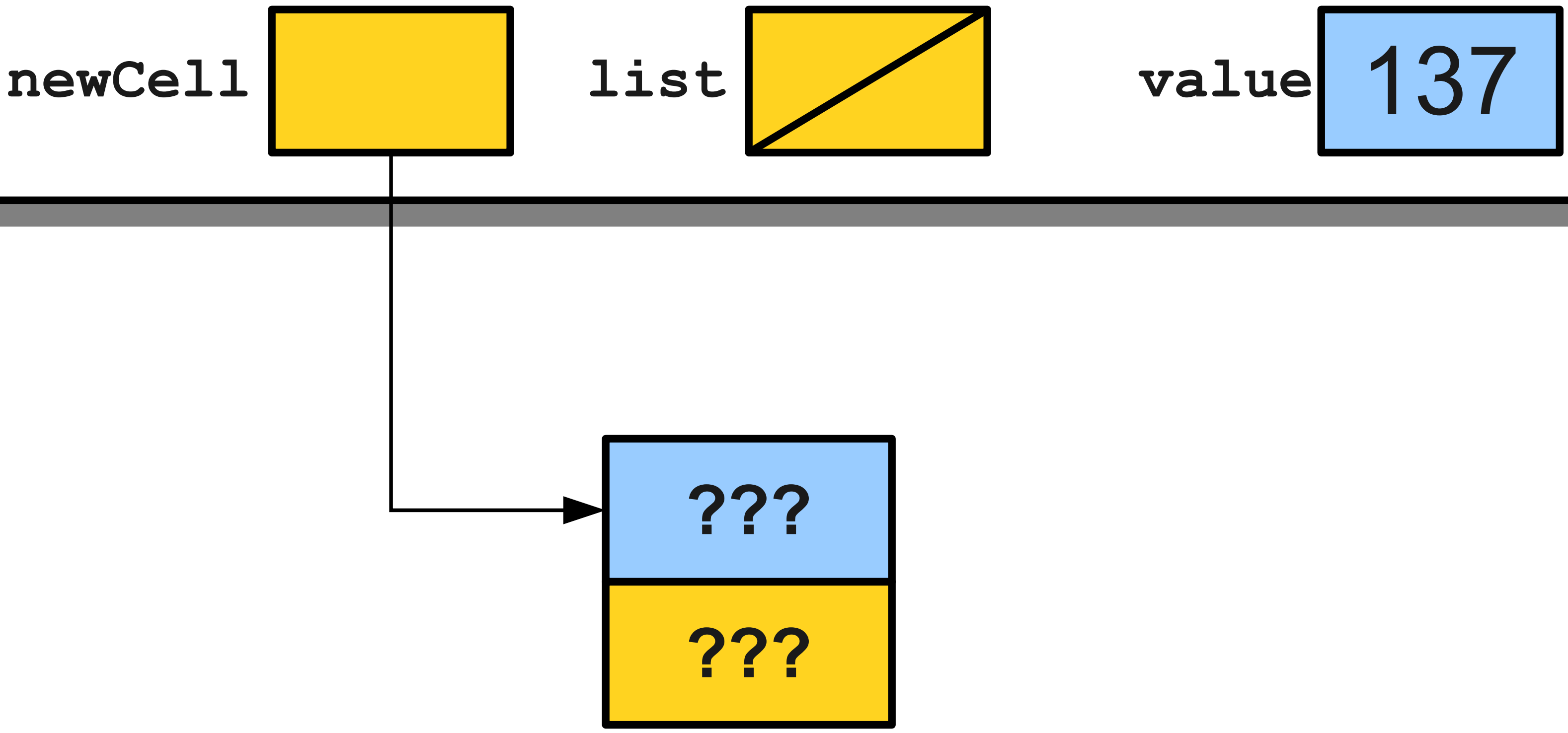
```
int main() {

}
```

```
void listInsert(Cell* list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```



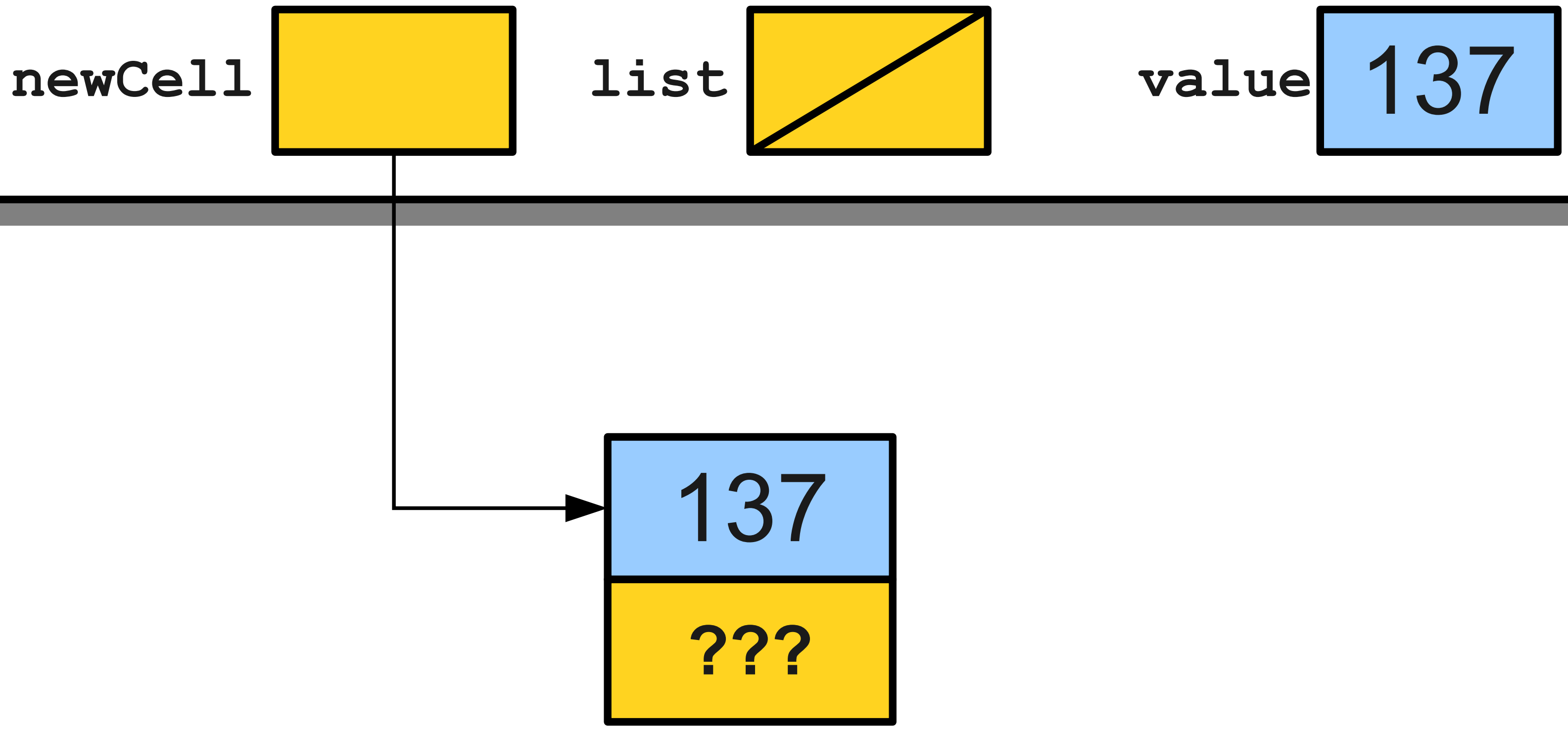
```
int main() {

}
```

```
void listInsert(Cell* list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```



