Linked Lists

Part Two

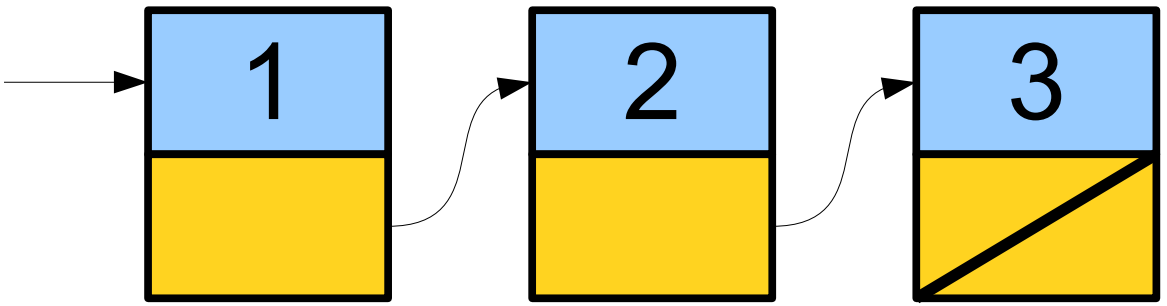
Announcements

- Assignment 4 due right now
- Assignment 5: **Priority Queue** out today!
  - Really important data structure for pathfinding.
- **PLEASE PICK UP YOUR MIDTERMS!**
  - Please let me know if you find any grading mistakes!
  - Reminder: Reduced points for infinite recursion.

Recap from Last Time

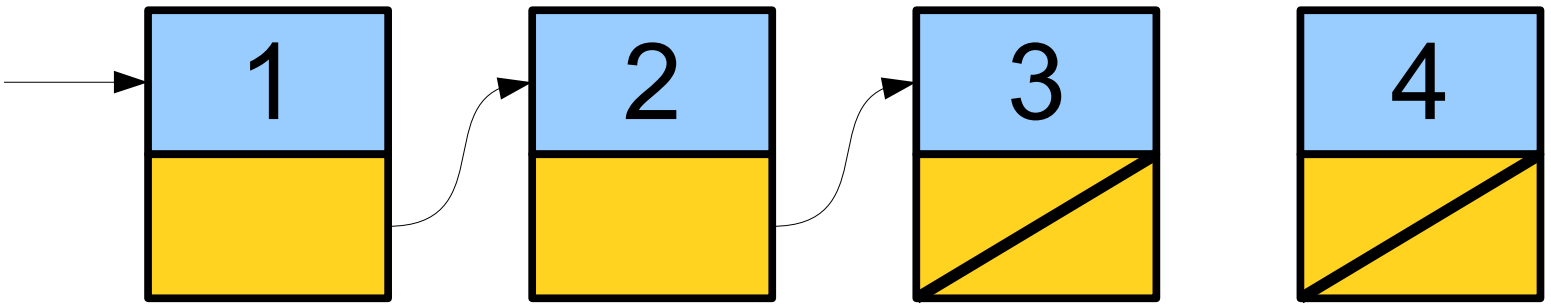
Linked Lists at a Glance

- 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.



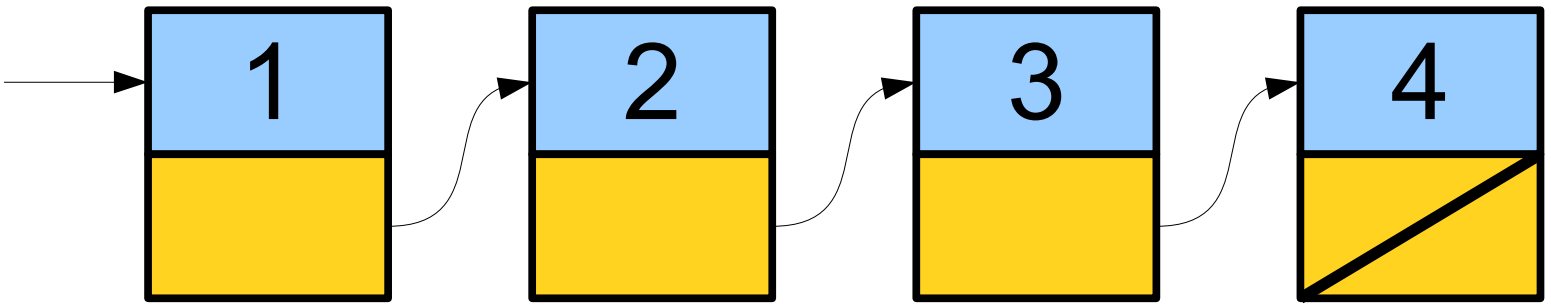
Linked Lists at a Glance

- 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.



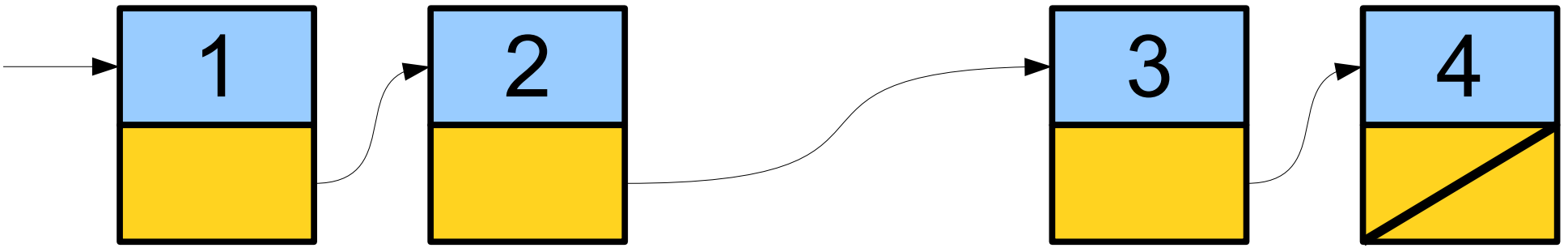
Linked Lists at a Glance

- 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.



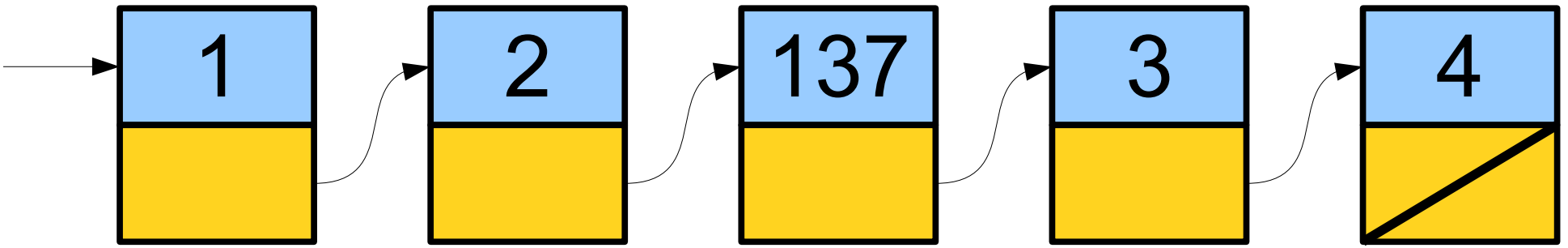
Linked Lists at a Glance

- 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.



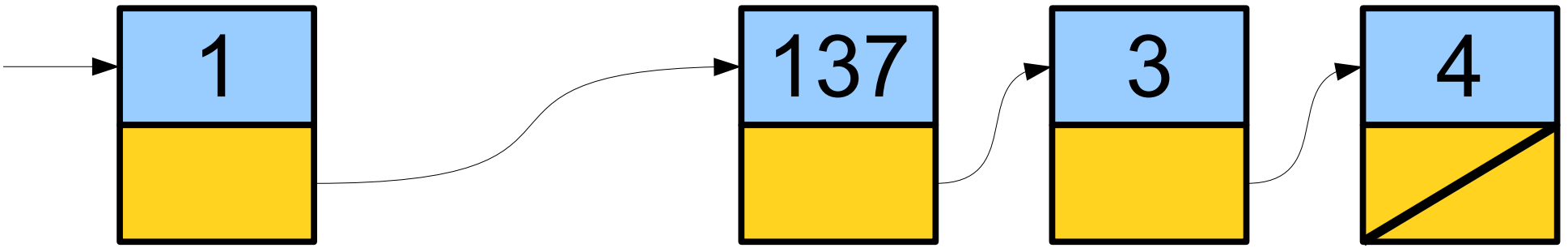
Linked Lists at a Glance

- 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 {
    Type value;
    Cell* next;
};
```

- **The structure is defined recursively!**

Building a Linked List

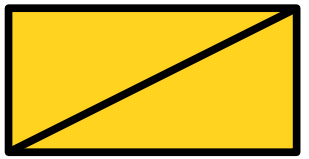
```
Cell* result = NULL;
while (true) {
   string line = getLine("Next entry? ");
   if (line == "") break;

   Cell* cell = new Cell;
   cell->value = line;

   cell->next = result;
   result = cell;
}
return result;
```

```
Cell* result = NULL;
while (true) {
   string line = getLine("Next entry? ");
   if (line == "") break;

   Cell* cell = new Cell;
   cell->value = line;

   cell->next = result;
   result = cell;
}
return result;
```

```
Cell* result = NULL;
while (true) {
   string line = getLine("Next entry? ");
   if (line == "") break;

   Cell* cell = new Cell;
   cell->value = line;

   cell->next = result;
   result = cell;
}
return result;
```

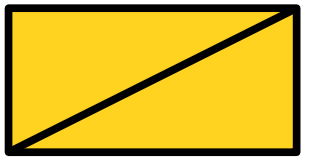
```
Cell* result = NULL;
while (true) {
   string line = getLine("Next entry? ");
   if (line == "") break;

   Cell* cell = new Cell;
   cell->value = line;

   cell->next = result;
   result = cell;
}
return result;
```

result

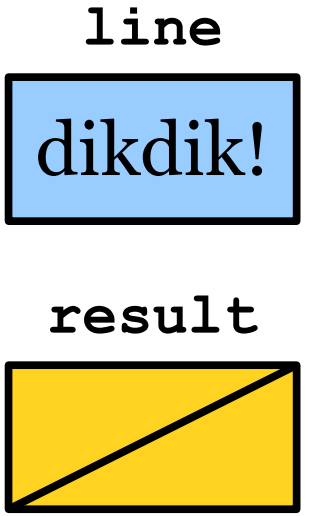


```
Cell* result = NULL;
while (true) {
   string line = getLine("Next entry? ");
   if (line == "") break;

   Cell* cell = new Cell;
   cell->value = line;

   cell->next = result;
   result = cell;
}
return result;
```

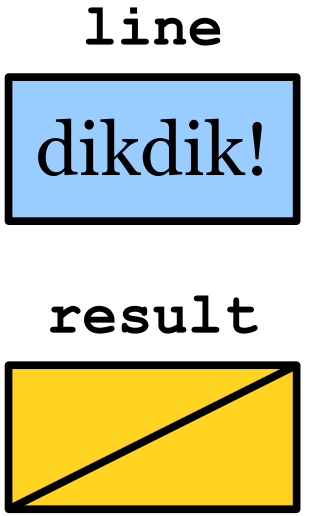
result



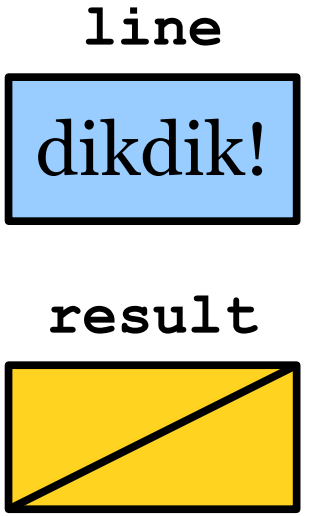
```
Cell* result = NULL;
while (true) {
   string line = getLine("Next entry? ");
   if (line == "") break;

   Cell* cell = new Cell;
   cell->value = line;

   cell->next = result;
   result = cell;
}
return result;
```



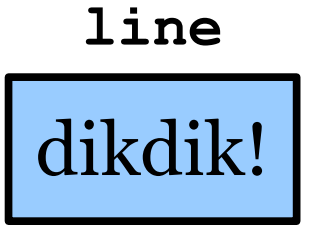
```
Cell* result = NULL;
while (true) {
   string line = getLine("Next entry? ");
   if (line == "") break;

   Cell* cell = new Cell;
   cell->value = line;

   cell->next = result;
   result = cell;
}
return result;
```

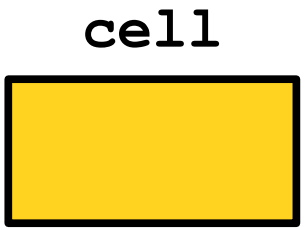


```
Cell* result = NULL;
while (true) {
   string line = getLine("Next entry? ");
   if (line == "") break;

   Cell* cell = new Cell;
   cell->value = line;

   cell->next = result;
   result = cell;
}
return result;
```

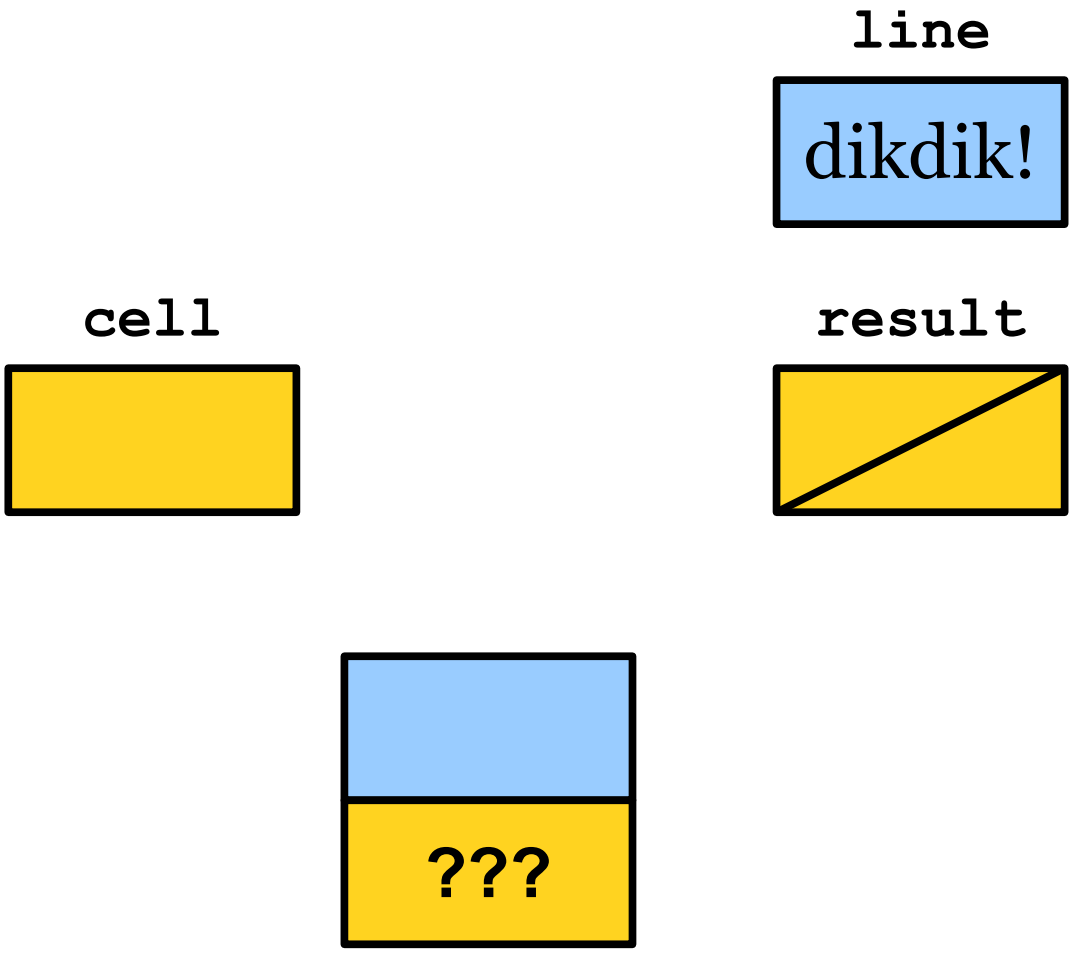


```
Cell* result = NULL;
while (true) {
   string line = getLine("Next entry? ");
   if (line == "") break;