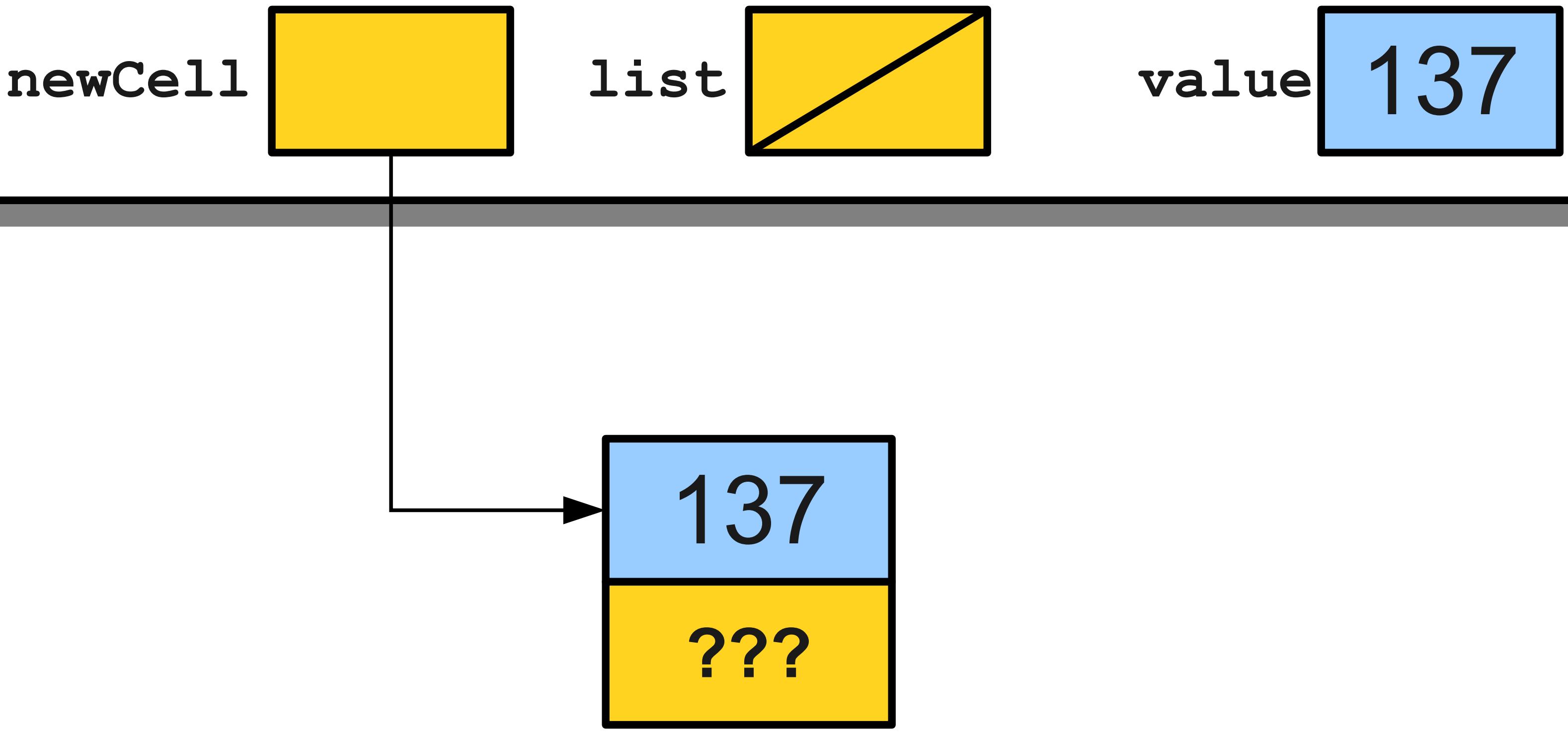
```
int main() {


}
```

```
void listInsert(Cell* list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```



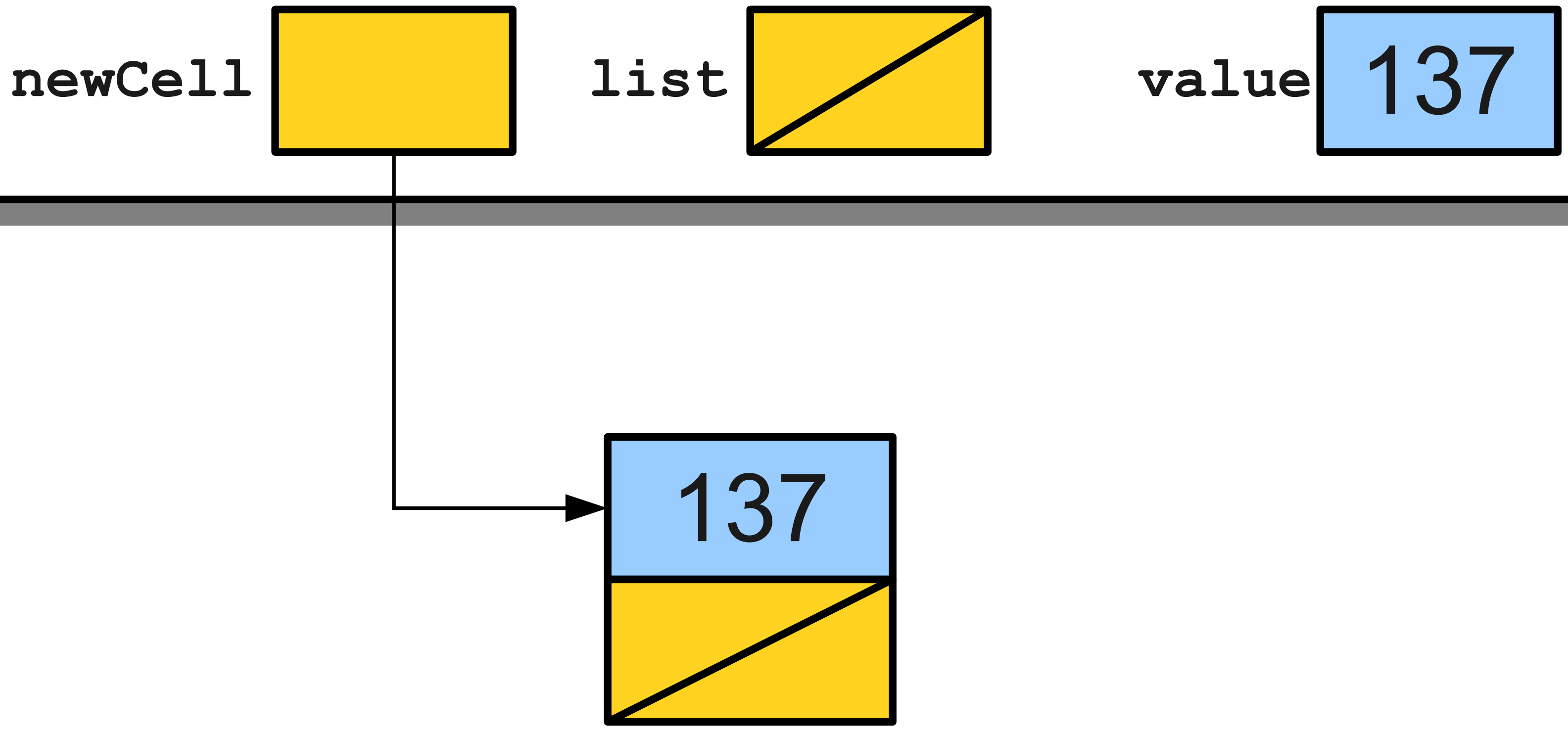
```
int main() {



}
```

```
void listInsert(Cell* list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```



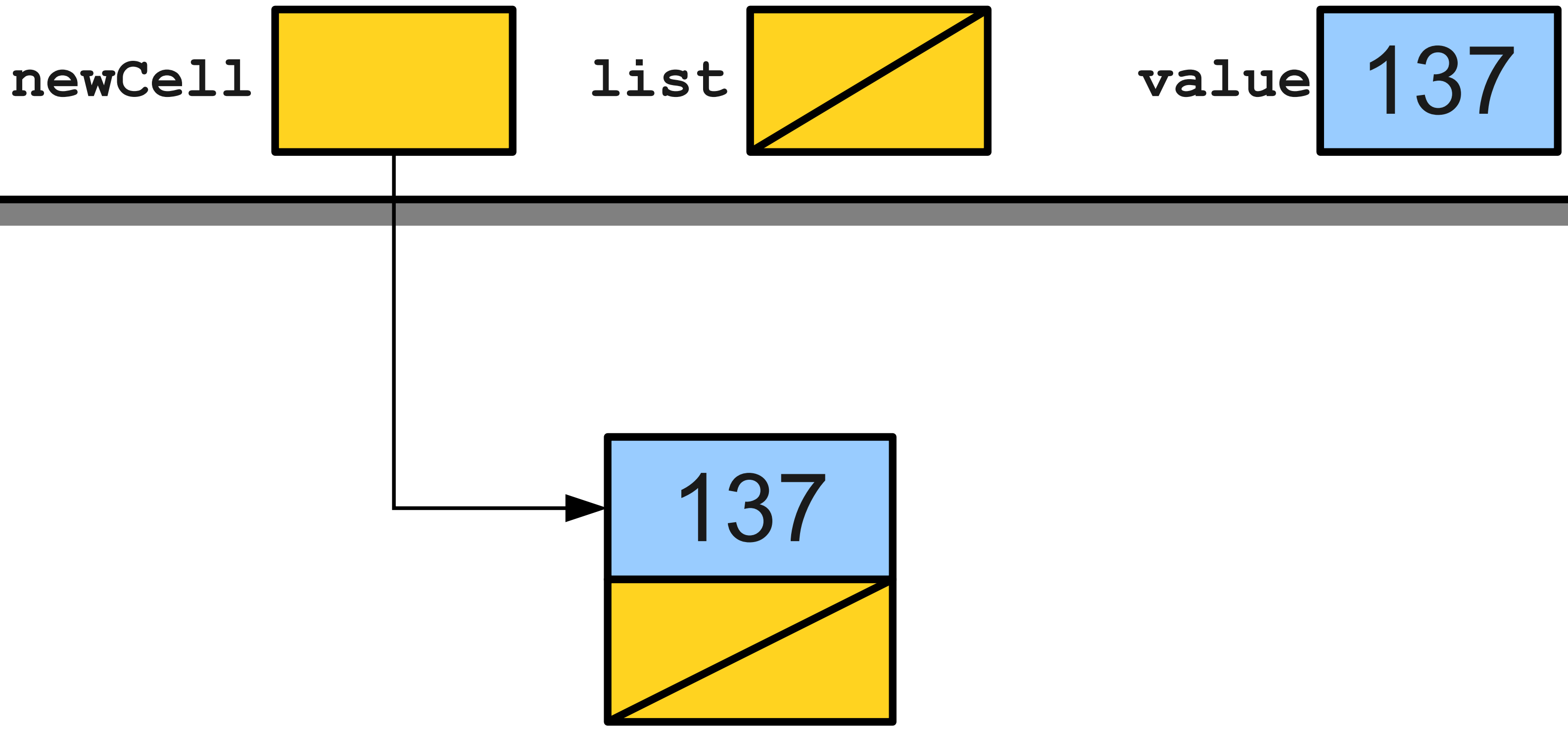
```
int main() {
}
```

```
void listInsert(Cell* list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```



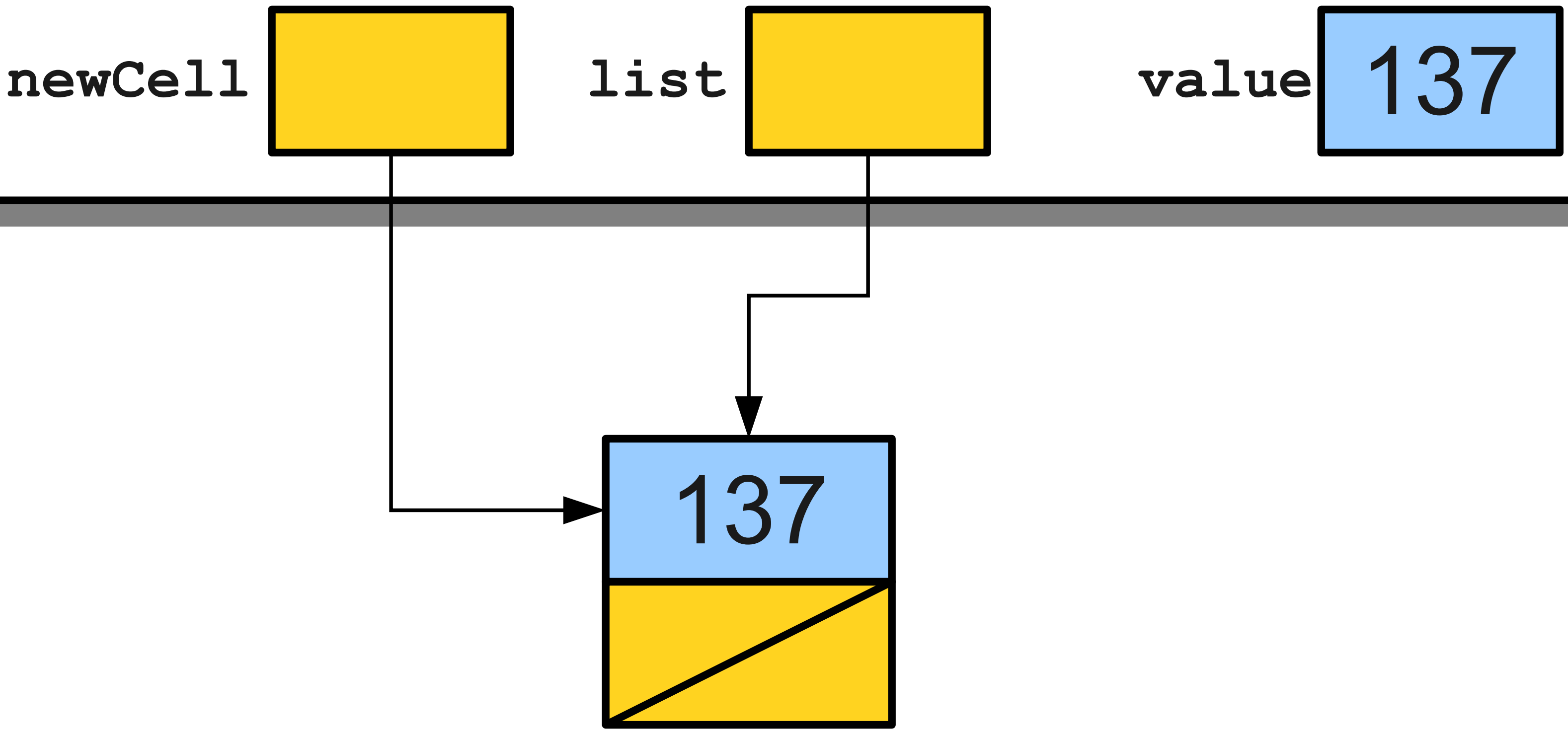
```
int main() {
}
```

```
void listInsert(Cell* list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```

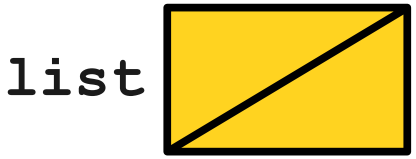


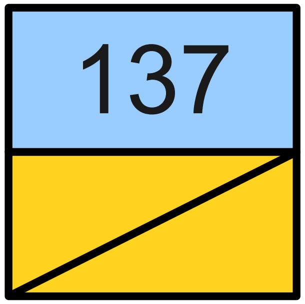
```
int main() {



}
```

```
void listInsert(Cell* list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```



```
int main() {
    Cell* list = NULL;
    ListInsert(list, 137);
    ListInsert(list, 42);
    ListInsert(list, 271);
}
```





Pointers by Reference

- In order to resolve this problem, we must pass the linked list pointer by reference.
- Our new function:

```
void listInsert(Cell*& list, int value) {
    Cell* newCell = new Cell;
    cell->value = value;
    cell->next = list;
    list = cell;
}
```

Pointers by Reference

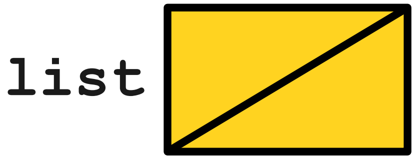
- In order to resolve this problem, we must pass the linked list pointer by reference.
- Our new function:

```
void listInsert(Cell*& list, int value) {
    Cell* newCell = new Cell;
    cell->value = value;
    cell->next = list;
    list = cell;
}
```

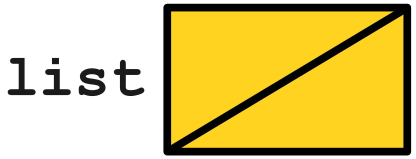
```
int main() {
    Cell* list = NULL;
    ListInsert(list, 137);
    ListInsert(list, 42);
    ListInsert(list, 271);
}
```

```
int main() {
    Cell* list = NULL;
    ListInsert(list, 137);
    ListInsert(list, 42);
    ListInsert(list, 271);
}
```

```
int main() {
    Cell* list = NULL;
    ListInsert(list, 137);
    ListInsert(list, 42);
    ListInsert(list, 271);
}
```

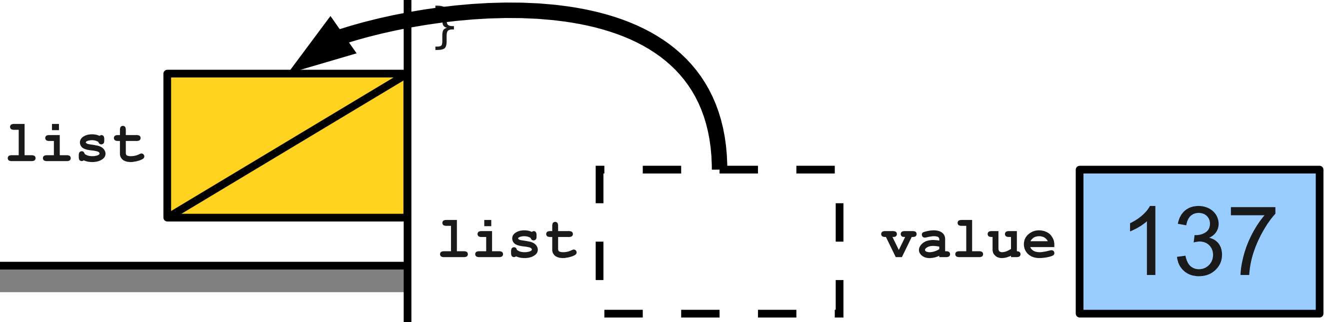