   Cell* cell = new Cell;
   cell->value = line;

   cell->next = result;
   result = cell;
}
return result;
```

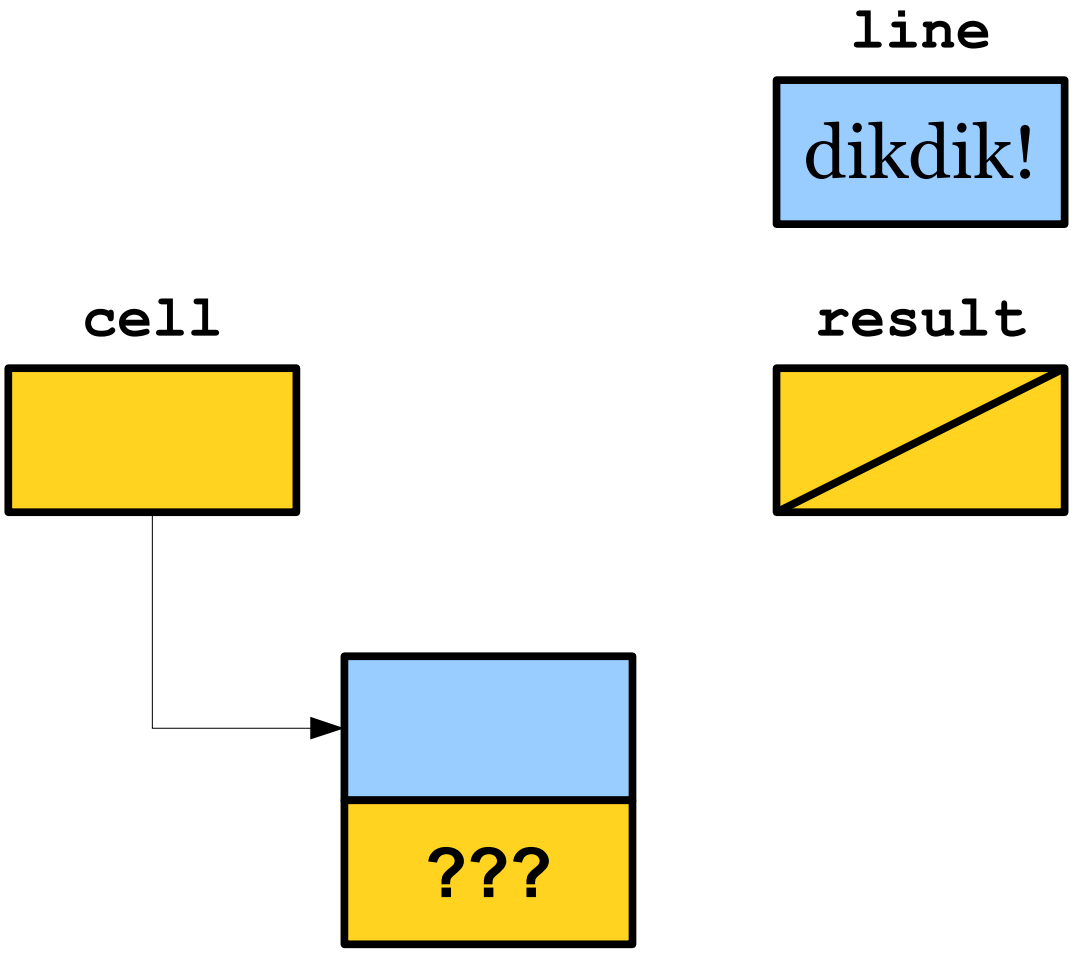




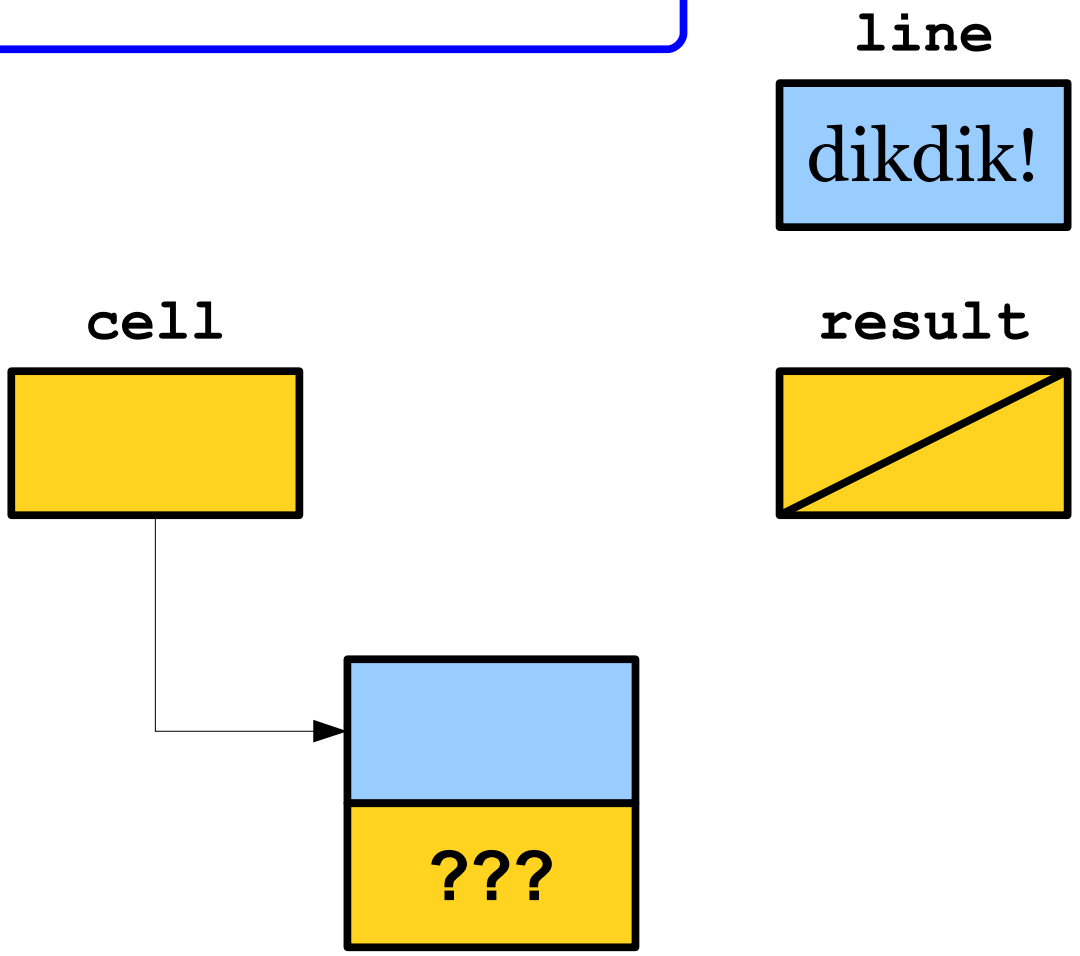
```
Cell* result = NULL;
while (true) {
   string line = getLine("Next entry? ");
   if (line == "") break;

   Cell* cell = new Cell;
   cell->value = line;

   cell->next = result;
   result = cell;
}
return result;
```



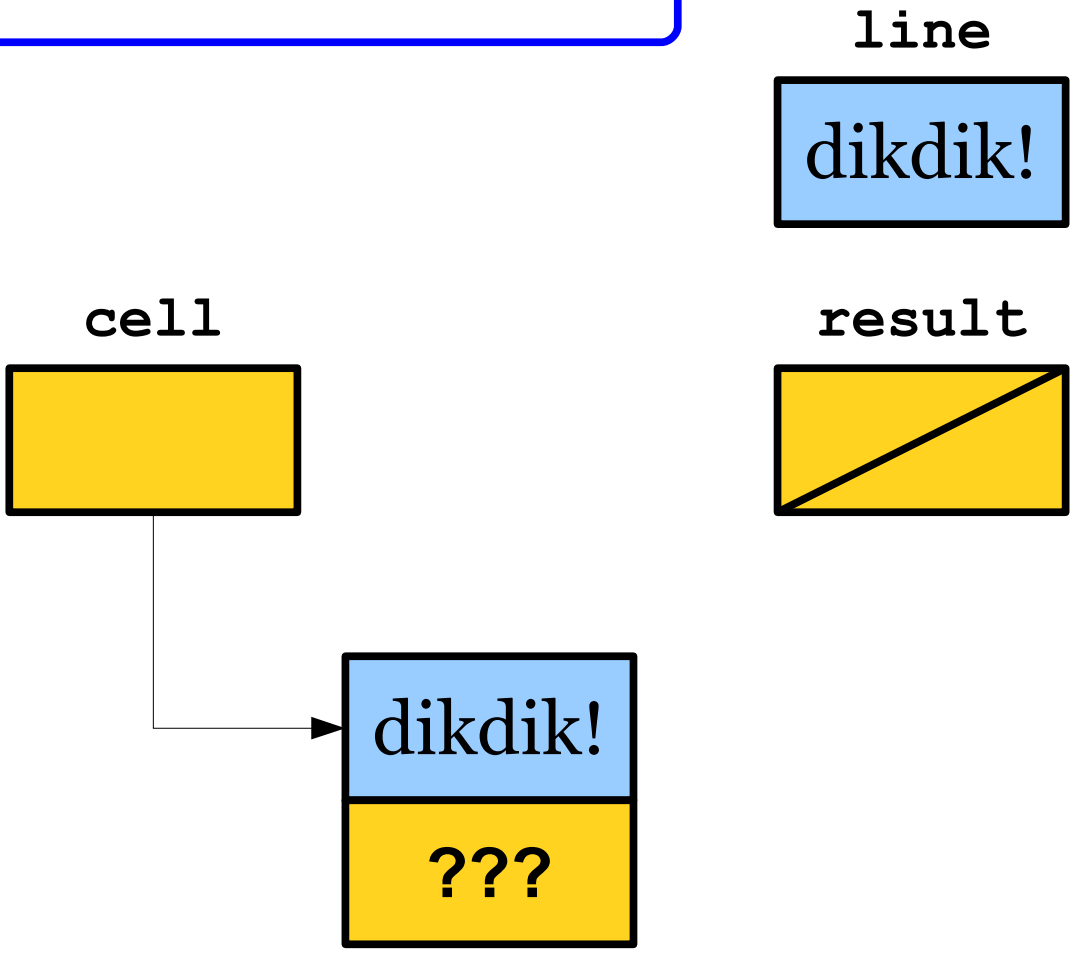
```
Cell* result = NULL;
while (true) {
   string line = getLine("Next entry? ");
   if (line == "") break;

   Cell* cell = new Cell;
   cell->value = line;

   cell->next = result;
   result = cell;
}
return result;
```



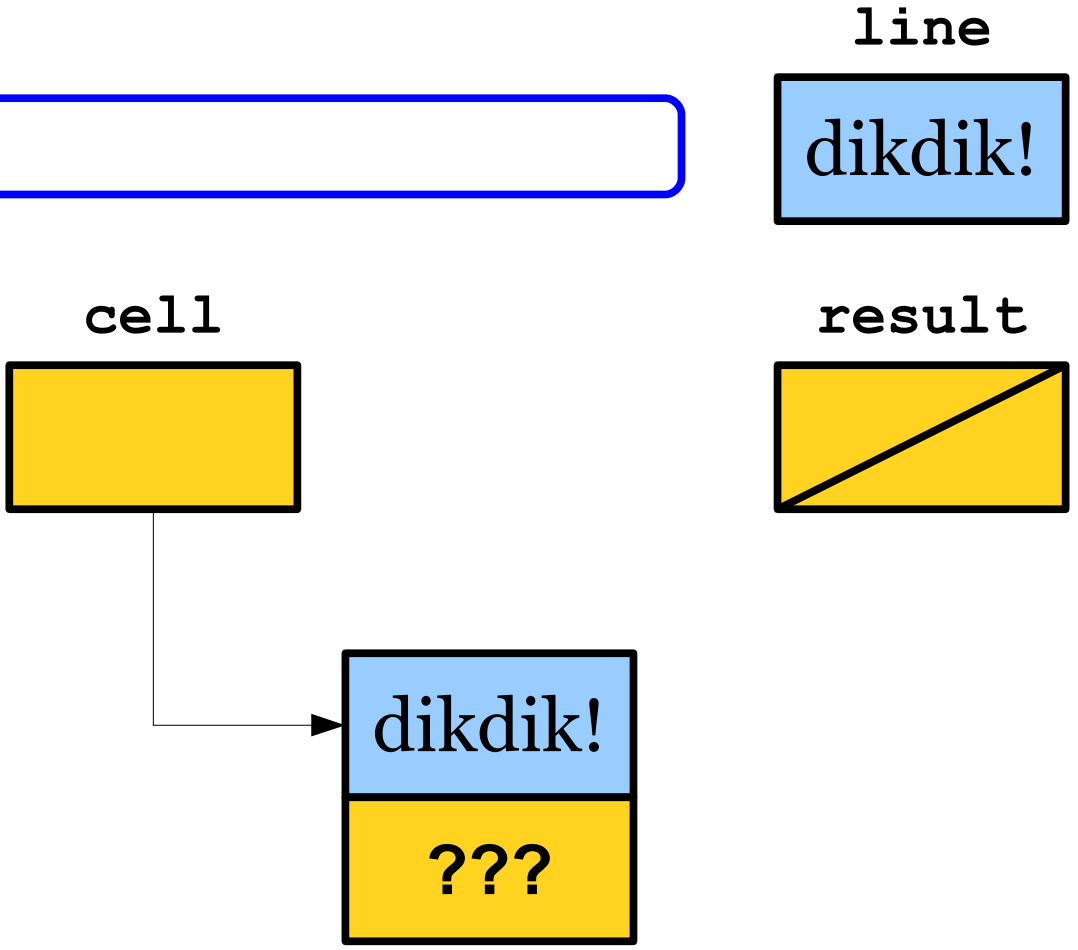
```
Cell* result = NULL;
while (true) {
   string line = getLine("Next entry? ");
   if (line == "") break;

   Cell* cell = new Cell;
   cell->value = line;

   cell->next = result;
   result = cell;
}
return result;
```



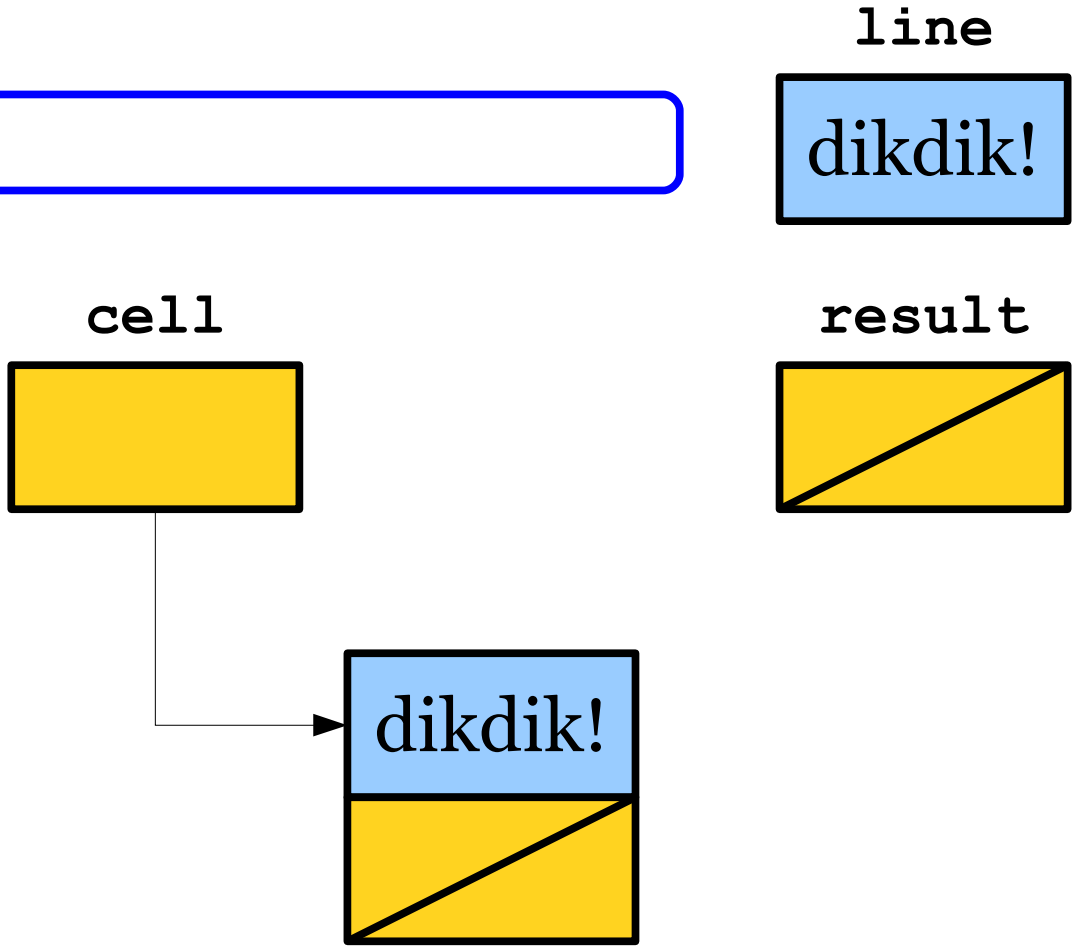
```
Cell* result = NULL;
while (true) {
   string line = getLine("Next entry? ");
   if (line == "") break;

   Cell* cell = new Cell;
   cell->value = line;

   cell->next = result;
   result = cell;
}
return result;
```



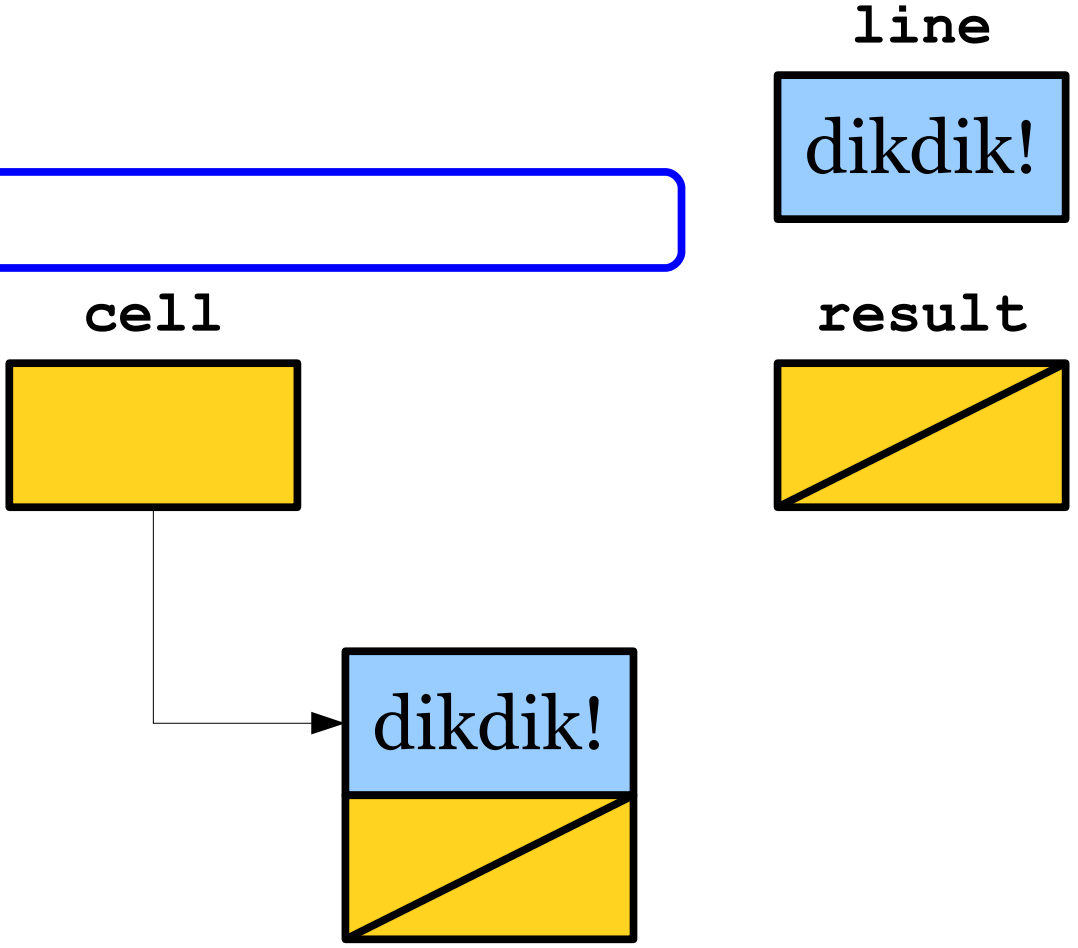
```
Cell* result = NULL;
while (true) {
   string line = getLine("Next entry? ");
   if (line == "") break;