```
int main() {
    Cell* list = NULL;
    ListInsert(list, 137);
    ListInsert(list, 42);
    ListInsert(list, 271);
}
```



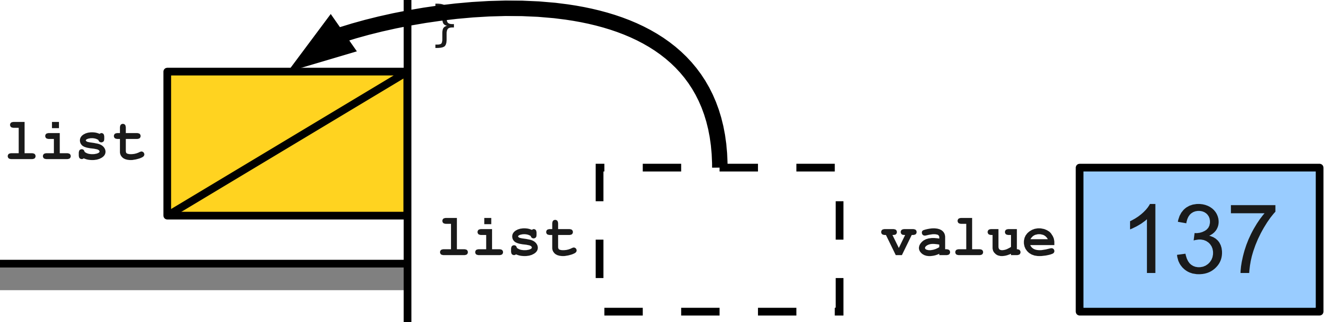
```
int main() {
    Cell* list = N
    ListInsert(lis
    ListInsert(lis
    ListInsert(lis
}
```

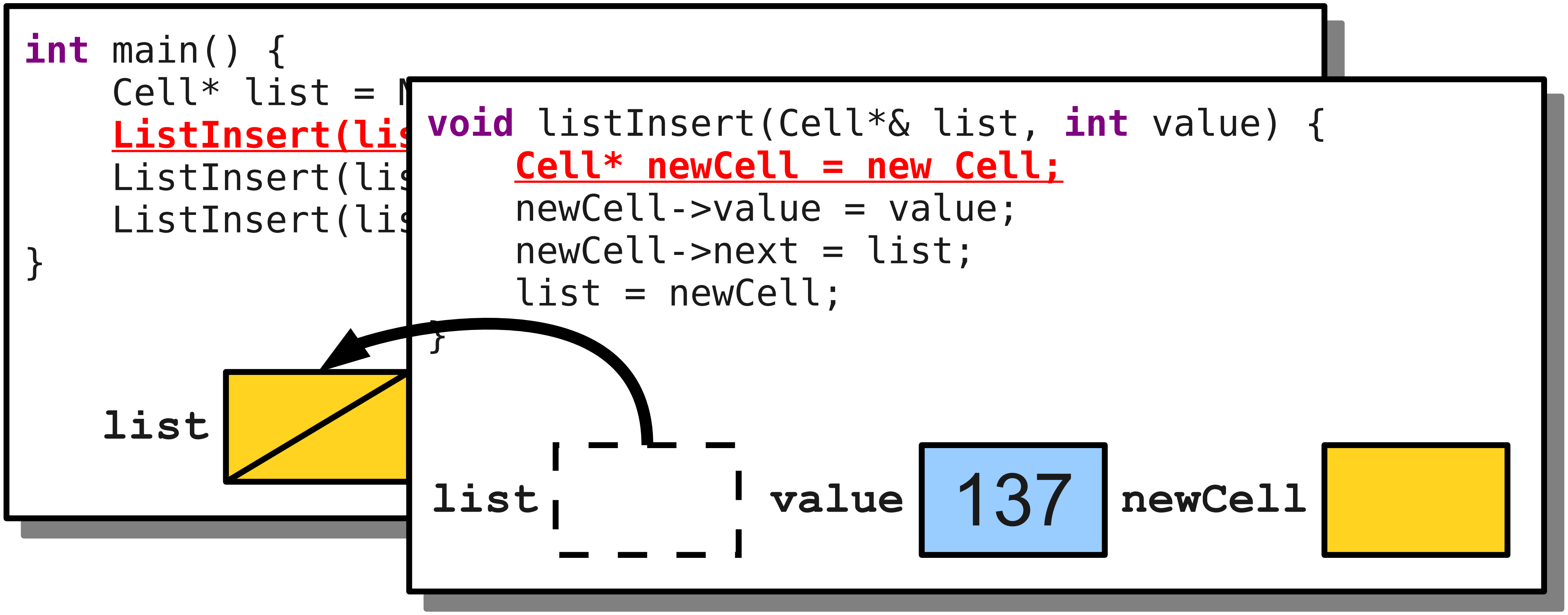
```
void listInsert(Cell*& list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```



```
int main() {
    Cell* list = N
    ListInsert(lis
    ListInsert(lis
    ListInsert(lis
}
```

```
void listInsert(Cell*& list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```



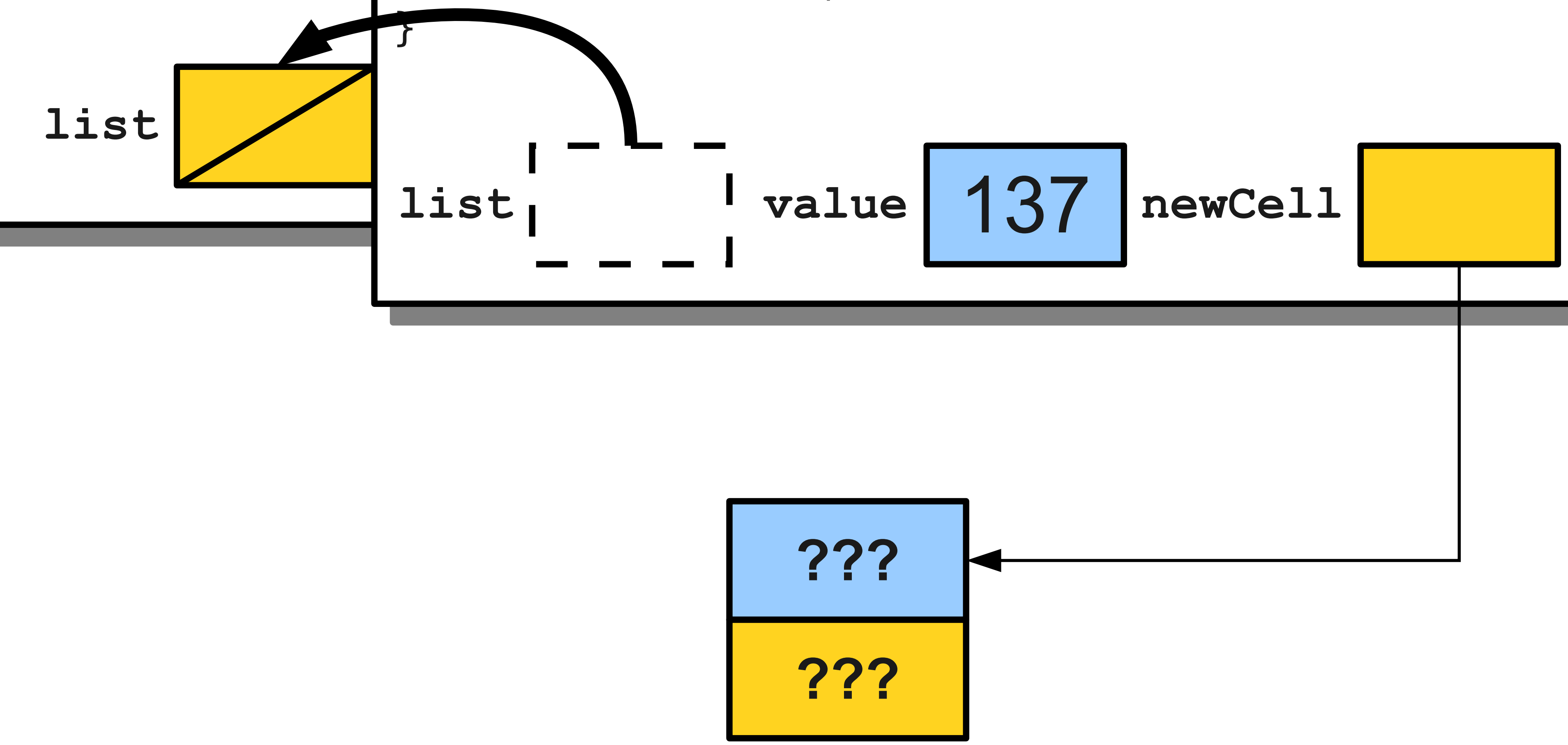


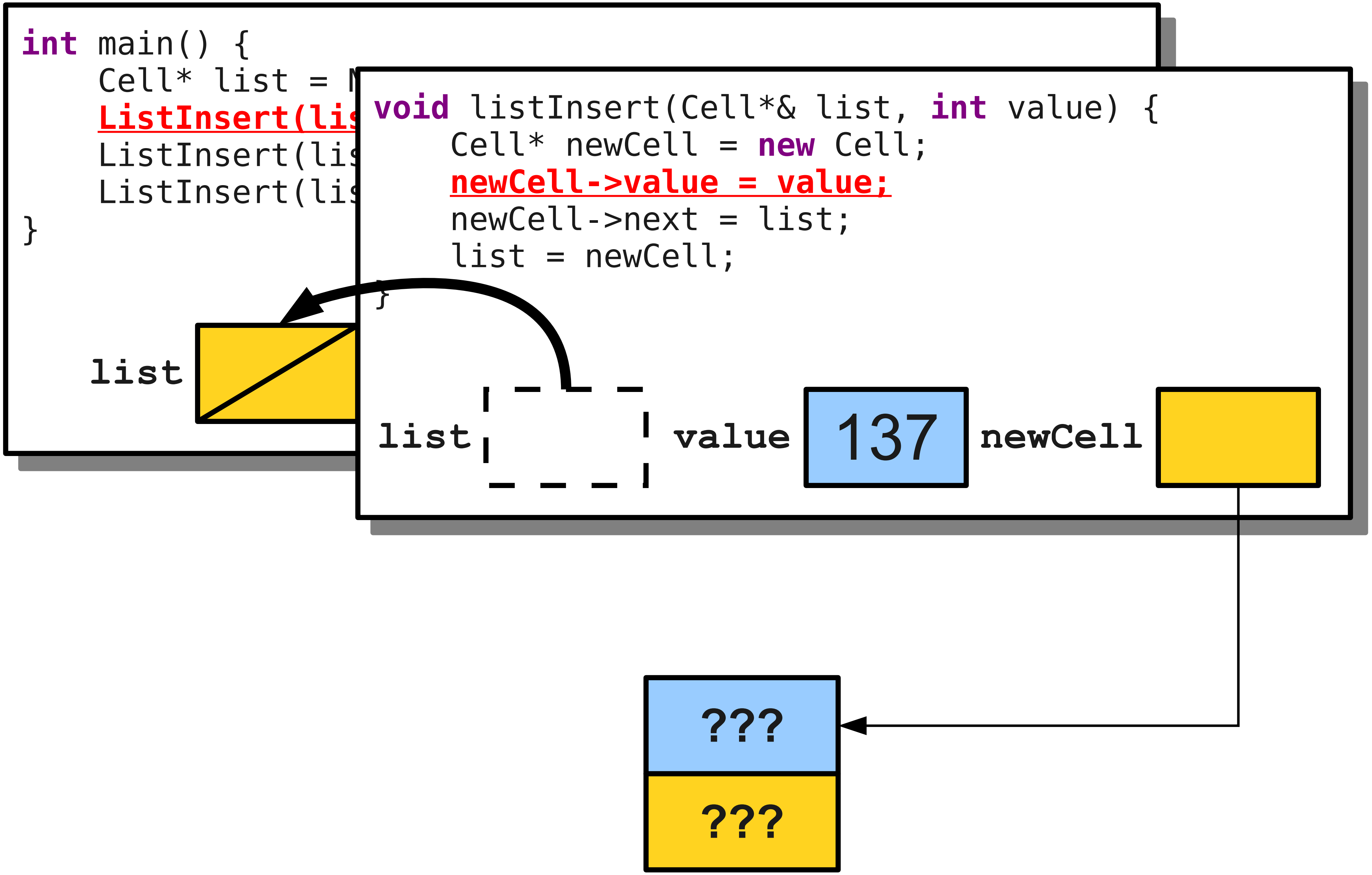
```
int main() {
    Cell* list = N
    ListInsert(lis
    ListInsert(lis
    ListInsert(lis
}
```