   Cell* cell = new Cell;
   cell->value = line;

   cell->next = result;
   result = cell;
}
return result;
```



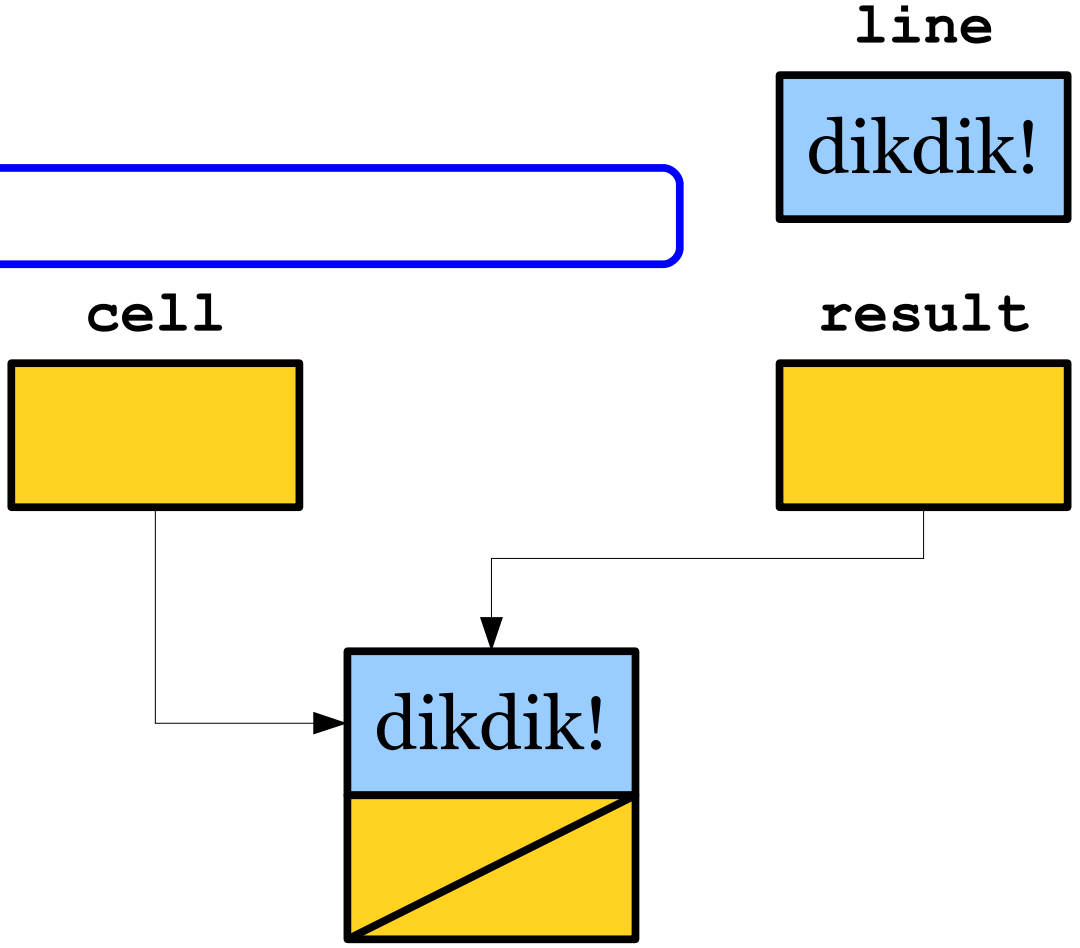
```
Cell* result = NULL;
while (true) {
   string line = getLine("Next entry? ");
   if (line == "") break;

   Cell* cell = new Cell;
   cell->value = line;

   cell->next = result;
   result = cell;
}
return result;
```



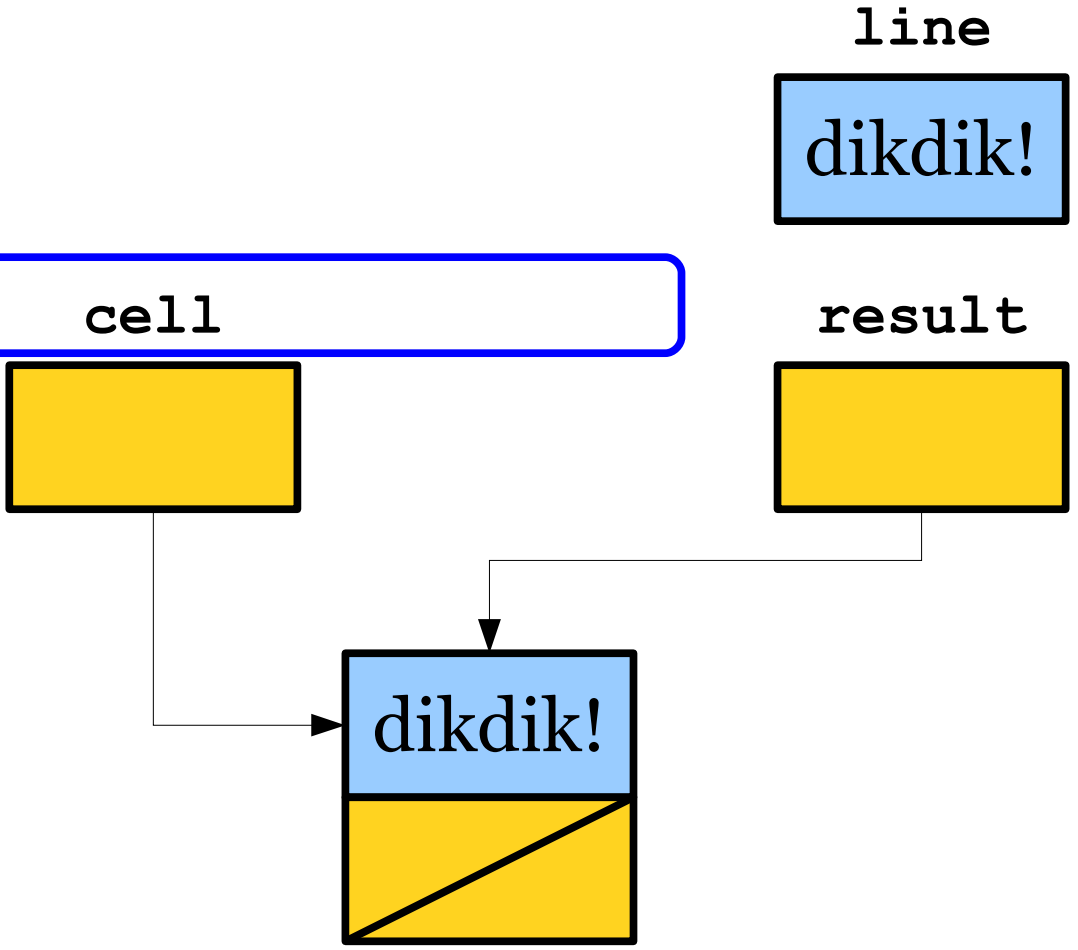
```
Cell* result = NULL;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```



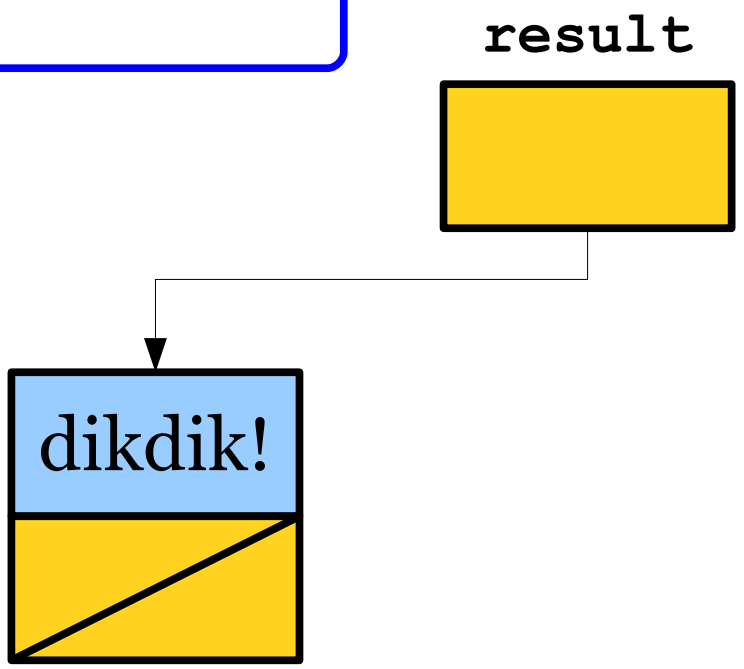
```
Cell* result = NULL;
while (true) {
   string line = getLine("Next entry? ");
   if (line == "") break;

   Cell* cell = new Cell;
   cell->value = line;

   cell->next = result;
   result = cell;
}
return result;
```



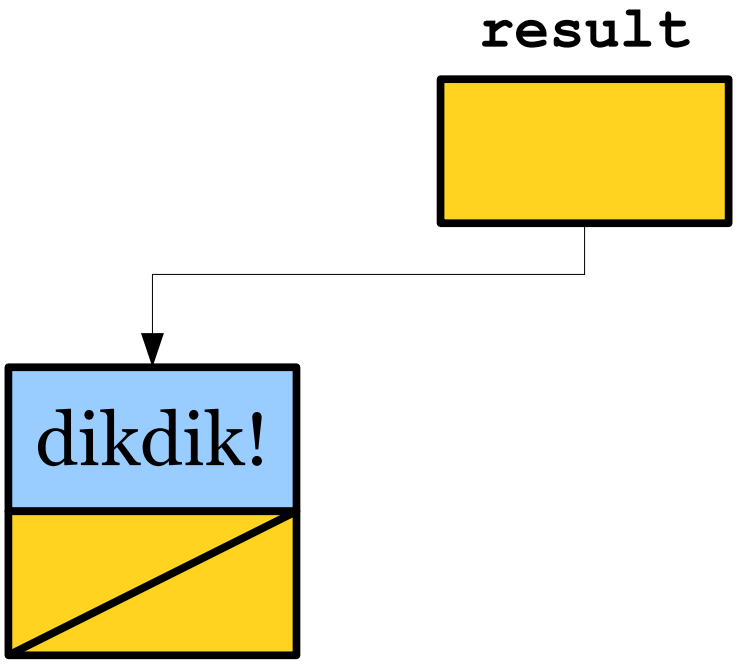
```
Cell* result = NULL;
while (true) {
   string line = getLine("Next entry? ");
   if (line == "") break;

   Cell* cell = new Cell;
   cell->value = line;

   cell->next = result;
   result = cell;
}
return result;
```



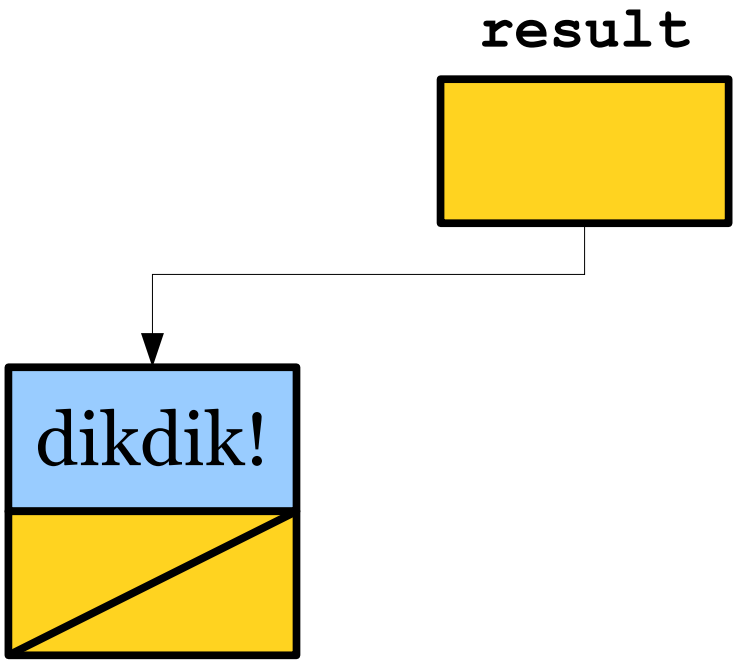
```
Cell* result = NULL;
while (true) {
   string line = getLine("Next entry? ");
   if (line == "") break;

   Cell* cell = new Cell;
   cell->value = line;

   cell->next = result;
   result = cell;
}
return result;
```



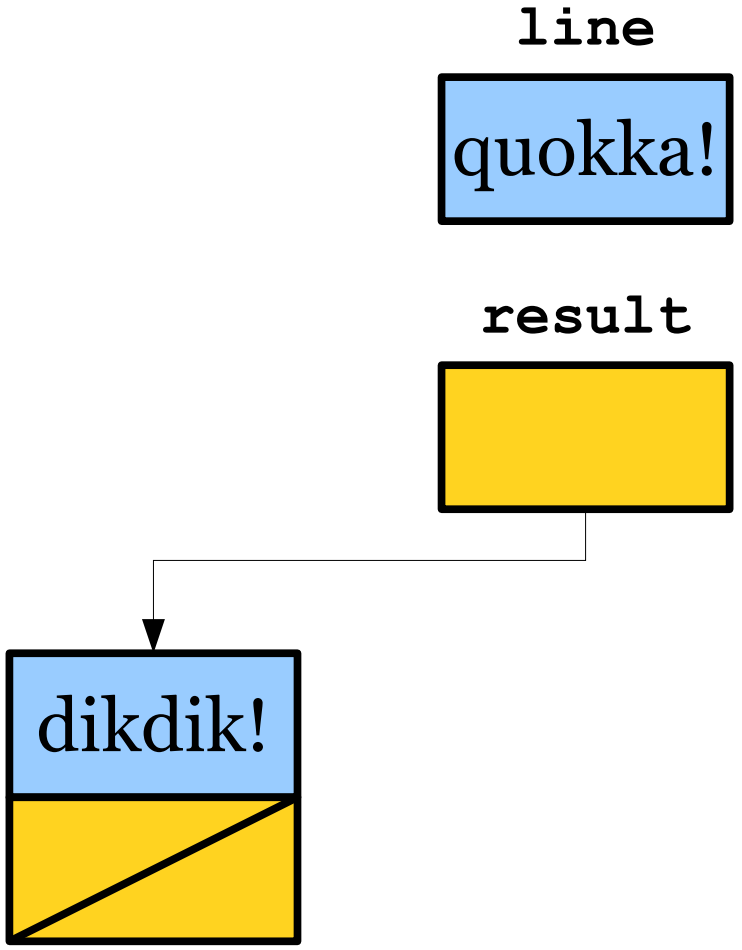
```
Cell* result = NULL;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```



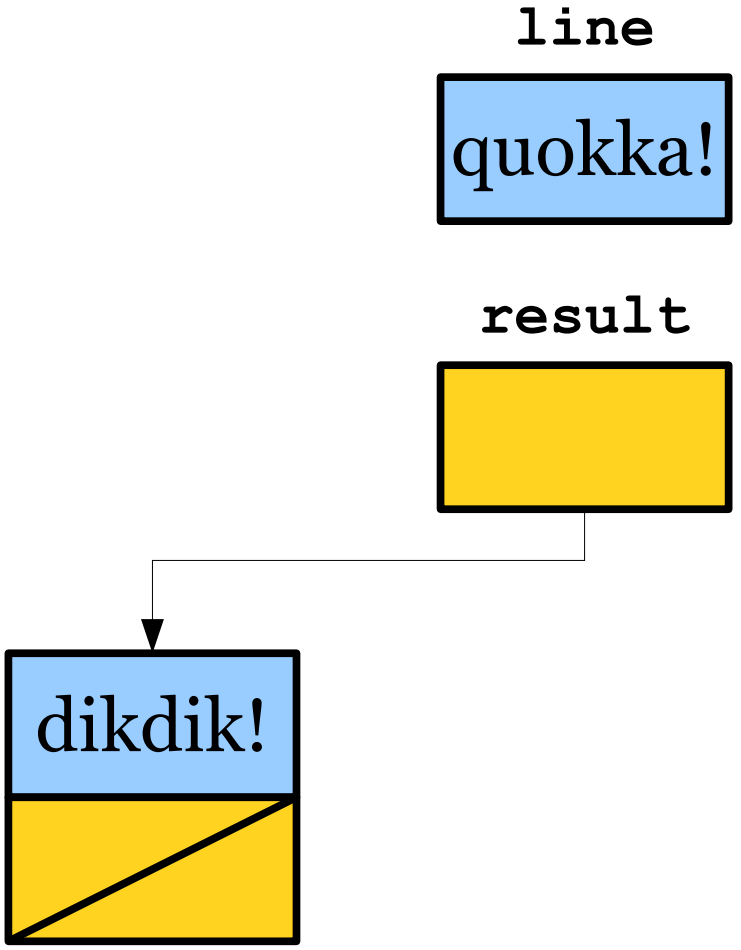
```
Cell* result = NULL;
while (true) {
   string line = getLine("Next entry? ");
   if (line == "") break;