```
void listInsert(Cell*& list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```

```
int main() {
    Cell* list = N
    ListInsert(lis
    ListInsert(lis
    ListInsert(lis
}
```

```
void listInsert(Cell*& list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```



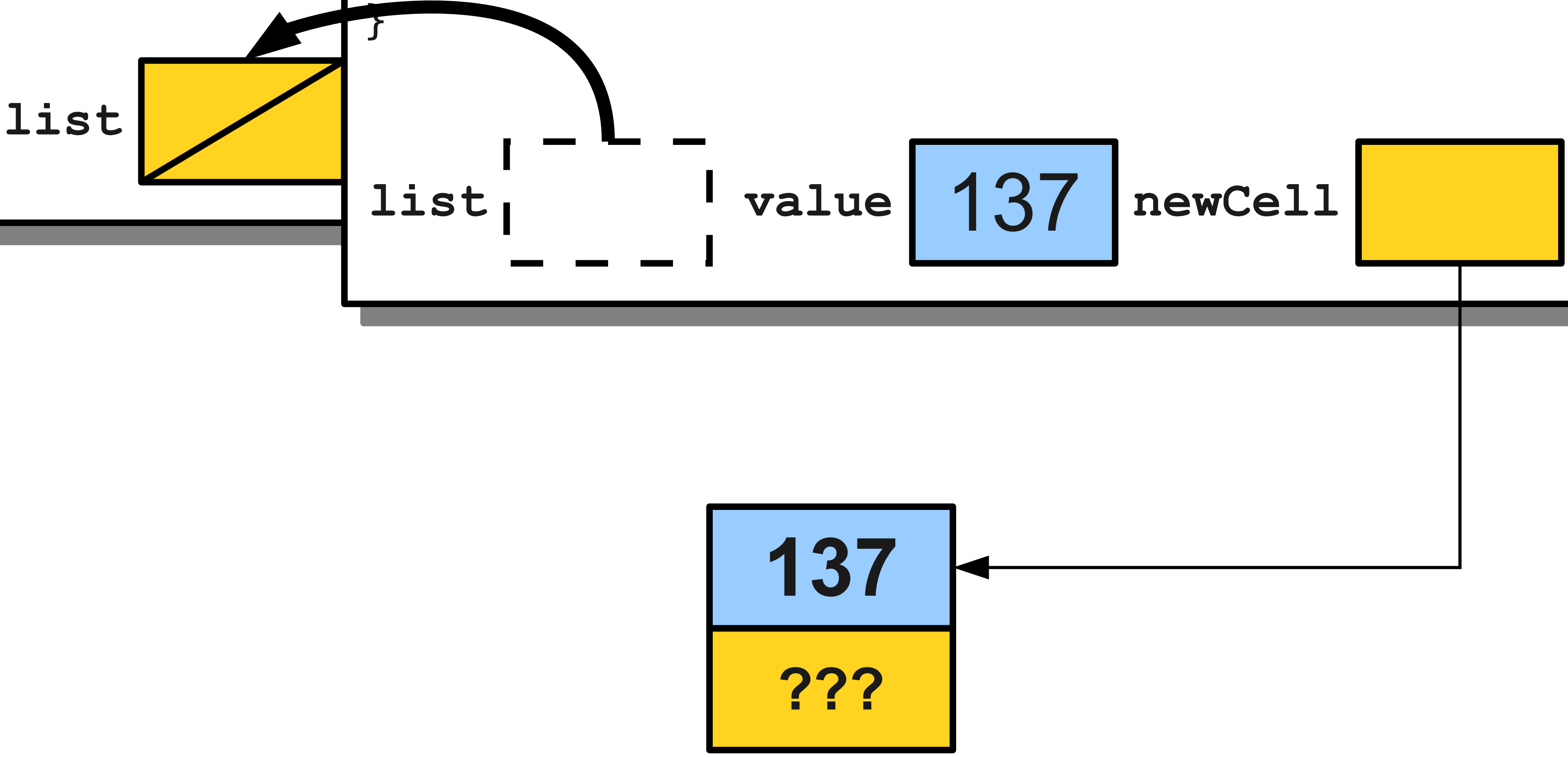


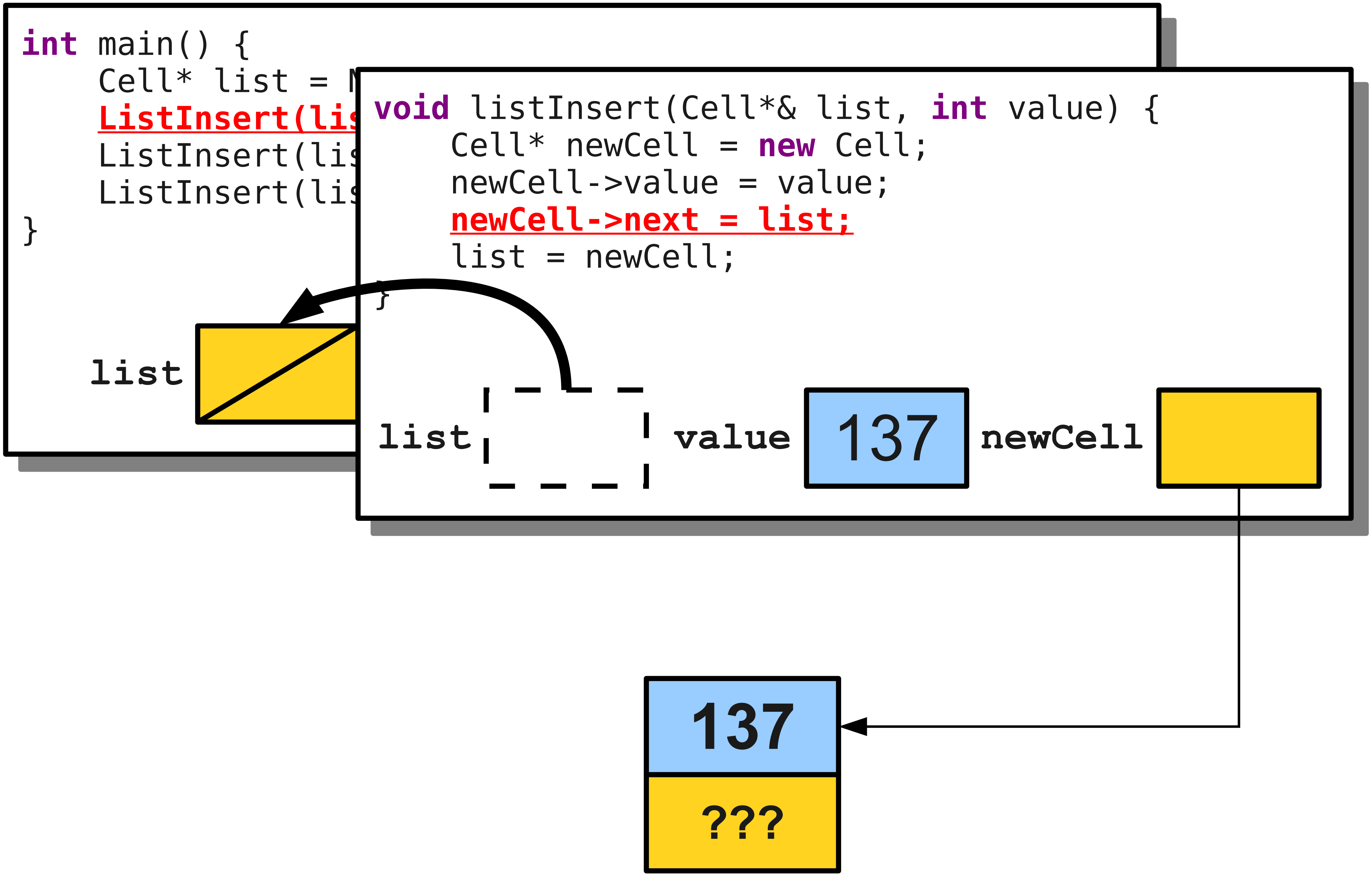
```
int main() {
    Cell* list = N
    ListInsert(lis
    ListInsert(lis
    ListInsert(lis
}
```

```
void listInsert(Cell*& list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```

```
int main() {
    Cell* list = N
    ListInsert(lis
    ListInsert(lis
    ListInsert(lis
}
```

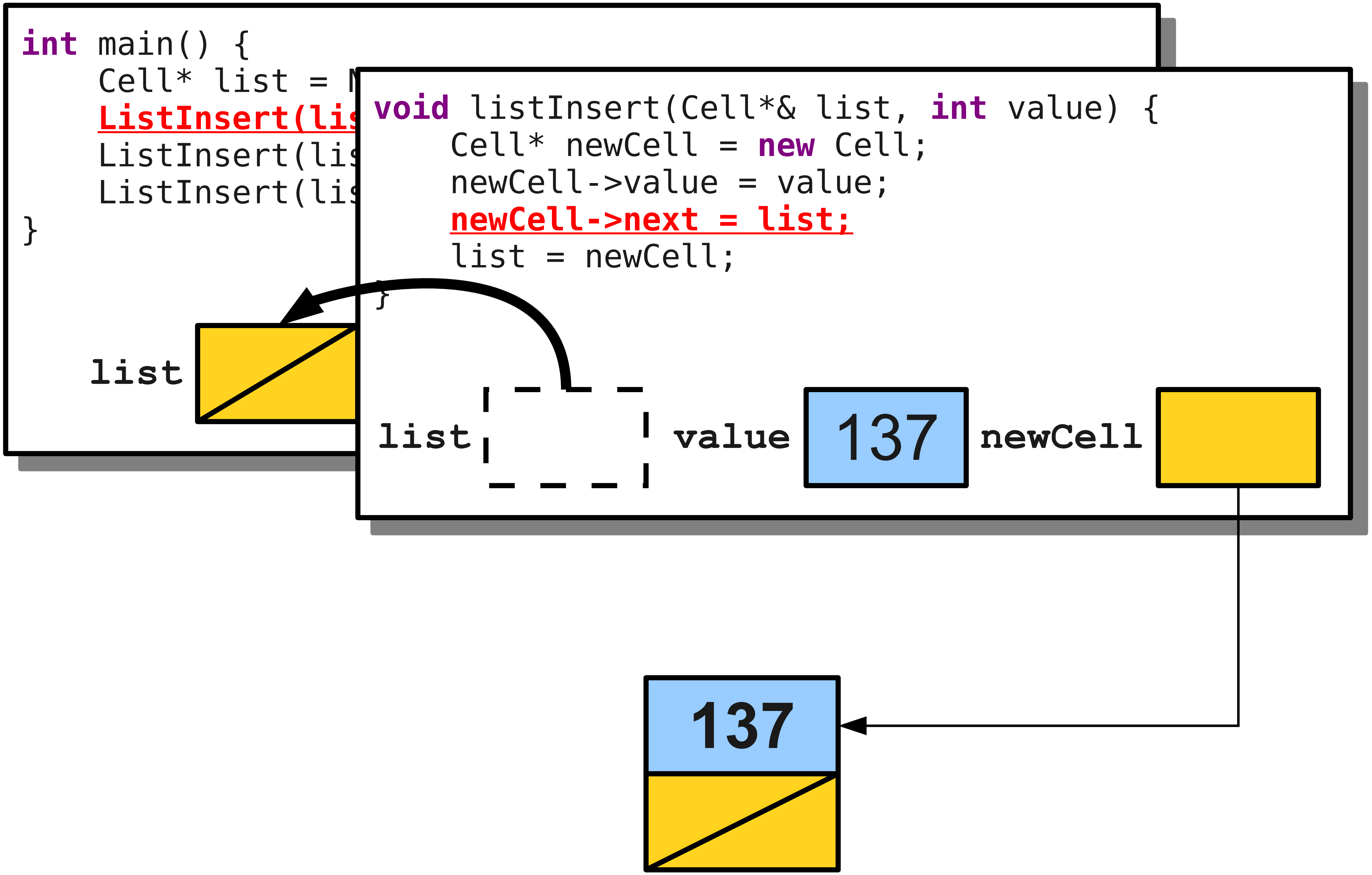
```
void listInsert(Cell*& list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```





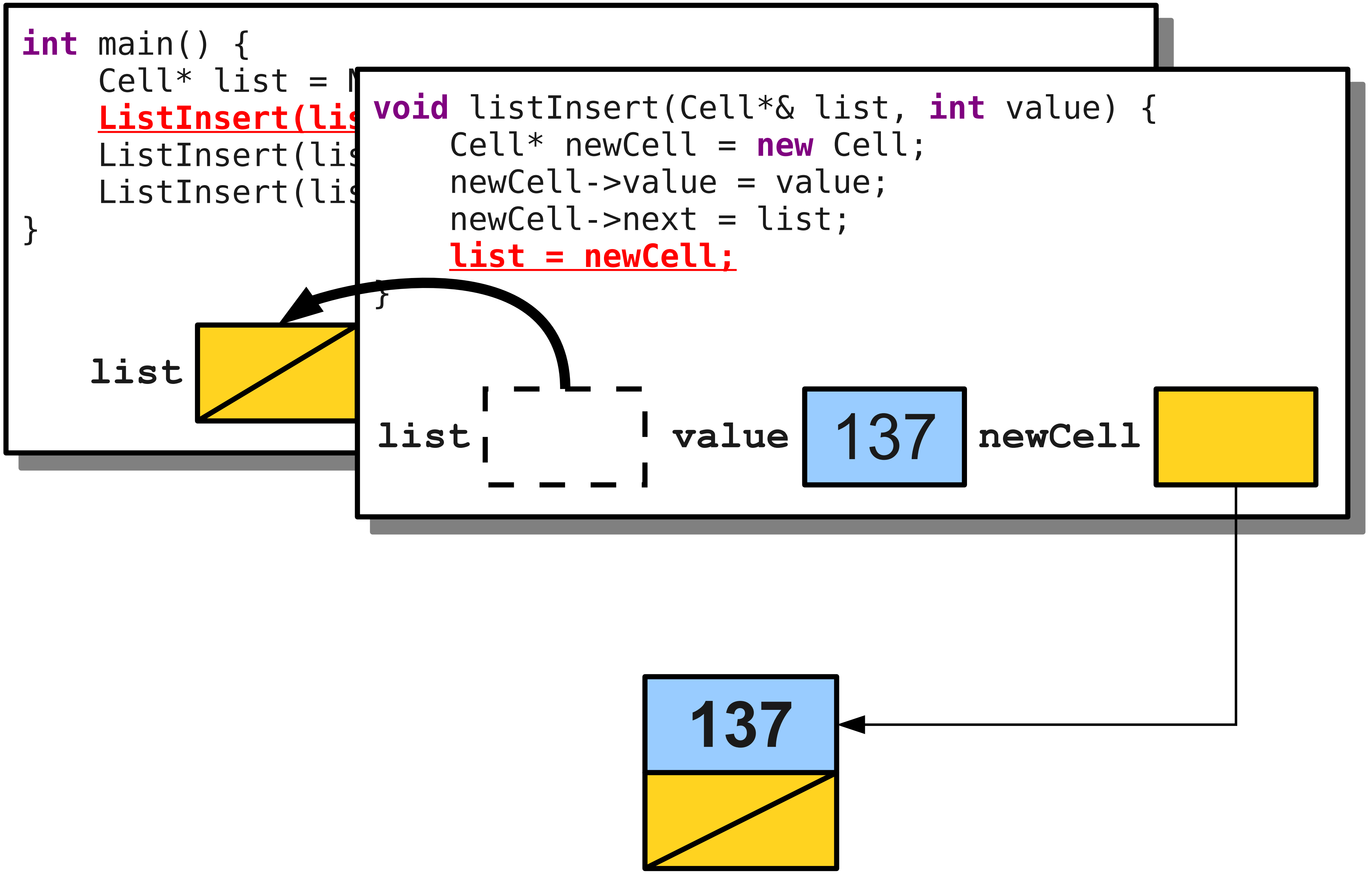
```
int main() {
    Cell* list = N
    ListInsert(lis
    ListInsert(lis
    ListInsert(lis
}
```

```
void listInsert(Cell*& list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```



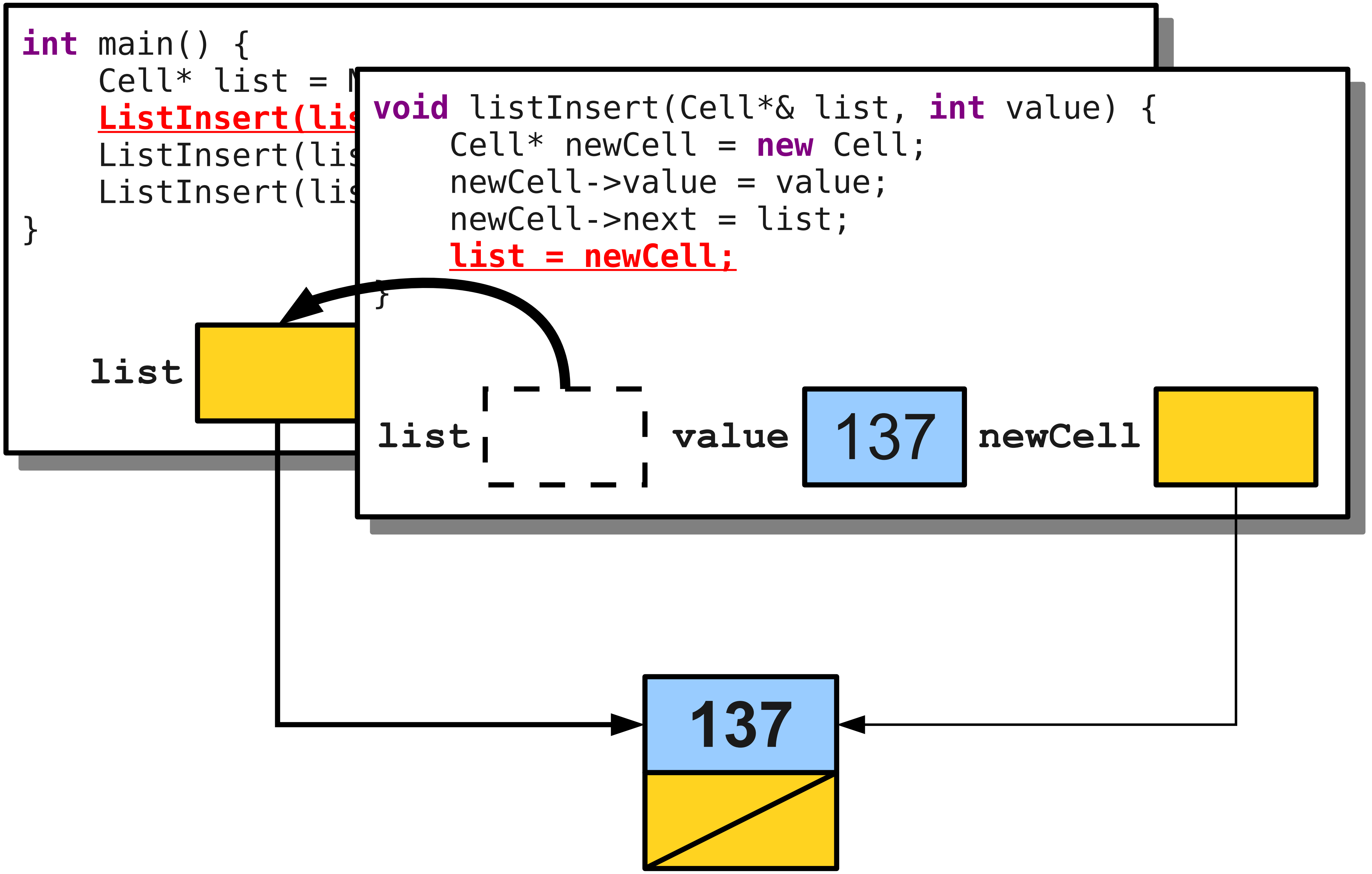
```
int main() {
    Cell* list = N
    ListInsert(lis
    ListInsert(lis
    ListInsert(lis
}
```

```
void listInsert(Cell*& list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```