   Cell* cell = new Cell;
   cell->value = line;

   cell->next = result;
   result = cell;
}
return result;
```



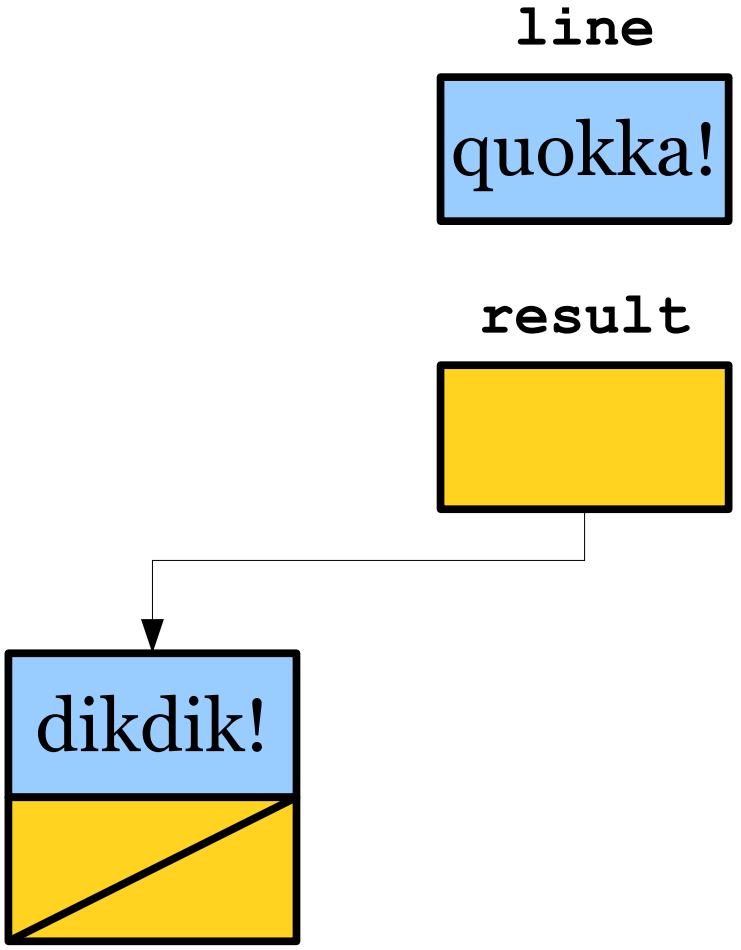
```
Cell* result = NULL;
while (true) {
   string line = getLine("Next entry? ");
   if (line == "") break;

   Cell* cell = new Cell;
   cell->value = line;

   cell->next = result;
   result = cell;
}
return result;
```



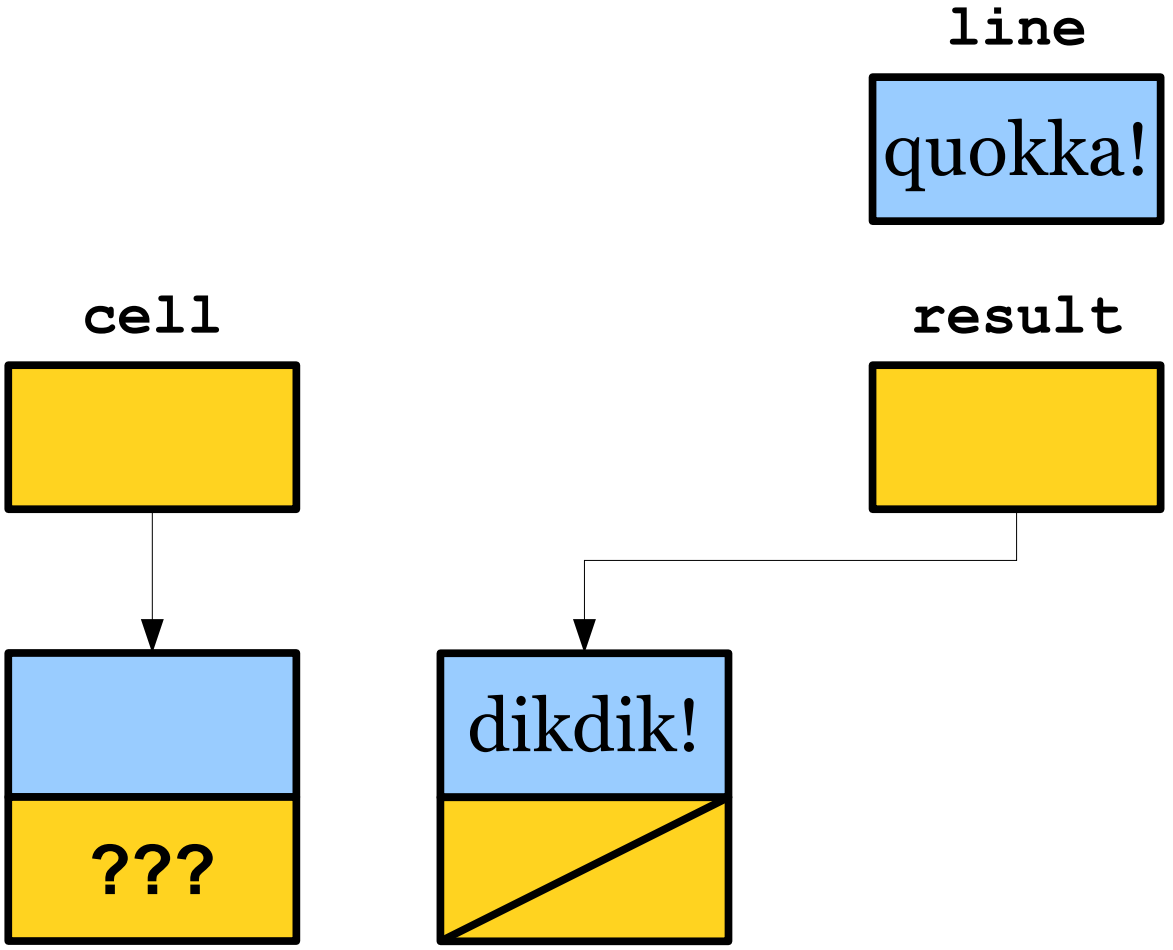
```
Cell* result = NULL;
while (true) {
   string line = getLine("Next entry? ");
   if (line == "") break;

   Cell* cell = new Cell;
   cell->value = line;

   cell->next = result;
   result = cell;
}
return result;
```



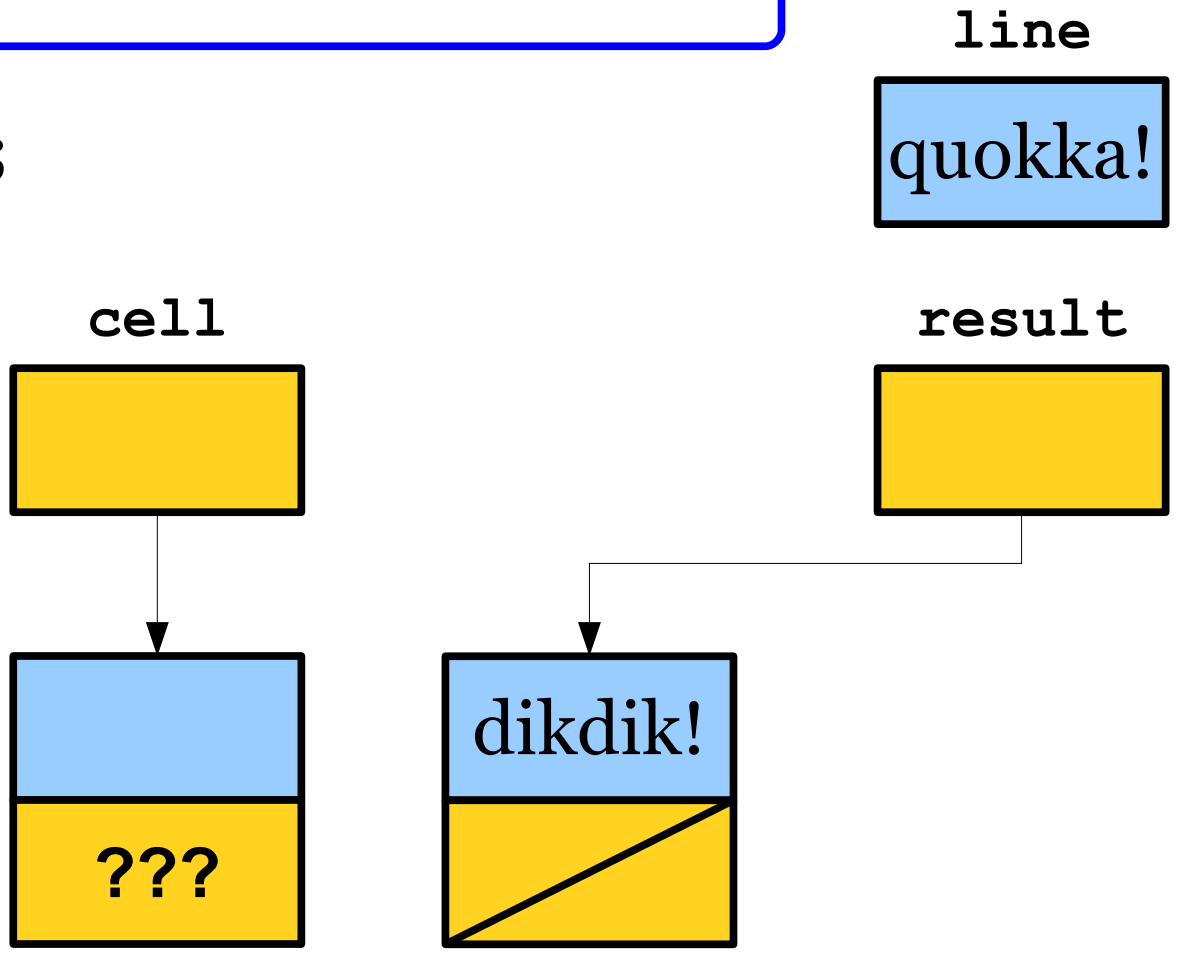
```
Cell* result = NULL;
while (true) {
   string line = getLine("Next entry? ");
   if (line == "") break;

   Cell* cell = new Cell;
   cell->value = line;

   cell->next = result;
   result = cell;
}
return result;
```



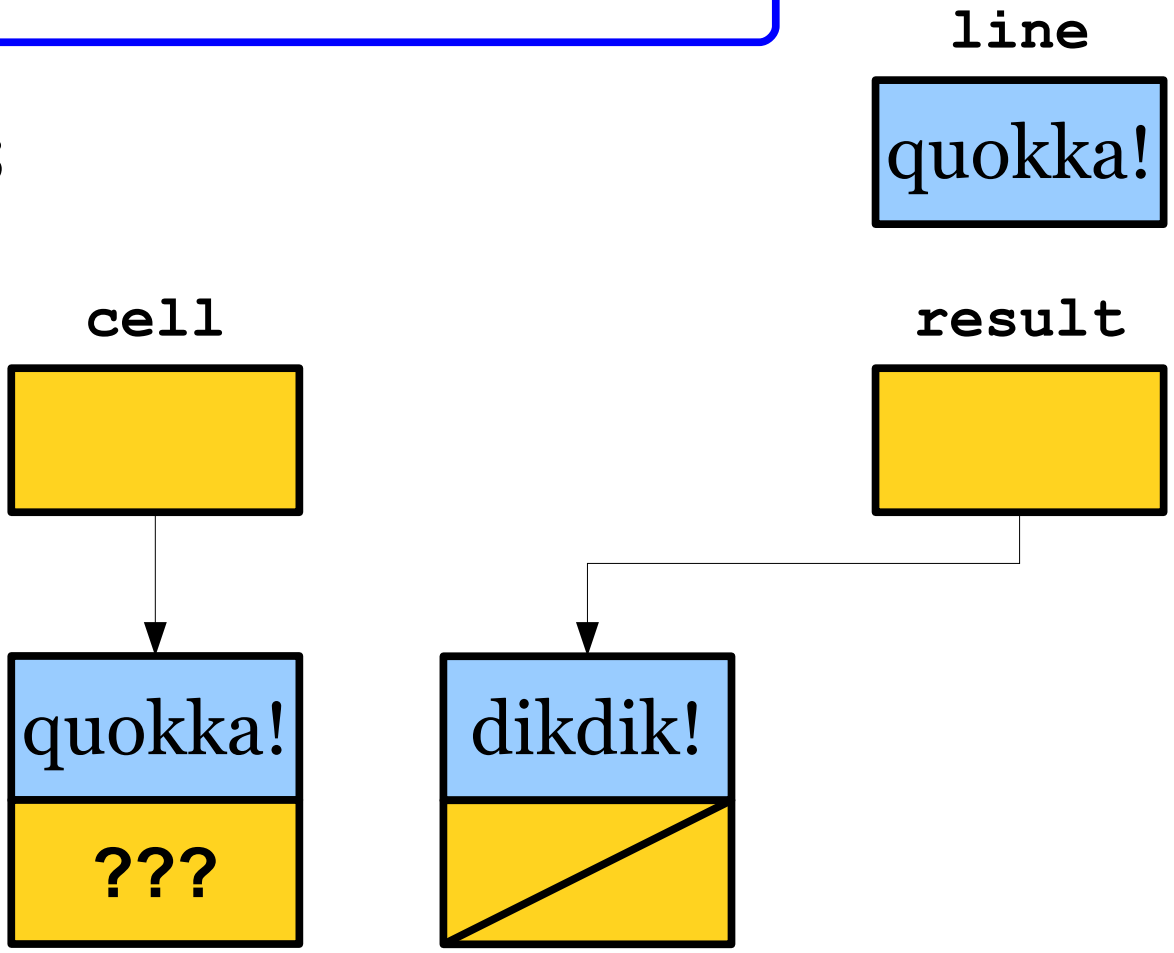
```
Cell* result = NULL;
while (true) {
   string line = getLine("Next entry? ");
   if (line == "") break;

   Cell* cell = new Cell;
   cell->value = line;

   cell->next = result;
   result = cell;
}
return result;
```



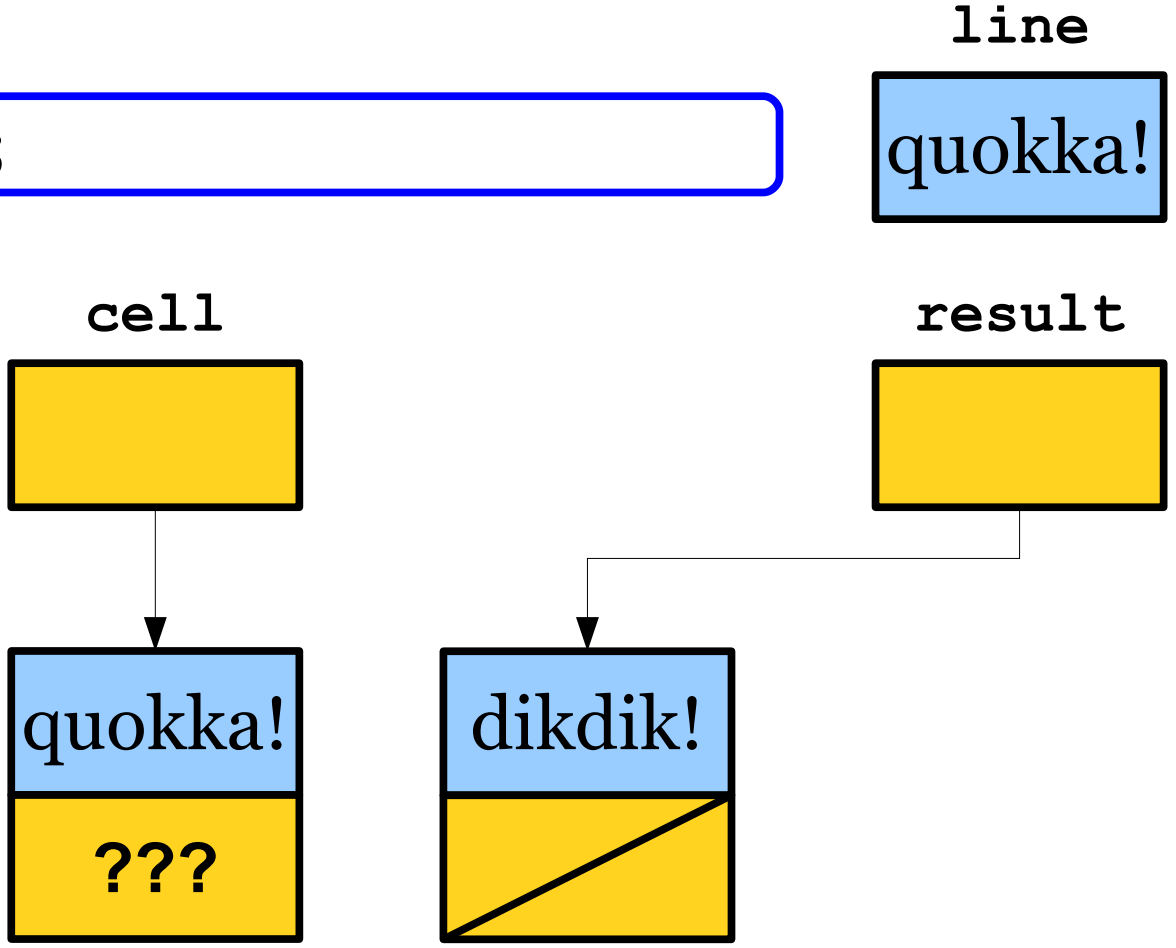
```
Cell* result = NULL;
while (true) {
   string line = getLine("Next entry? ");
   if (line == "") break;