```
int main() {
    Cell* list = N
    ListInsert(lis
    ListInsert(lis
    ListInsert(lis
}
```

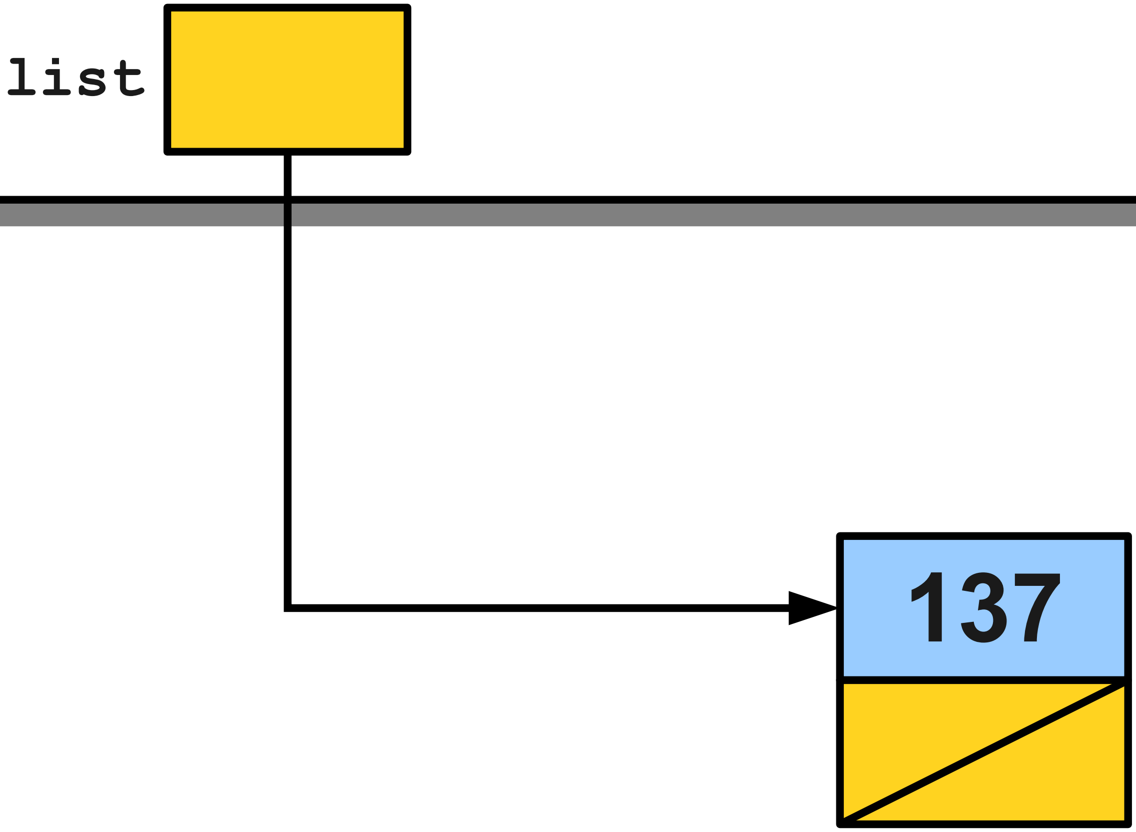
```
void listInsert(Cell*& list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```



```
int main() {
    Cell* list = N
    ListInsert(lis
    ListInsert(lis
    ListInsert(lis
}
```

```
void listInsert(Cell*& list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```