   Cell* cell = new Cell;
   cell->value = line;

   cell->next = result;
   result = cell;
}
return result;
```



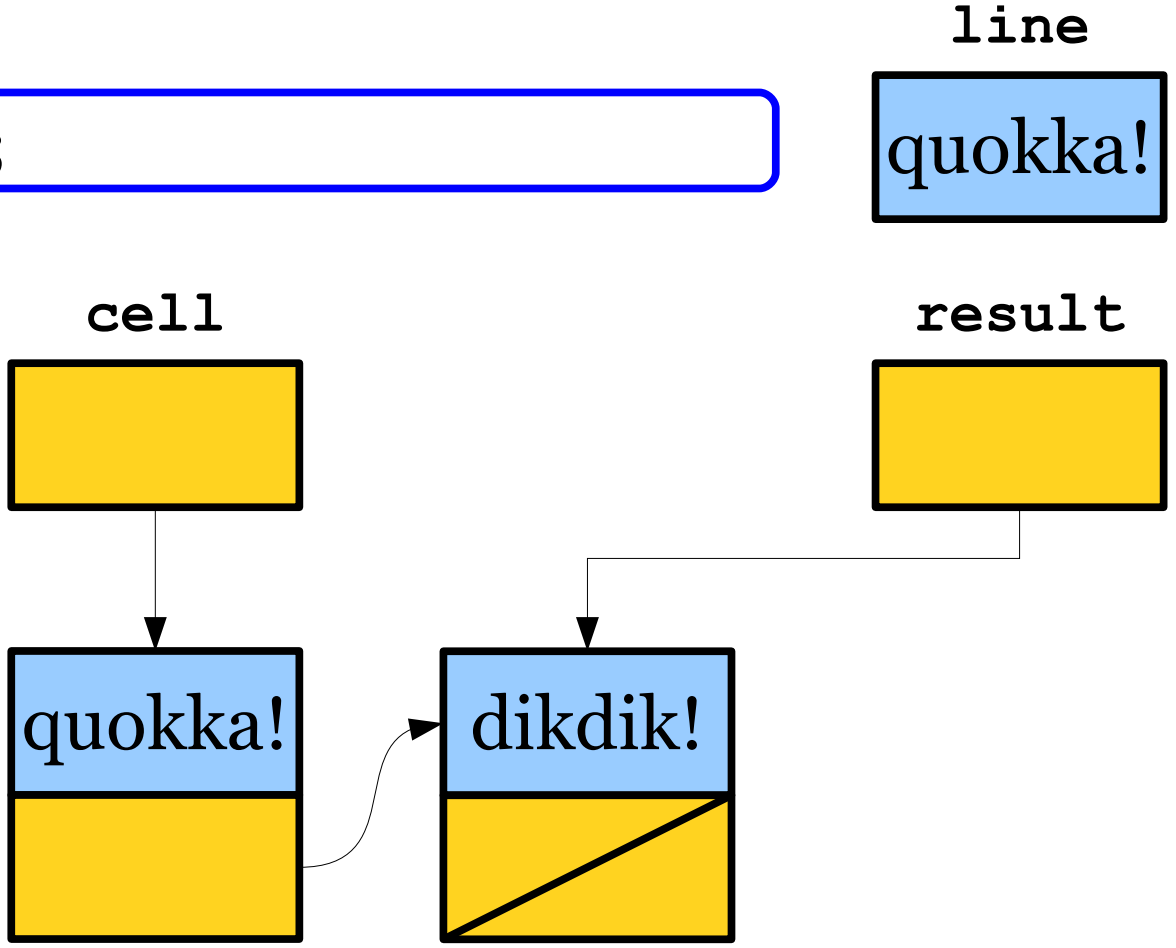
```
Cell* result = NULL;
while (true) {
   string line = getLine("Next entry? ");
   if (line == "") break;

   Cell* cell = new Cell;
   cell->value = line;

   cell->next = result;
   result = cell;
}
return result;
```



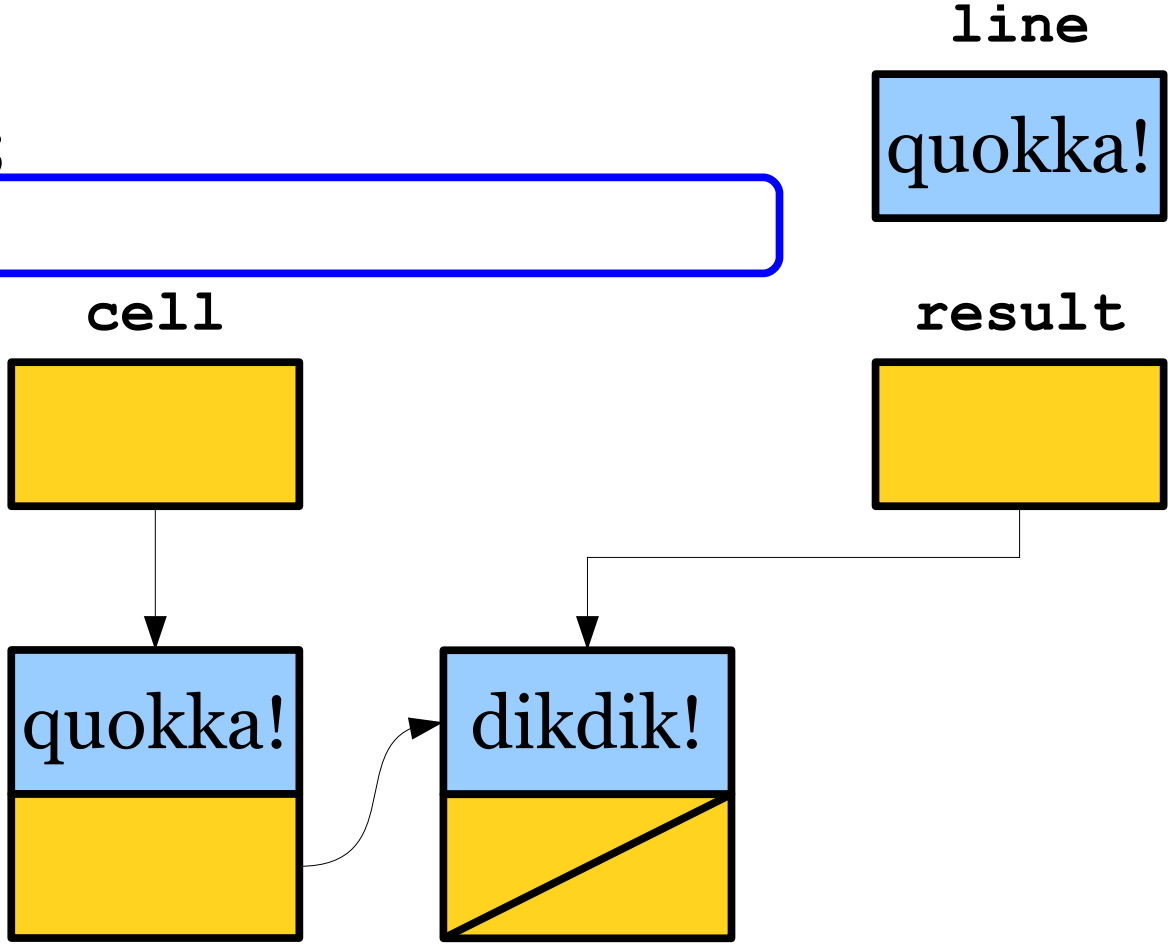
```
Cell* result = NULL;
while (true) {
   string line = getLine("Next entry? ");
   if (line == "") break;

   Cell* cell = new Cell;
   cell->value = line;

   cell->next = result;
   result = cell;
}
return result;
```



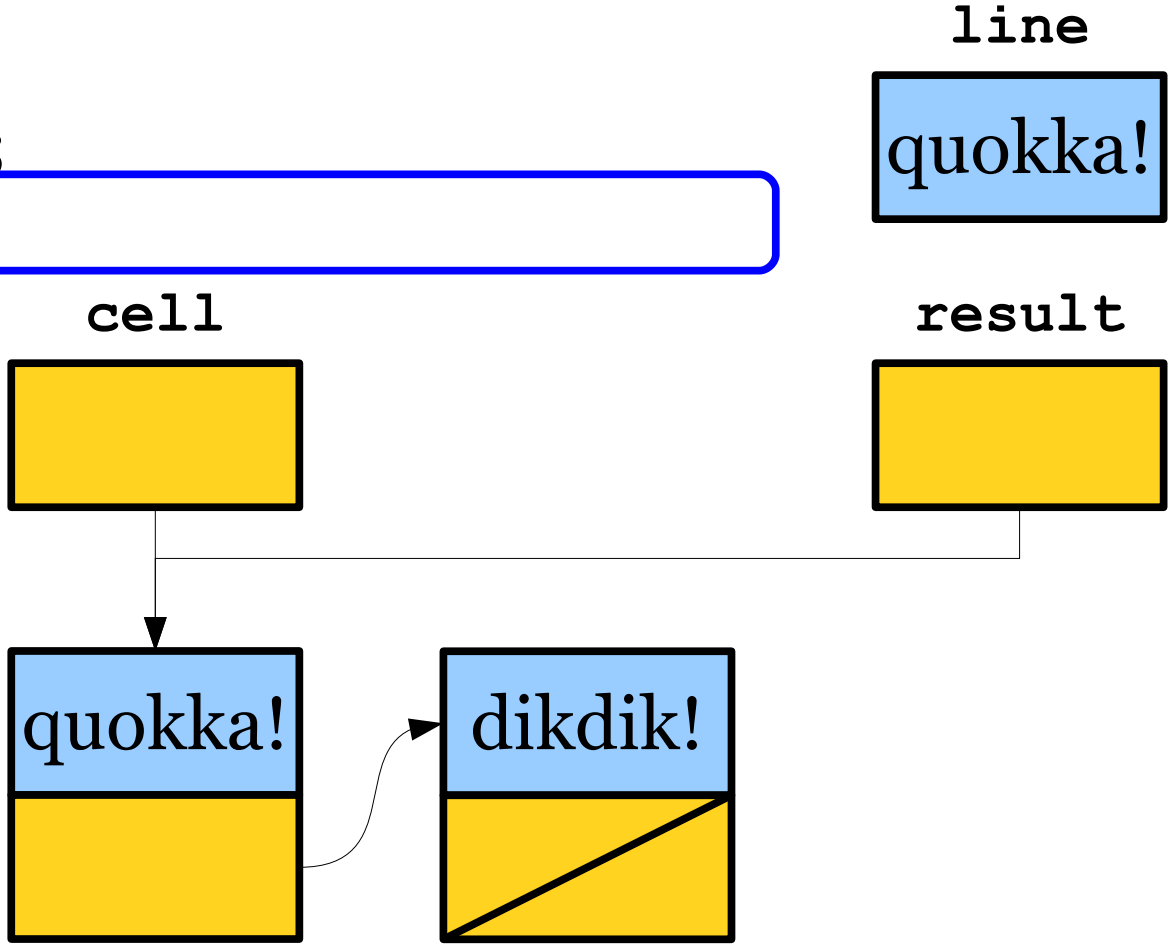
```
Cell* result = NULL;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```



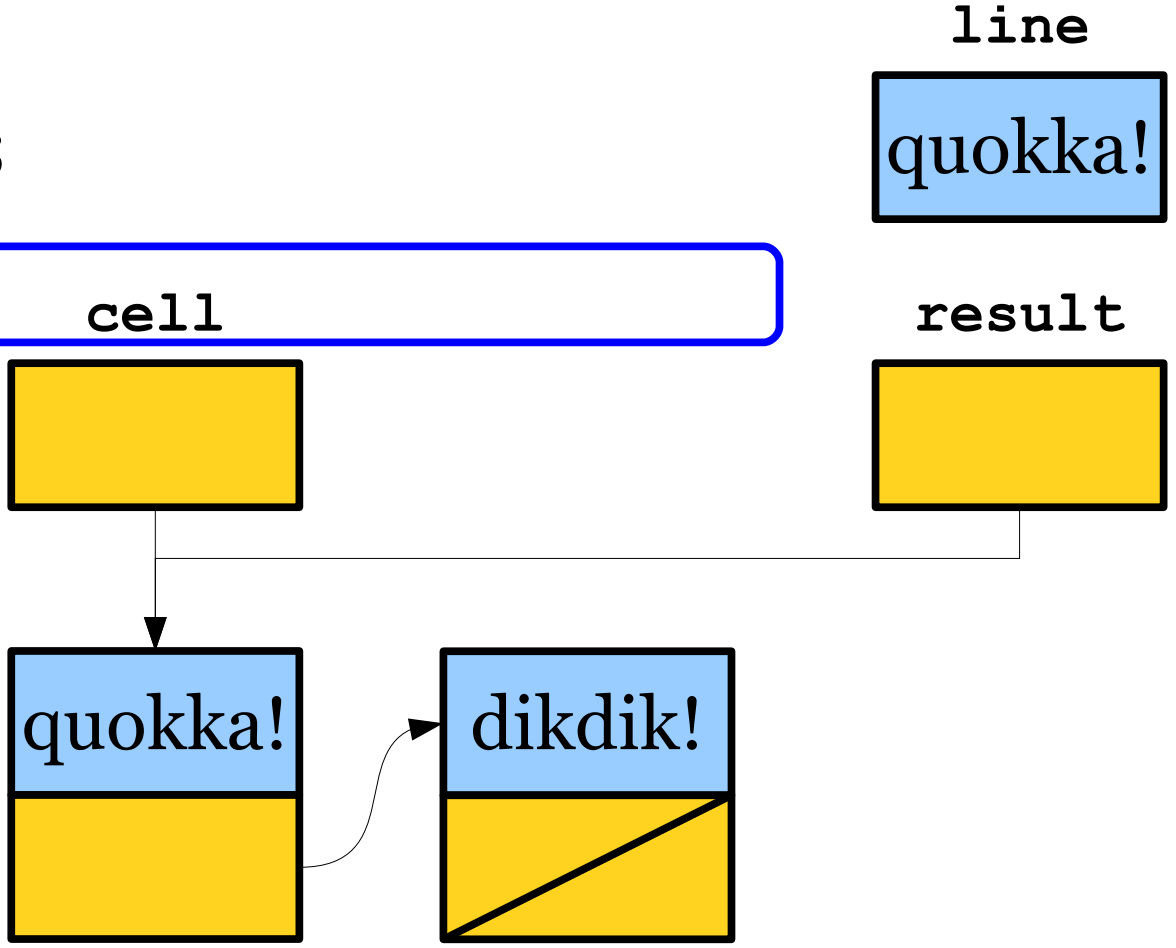
```
Cell* result = NULL;
while (true) {
   string line = getLine("Next entry? ");
   if (line == "") break;

   Cell* cell = new Cell;
   cell->value = line;

   cell->next = result;
   result = cell;
}
return result;
```



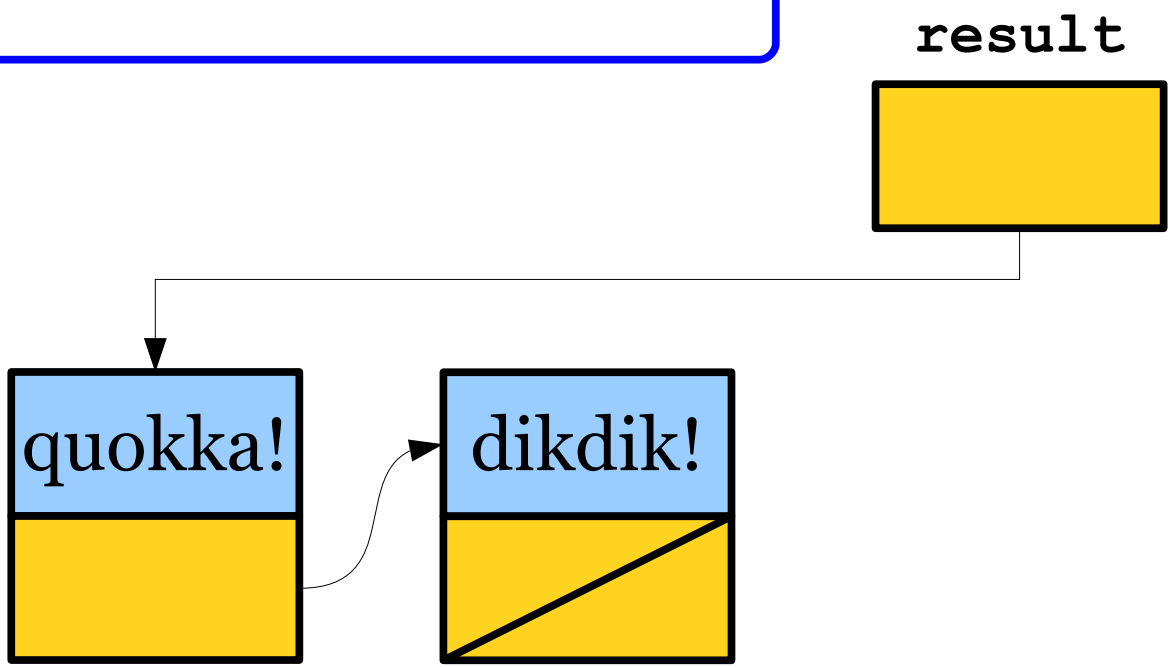
```
Cell* result = NULL;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```



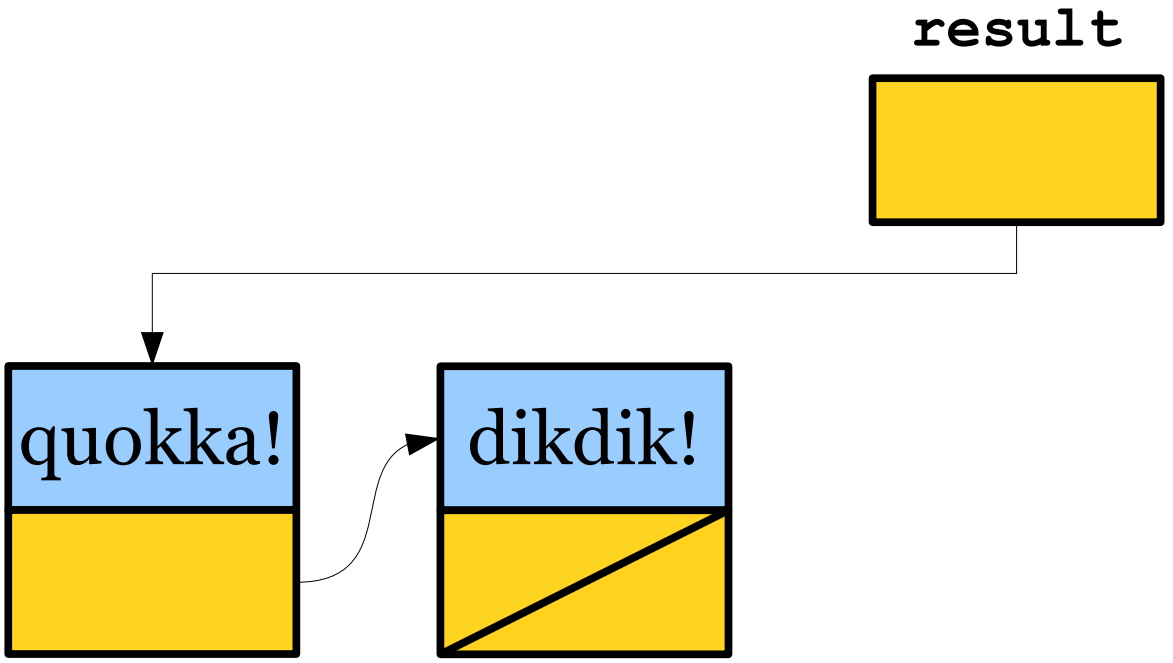
```
Cell* result = NULL;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```



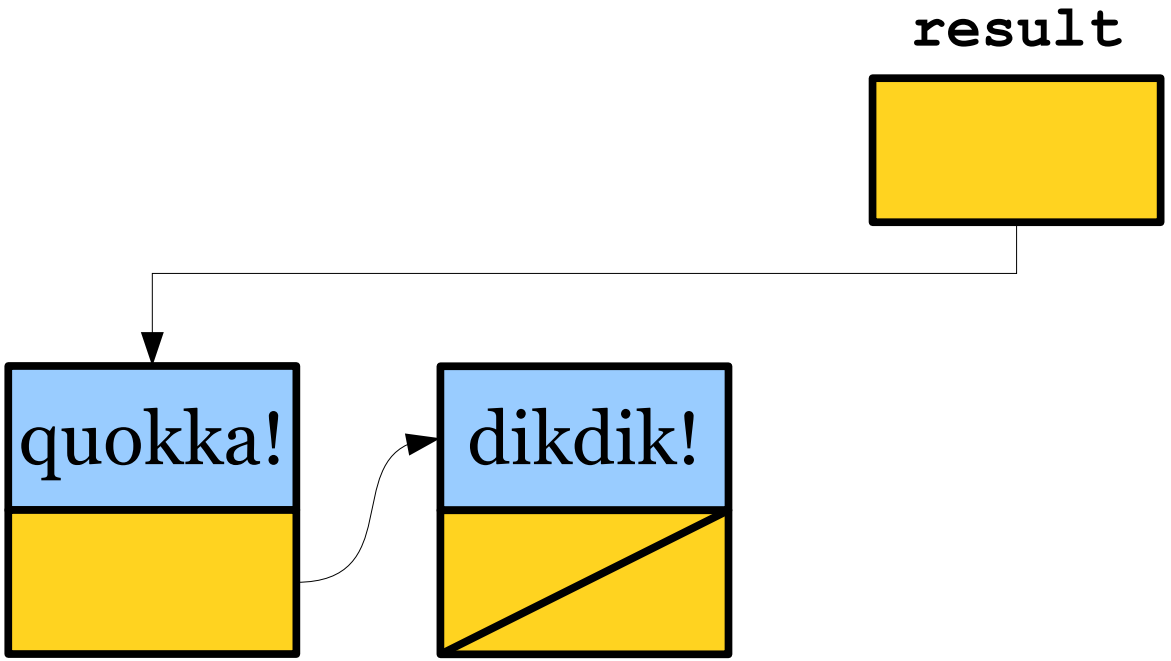
```
Cell* result = NULL;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```



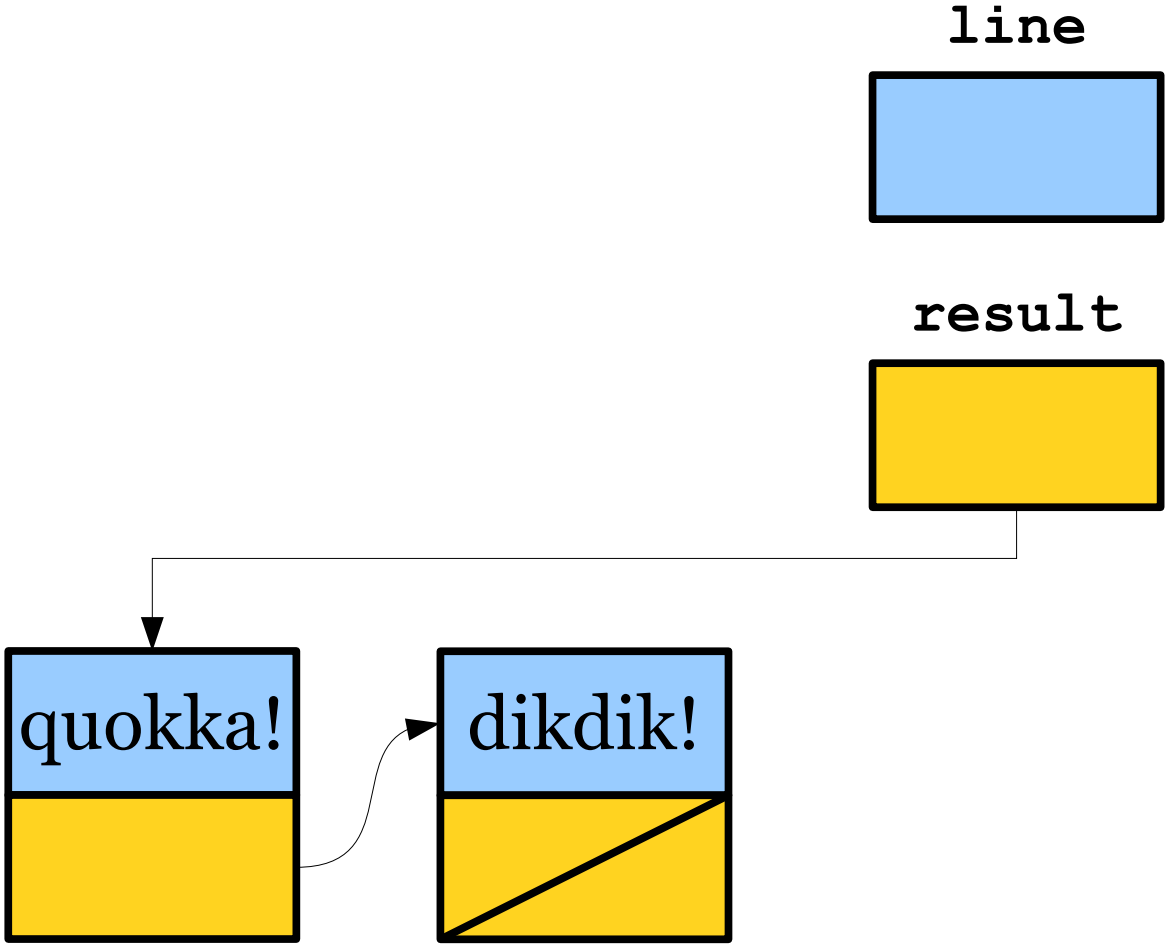
```
Cell* result = NULL;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```



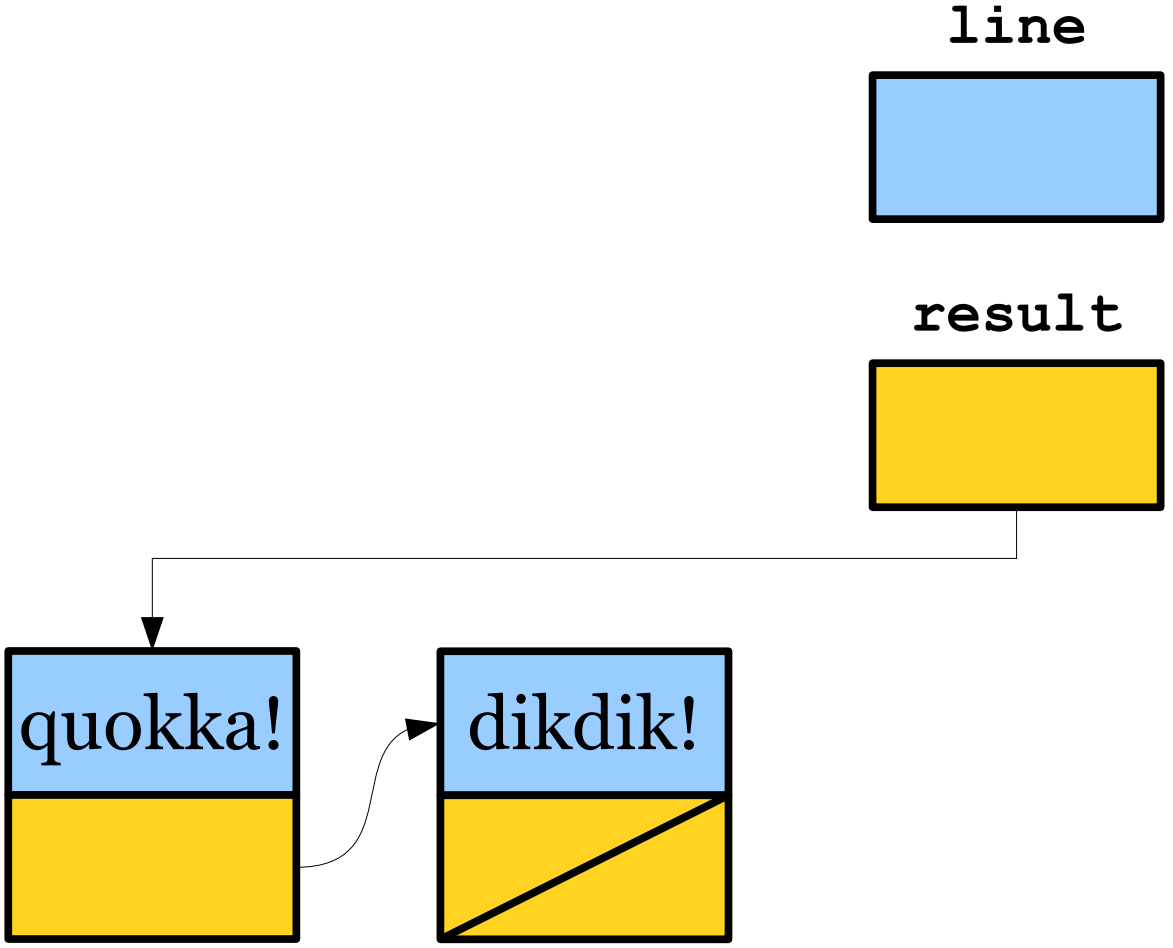
```
Cell* result = NULL;
while (true) {
   string line = getLine("Next entry? ");
   if (line == "") break;

   Cell* cell = new Cell;
   cell->value = line;

   cell->next = result;
   result = cell;
}
return result;
```



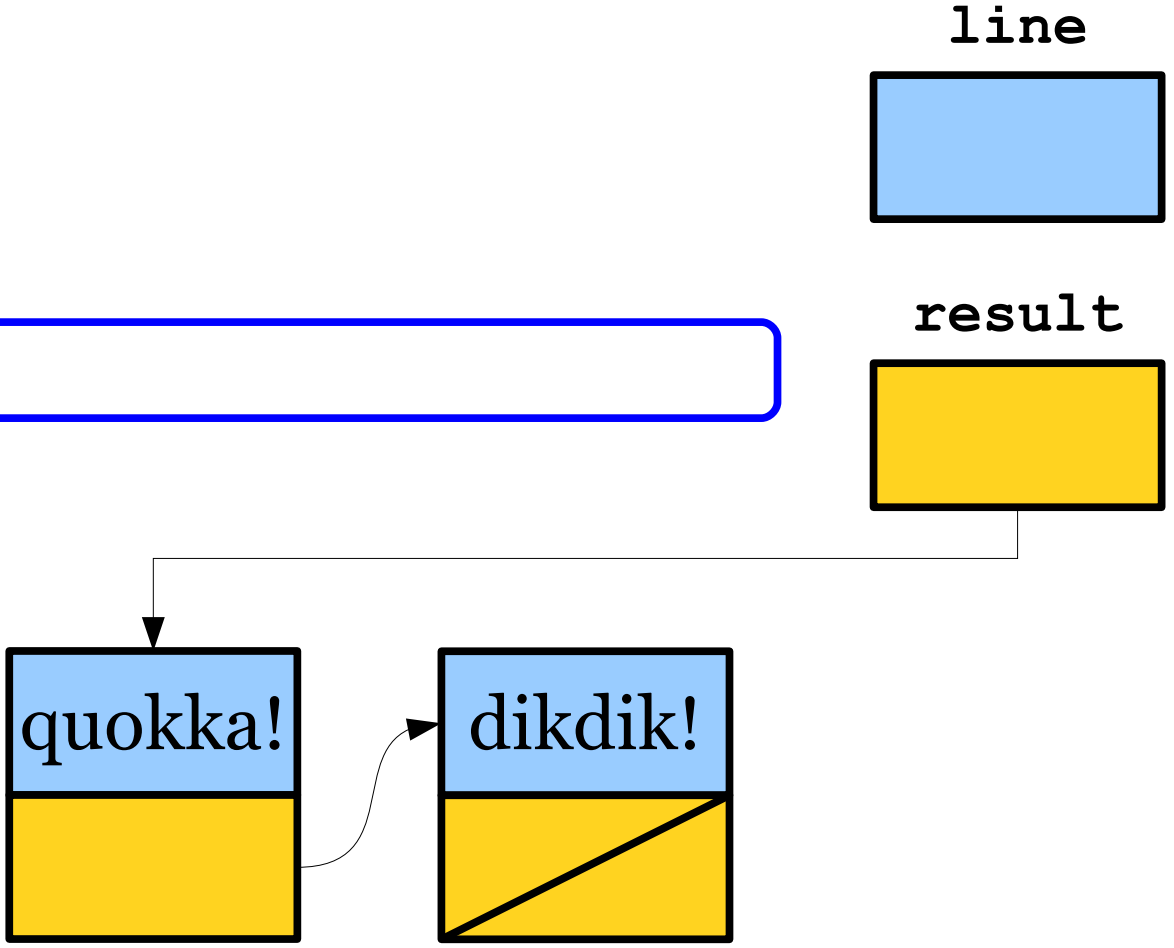
```
Cell* result = NULL;
while (true) {
   string line = getLine("Next entry? ");
   if (line == "") break;

   Cell* cell = new Cell;
   cell->value = line;

   cell->next = result;
   result = cell;
}
return result;
```



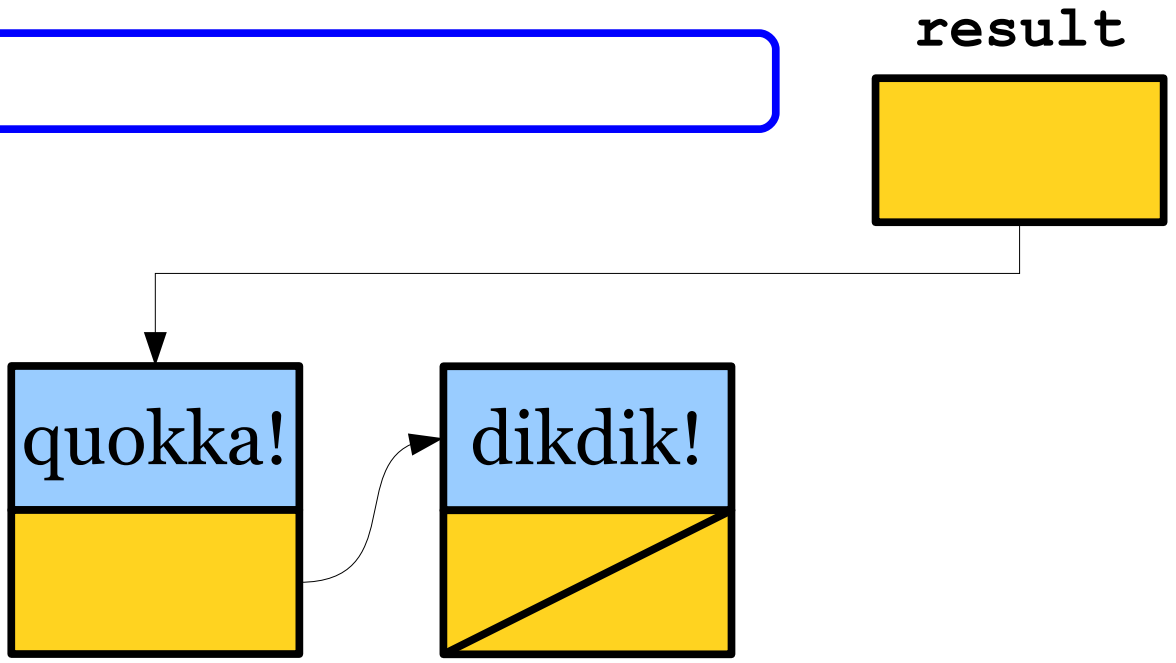
```
Cell* result = NULL;
while (true) {
   string line = getLine("Next entry? ");
   if (line == "") break;

   Cell* cell = new Cell;
   cell->value = line;

   cell->next = result;
   result = cell;
}
return result;
```



```
Cell* result = NULL;
while (true) {
   string line = getLine("Next entry? ");
   if (line == "") break;

   Cell* cell = new Cell;
   cell->value = line;

   cell->next = result;
   result = cell;
}
return result;
```



```
Cell* result = NULL;
while (true) {
   string line = getLine("Next entry? ");
   if (line == "") break;

   Cell* cell = new Cell;
   cell->value = line;

   cell->next = result;
   result = cell;
}
return result;
```