```
int main() {
    Cell* list = NULL;
    ListInsert(list, 137);
    ListInsert(list, 42);
    ListInsert(list, 271);
}
```

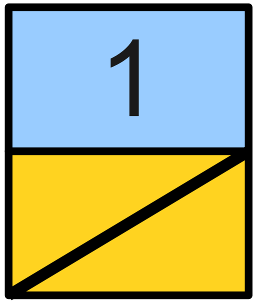


Reimplementing `Stack`

- We have already seen one way to implement the stack (dynamic arrays).
- We can also implement a stack efficiently using a linked list.
- **Push**: Prepend a new cell to the front of the list.
- **Pop**: Remove the first cell of the list.

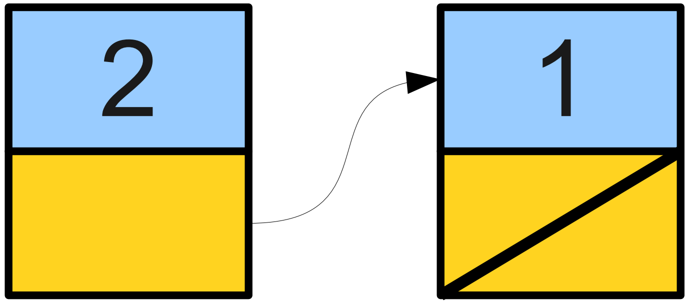
Reimplementing `Stack`

- We have already seen one way to implement the stack (dynamic arrays).
- We can also implement a stack efficiently using a linked list.
- **Push**: Prepend a new cell to the front of the list.
- **Pop**: Remove the first cell of the list.



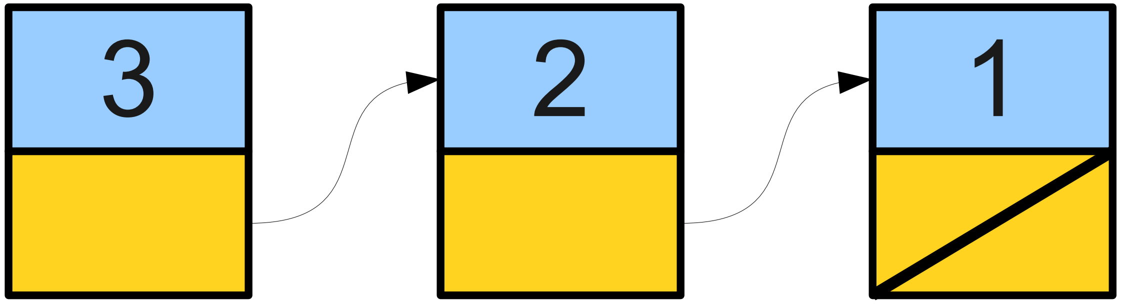
Reimplementing `Stack`

- We have already seen one way to implement the stack (dynamic arrays).
- We can also implement a stack efficiently using a linked list.
- **Push**: Prepend a new cell to the front of the list.
- **Pop**: Remove the first cell of the list.



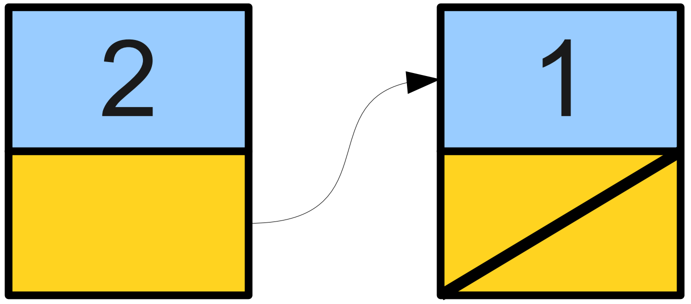
Reimplementing `Stack`

- We have already seen one way to implement the stack (dynamic arrays).
- We can also implement a stack efficiently using a linked list.
- **Push**: Prepend a new cell to the front of the list.
- **Pop**: Remove the first cell of the list.



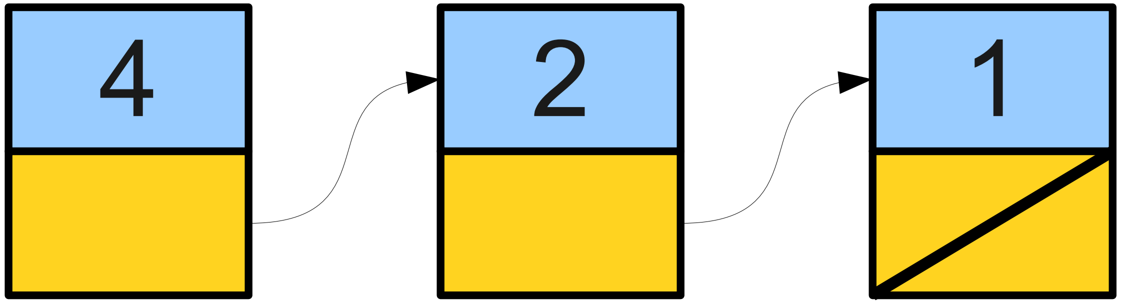
Reimplementing `Stack`

- We have already seen one way to implement the stack (dynamic arrays).
- We can also implement a stack efficiently using a linked list.
- **Push**: Prepend a new cell to the front of the list.
- **Pop**: Remove the first cell of the list.



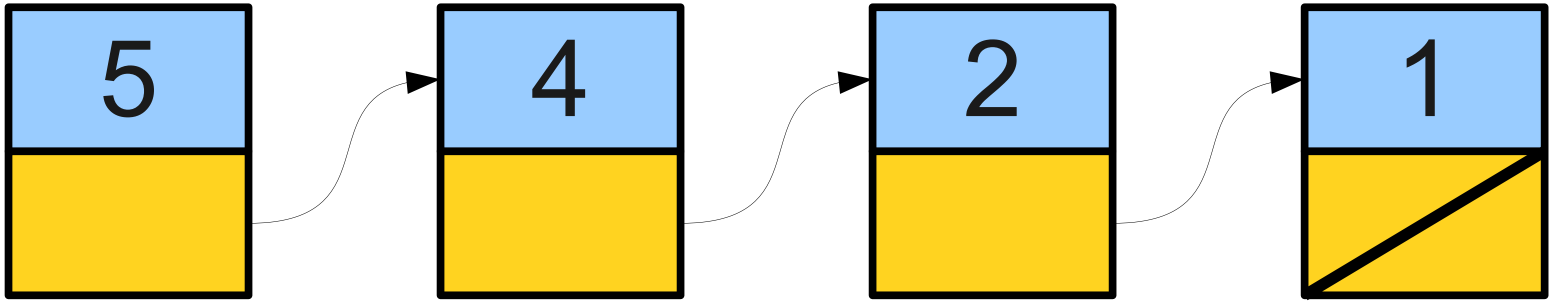
Reimplementing `Stack`

- We have already seen one way to implement the stack (dynamic arrays).
- We can also implement a stack efficiently using a linked list.
- **Push**: Prepend a new cell to the front of the list.
- **Pop**: Remove the first cell of the list.



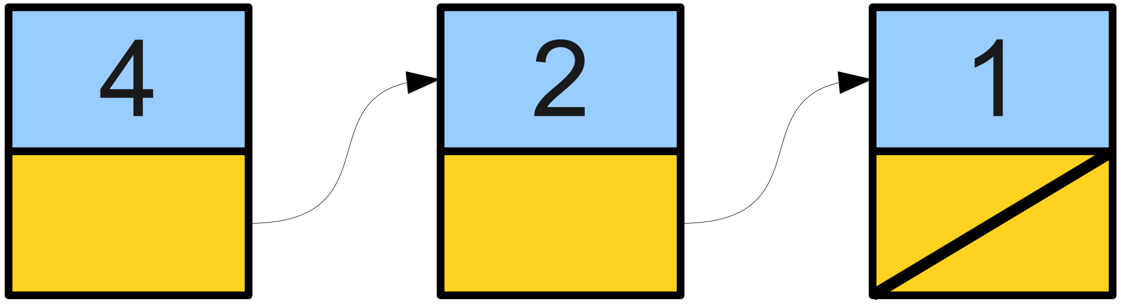
Reimplementing `Stack`

- We have already seen one way to implement the stack (dynamic arrays).
- We can also implement a stack efficiently using a linked list.
- **Push**: Prepend a new cell to the front of the list.
- **Pop**: Remove the first cell of the list.



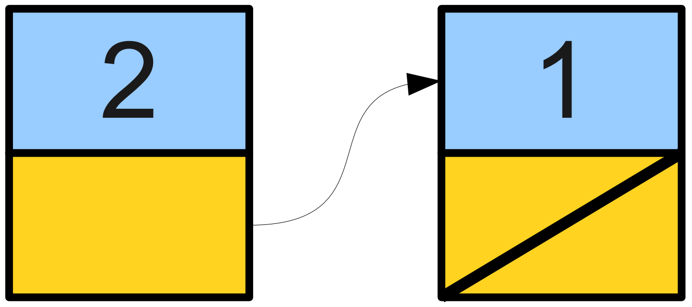
Reimplementing `Stack`

- We have already seen one way to implement the stack (dynamic arrays).
- We can also implement a stack efficiently using a linked list.
- **Push**: Prepend a new cell to the front of the list.
- **Pop**: Remove the first cell of the list.



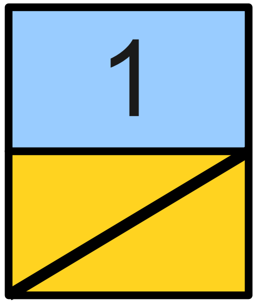
Reimplementing `Stack`

- We have already seen one way to implement the stack (dynamic arrays).
- We can also implement a stack efficiently using a linked list.
- **Push**: Prepend a new cell to the front of the list.
- **Pop**: Remove the first cell of the list.



Reimplementing `Stack`

- We have already seen one way to implement the stack (dynamic arrays).
- We can also implement a stack efficiently using a linked list.
- **Push**: Prepend a new cell to the front of the list.
- **Pop**: Remove the first cell of the list.



Reimplementing `Stack`

- We have already seen one way to implement the stack (dynamic arrays).
- We can also implement a stack efficiently using a linked list.
- **Push**: Prepend a new cell to the front of the list.
- **Pop**: Remove the first cell of the list.

Analyzing our Stack

- Push and pop are now **worst-case** O(1) instead of **average-case** O(1).
- What about the total runtime?

Analyzing our Stack

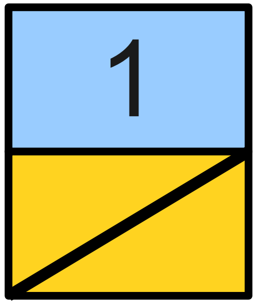
- Push and pop are now **worst-case** O(1) instead of **average-case** O(1).
- What about the total runtime?
- **Slower than before**.
- Why?
  - Cost of allocating individual linked list cells exceeds cost of allocating very few blocks and copying values over.
  - Trade average-case for worst-case speed.

Implementing `Queue`

- We can also implement the queue using a linked list.
- Idea:
  - To **enqueue**, append a new cell to the end of the list.
  - To **dequeue**, remove the first cell from the list.

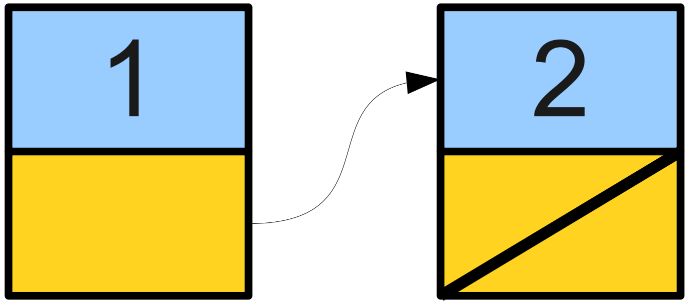
Implementing `Queue`

- We can also implement the queue using a linked list.
- Idea:
  - To **enqueue**, append a new cell to the end of the list.
  - To **dequeue**, remove the first cell from the list.



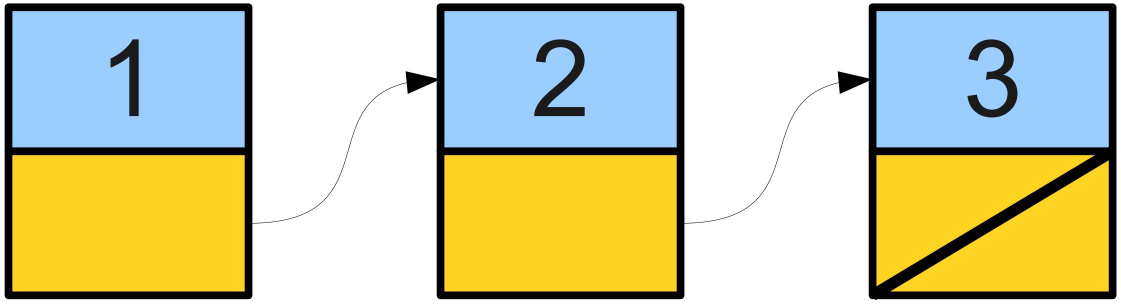
Implementing `Queue`

- We can also implement the queue using a linked list.
- Idea:
  - To **enqueue**, append a new cell to the end of the list.
  - To **dequeue**, remove the first cell from the list.



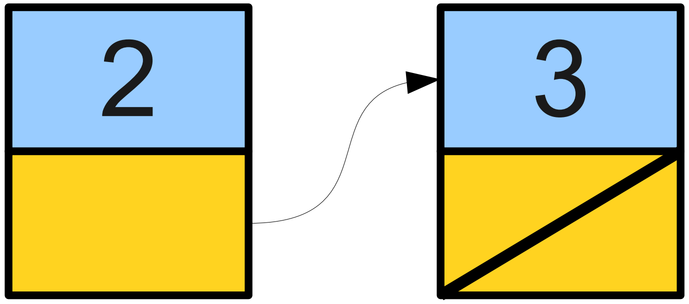
Implementing `Queue`

- We can also implement the queue using a linked list.
- Idea:
  - To **enqueue**, append a new cell to the end of the list.
  - To **dequeue**, remove the first cell from the list.



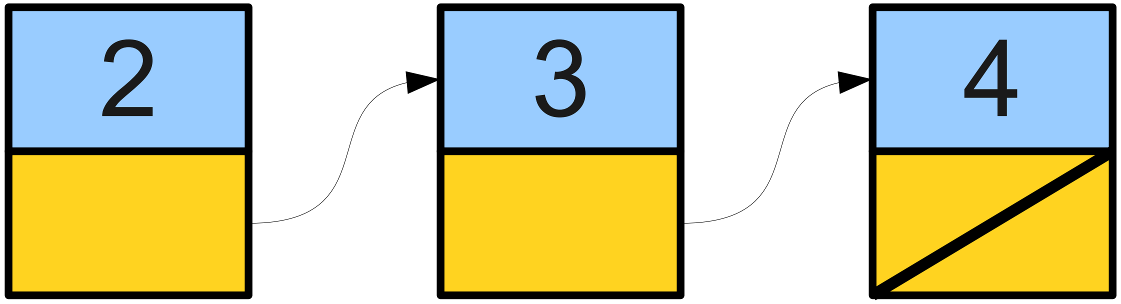
Implementing `Queue`

- We can also implement the queue using a linked list.
- Idea:
  - To **enqueue**, append a new cell to the end of the list.
  - To **dequeue**, remove the first cell from the list.



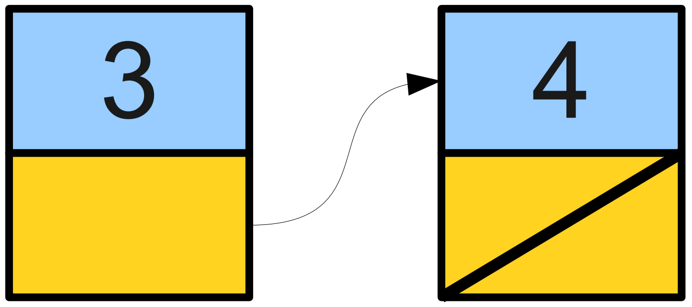
Implementing `Queue`

- We can also implement the queue using a linked list.
- Idea:
  - To **enqueue**, append a new cell to the end of the list.
  - To **dequeue**, remove the first cell from the list.



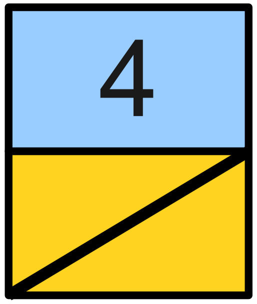
Implementing **`Queue`**

- We can also implement the queue using a linked list.
- Idea:
    - To **enqueue**, append a new cell to the end of the list.
    - To **dequeue**, remove the first cell from the list.



Implementing `Queue`

- We can also implement the queue using a linked list.
- Idea:
    - To **enqueue**, append a new cell to the end of the list.
    - To **dequeue**, remove the first cell from the list.



Implementing `Queue`

- We can also implement the queue using a linked list.
- Idea:
  - To **enqueue**, append a new cell to the end of the list.
  - To **dequeue**, remove the first cell from the list.

Analyzing Efficiency

- What is the big-O complexity of a dequeue?
- Answer: **O(1)**.
- What is the big-O complexity of an enqueue?
- Answer: **O(*n*)**.

Improving Efficiency

- The O(*n*) work in enqueue comes from scanning the list to find the end.
- **Idea**: What if we just stored a pointer to the very last cell in the list?
- Can immediately jump to the end to append a value.

Analyzing Efficiency

- What is the big-O complexity of a dequeue?
- Answer: **O(1)**.
- What is the big-O complexity of an enqueue?
- Answer: **O(1)**.

The Takeaway Point

- You can have multiple pointers into the same linked list.
- This makes it possible to efficiently insert values at multiple places in the list.

Next Time

- **Implementing Maps**
  - Implementation strategies
  - Hashing
  - Building a hash table