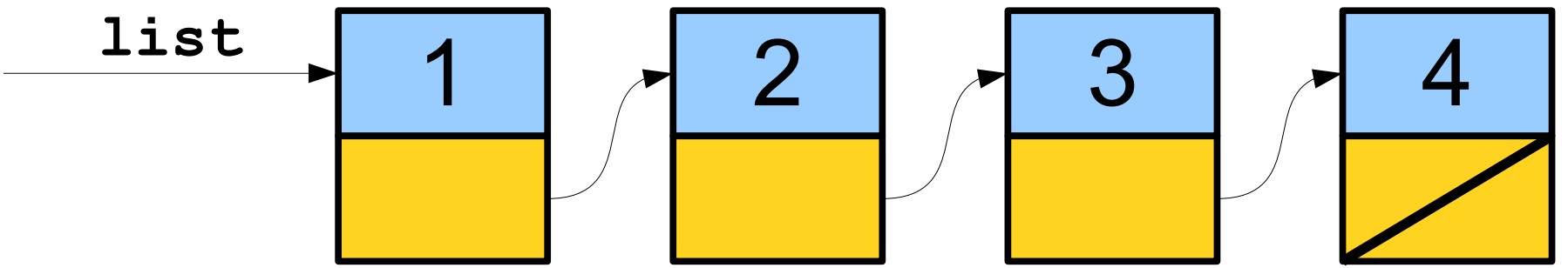


New Stuff

Traversing a Linked List

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

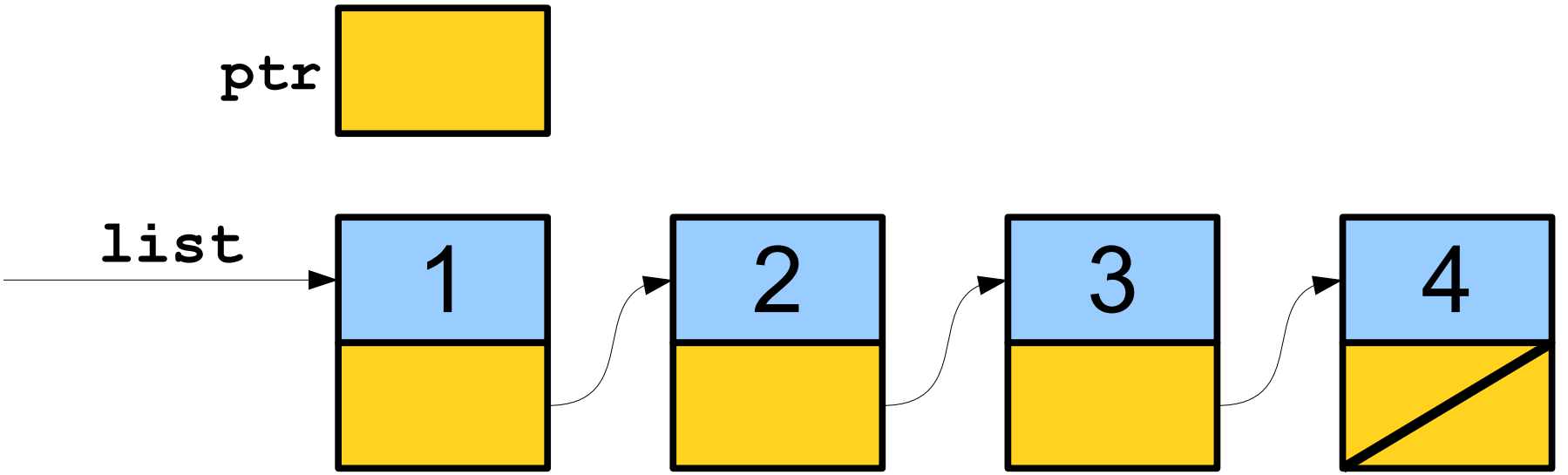
```
for (Cell* ptr = list; ptr != NULL; ptr = ptr->next) {
    /* … use ptr … */
}
```



Traversing a Linked List

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

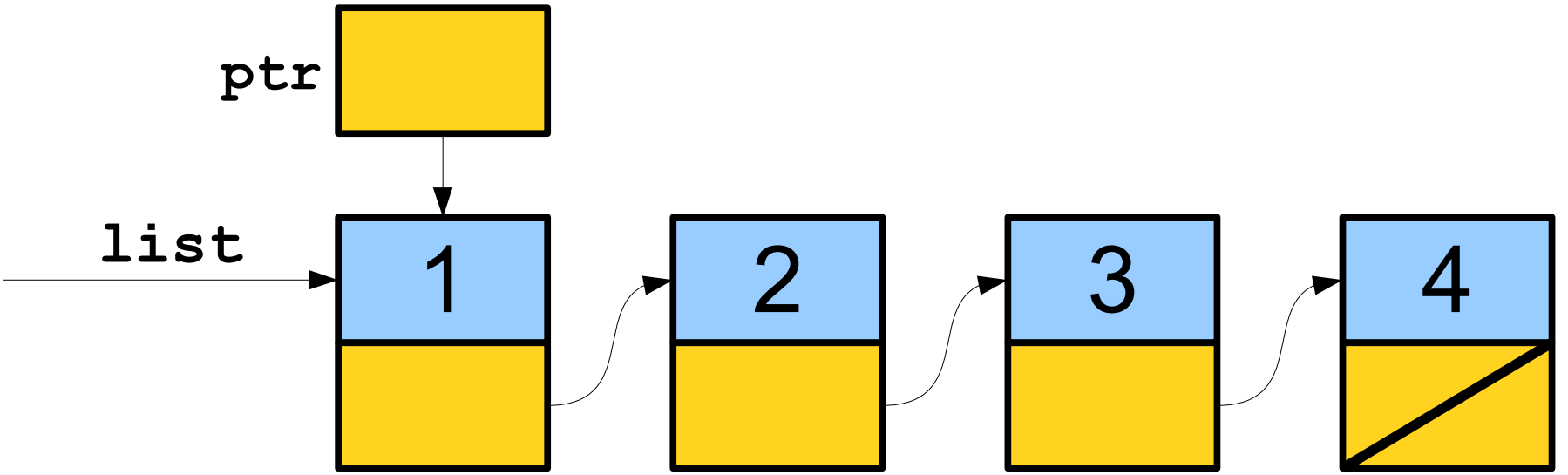
```
for (Cell* ptr = list; ptr != NULL; ptr = ptr->next) {
    /* … use ptr … */
}
```



Traversing a Linked List

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

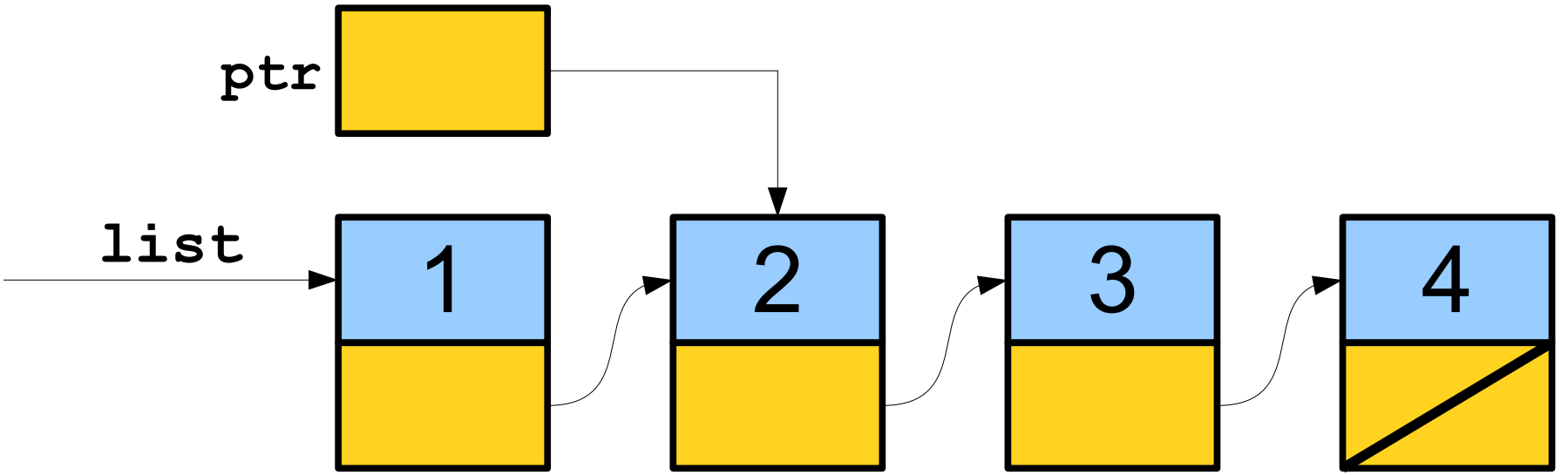
```
for (Cell* ptr = list; ptr != NULL; ptr = ptr->next) {
    /* … use ptr … */
}
```



Traversing a Linked List

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

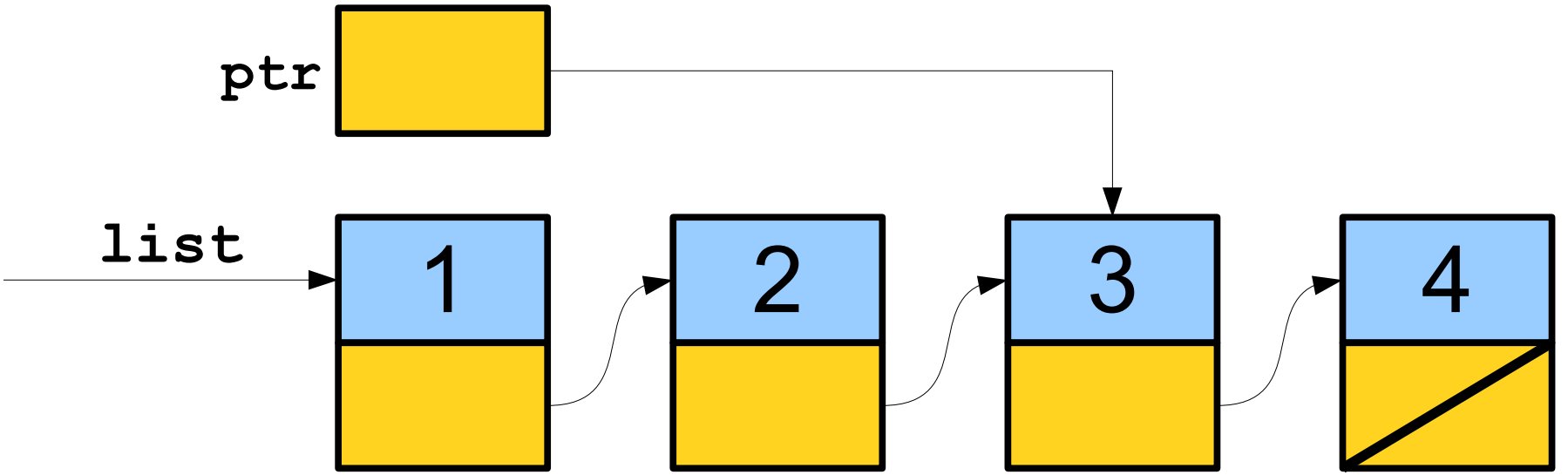
```
for (Cell* ptr = list; ptr != NULL; ptr = ptr->next) {
    /* … use ptr … */
}
```



Traversing a Linked List

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

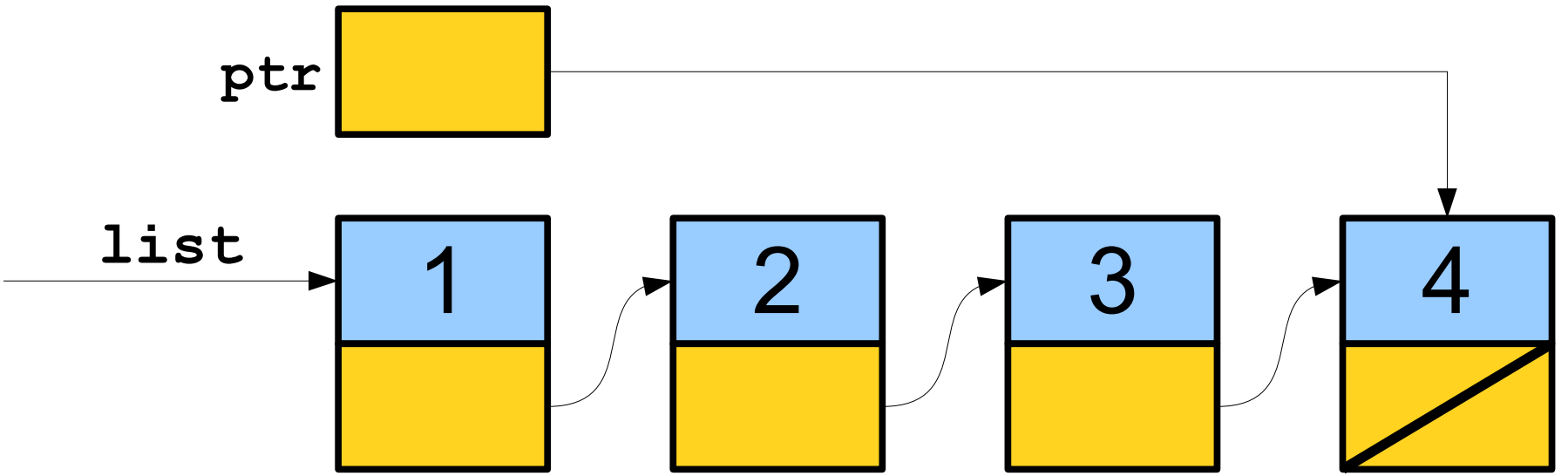
```
for (Cell* ptr = list; ptr != NULL; ptr = ptr->next) {
    /* … use ptr … */
}
```



Traversing a Linked List

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

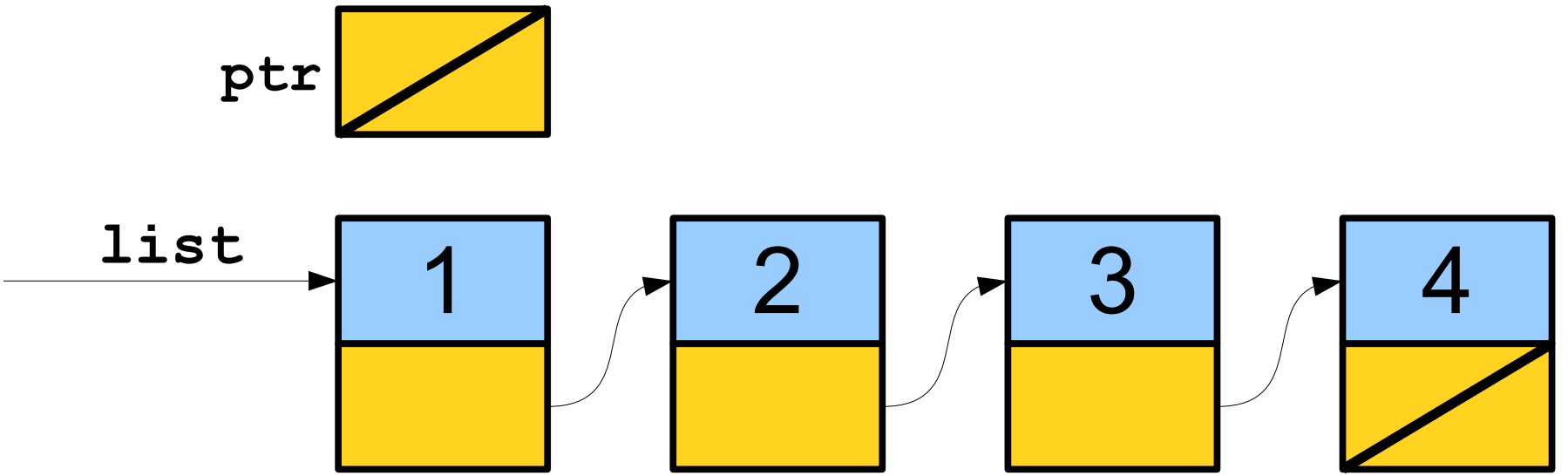
```
for (Cell* ptr = list; ptr != NULL; ptr = ptr->next) {
    /* … use ptr … */
}
```



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 … */
}
```



`printList()`
(Code)

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.
- Here's an ***Extremely Bad Idea***:

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

Freeing a Linked List

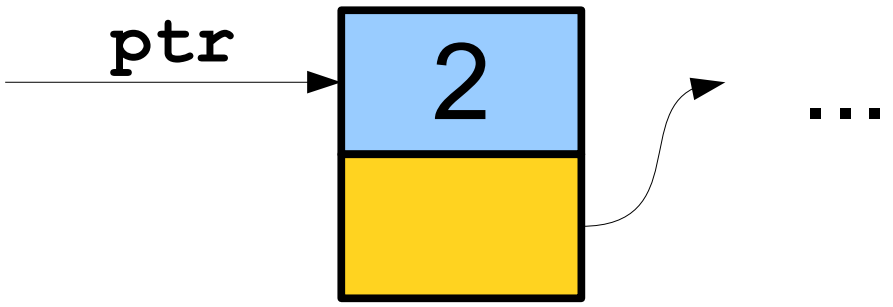
- All good things must come to an end, and we eventually need to reclaim the memory for a linked list.
- Here's an ***Extremely Bad Idea***:

```
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.
- Here's an ***Extremely Bad Idea***:

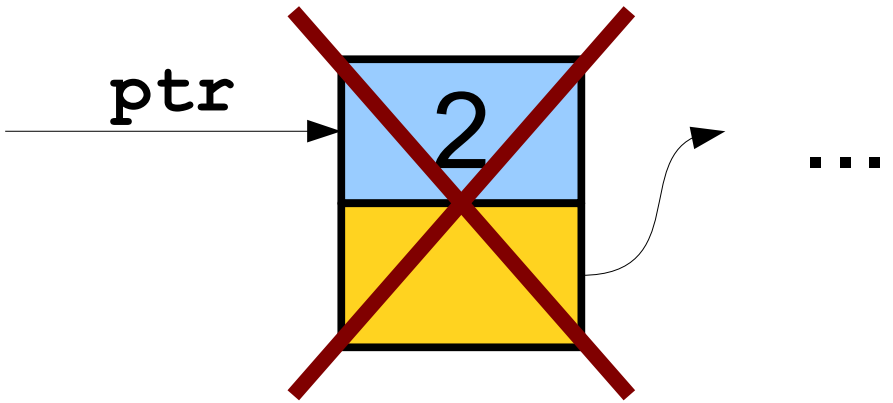
```
for (Cell* ptr = list; ptr != NULL; ptr = ptr->next) {
    delete ptr;
}
```



Freeing a Linked List

- All good things must come to an end, and we eventually need to reclaim the memory for a linked list.
- Here's 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.
- Here's 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?

`deleteList()`
(Code)

Linked Lists: The Tricky Parts

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

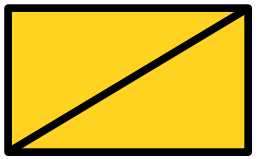
`listInsert()`
(Code)

What went wrong?

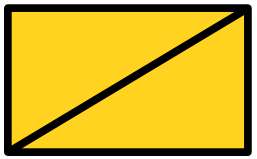
```
int main() {
    Cell* list = NULL;
    listInsert(list, 137);
    listInsert(list, 42);
    listInsert(list, 271);
}
```

```
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);
}
```

list 

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

list 

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

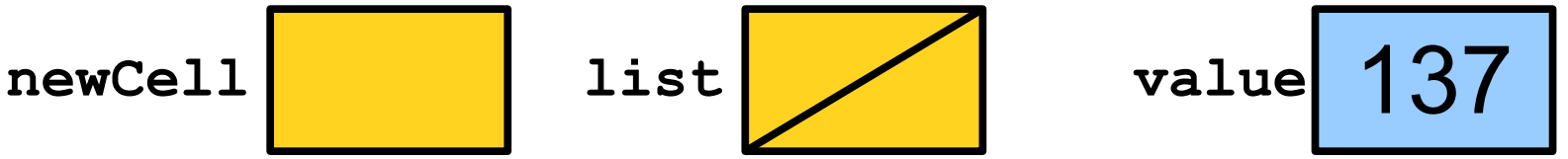
```
void listInsert(Cell* list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```

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

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

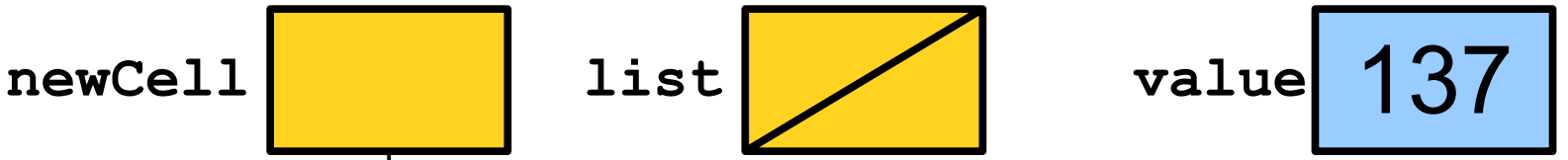
```
int main() {
}
```

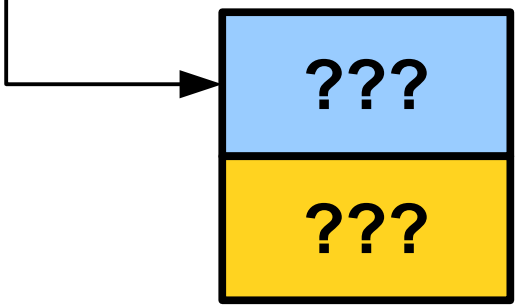
```
void listInsert(Cell* list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```



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

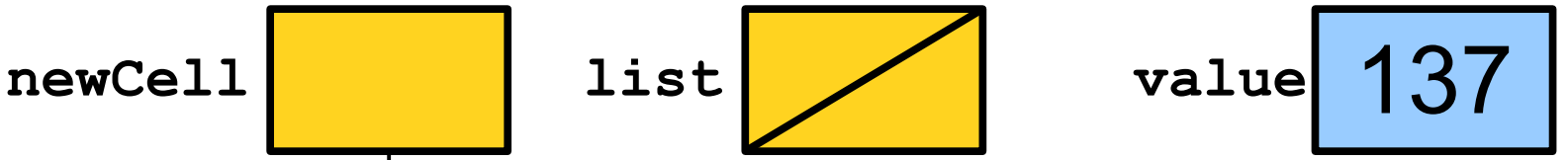
```
void listInsert(Cell* list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```

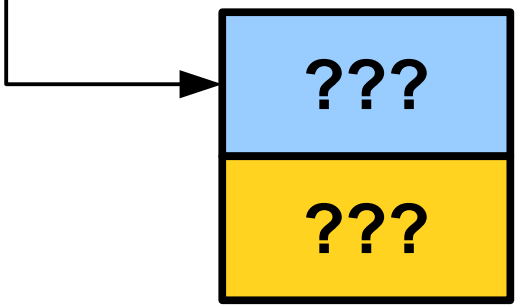




```
int main() {
}
```

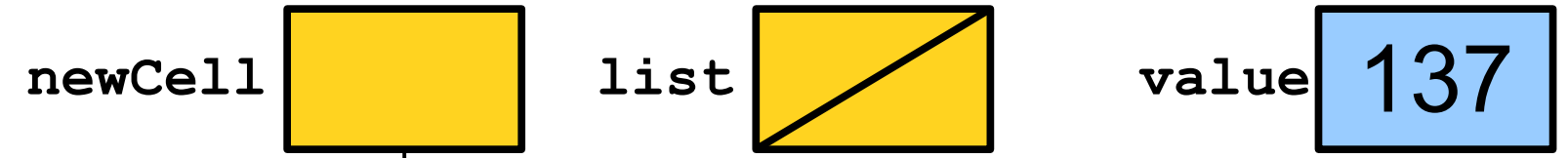
```
void listInsert(Cell* list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```

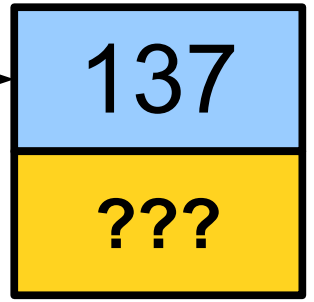




```
int main() {
}
```

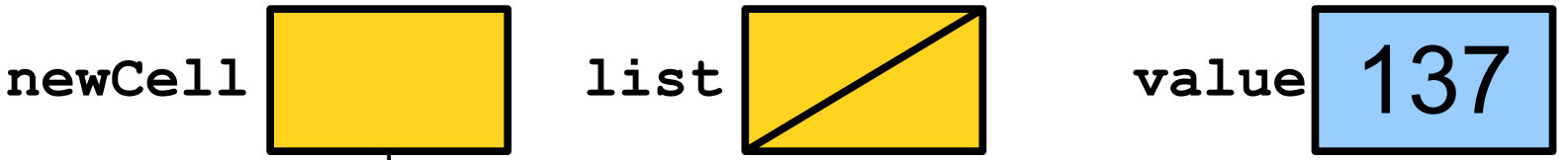
```
void listInsert(Cell* list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```

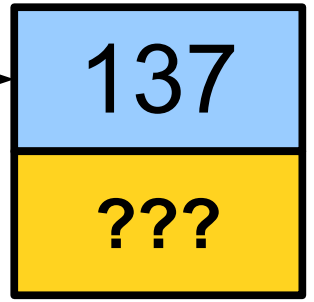




```
int main() {
}
```

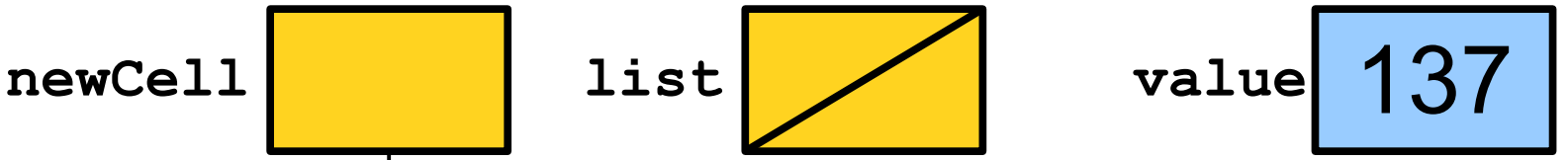
```
void listInsert(Cell* list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```

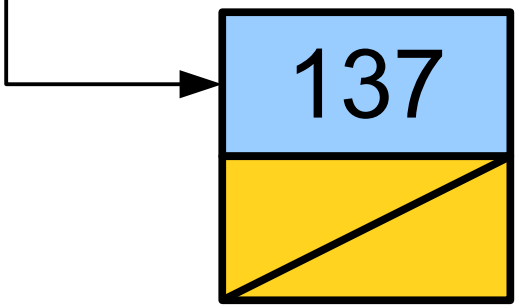




```
int main() {
}
```

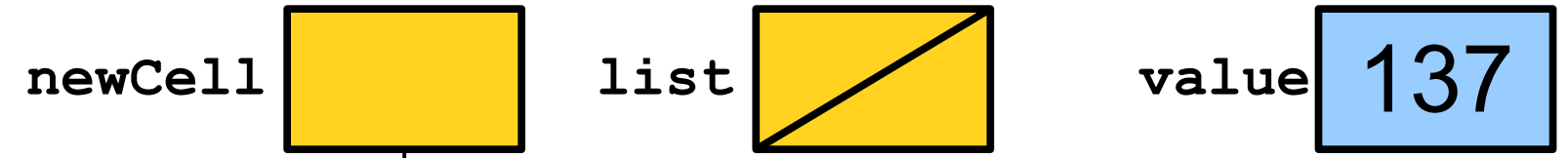
```
void listInsert(Cell* list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```

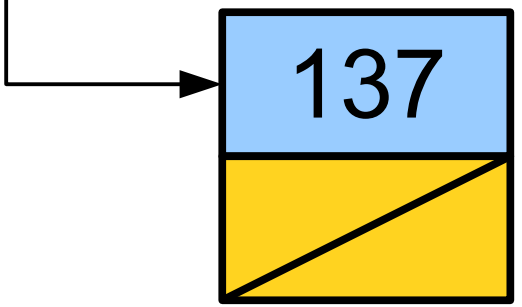




```
int main() {
}
```

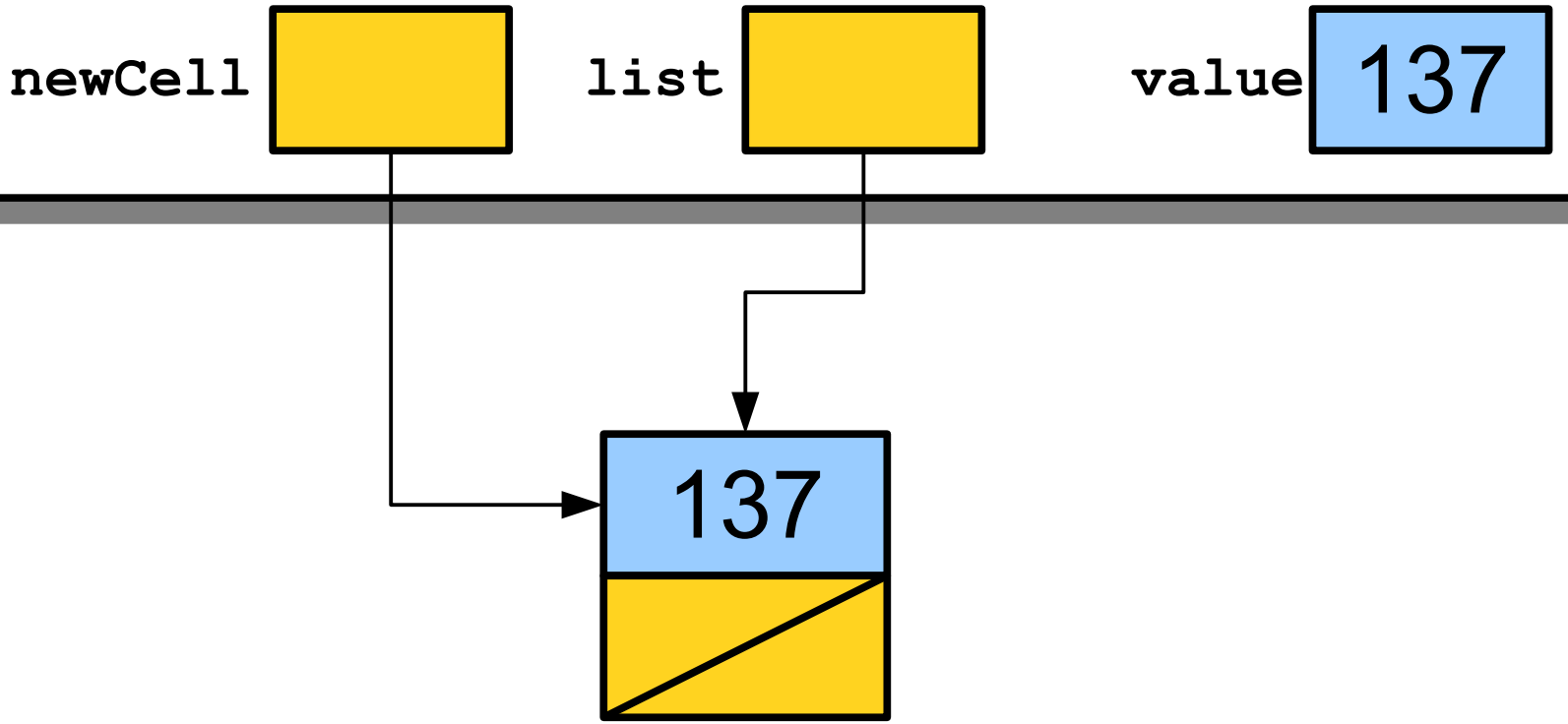
```
void listInsert(Cell* list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```



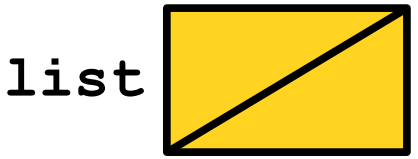


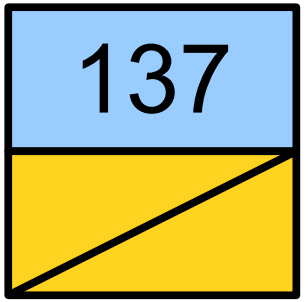
```
int main() {
}
```

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

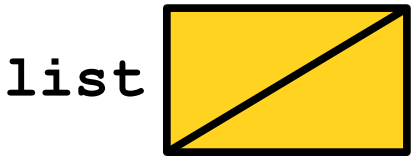


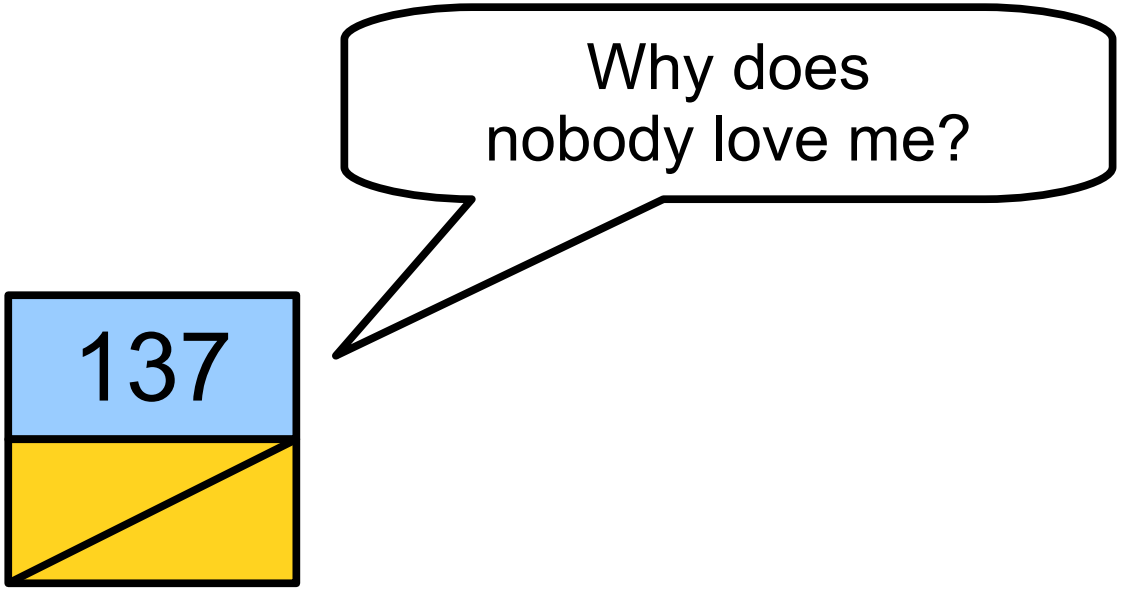
```
int main() {
    Cell* list = NULL;
    listInsert(list, 137);
    listInsert(list, 42);
    listInsert(list, 271);
}
```





```
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;
}
```

Pointers by Reference

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

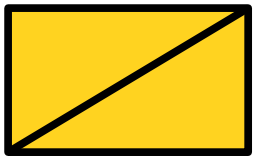
```
void listInsert(Cell*& list, int value) {
    Cell* newCell = new Cell;
    cell->value = value
    cell->next = list;
    list = cell;
}
```

This is a **reference to a pointer to a Cell**. It's often useful to read this from the right to the left.

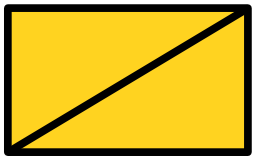
```
int main() {
    Cell* list = NULL;
    listInsert(list, 137);
    listInsert(list, 42);
    listInsert(list, 271);
}
```

```
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);
}
```

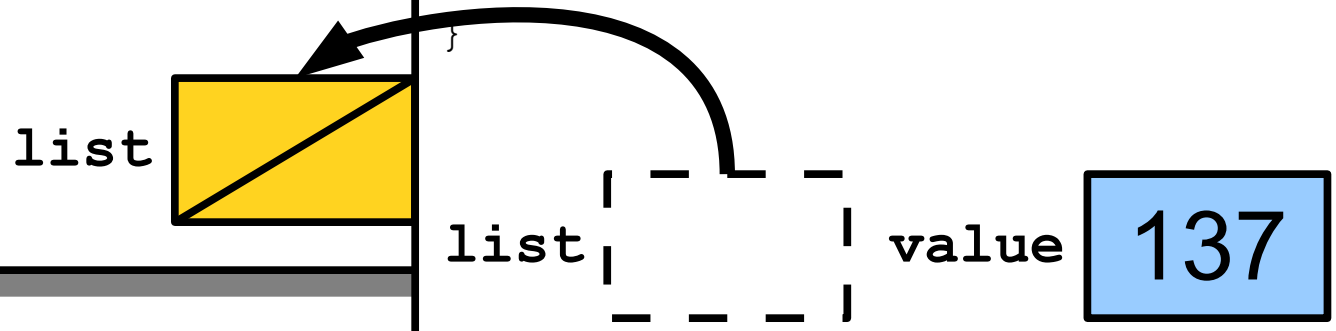
list 

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

list 

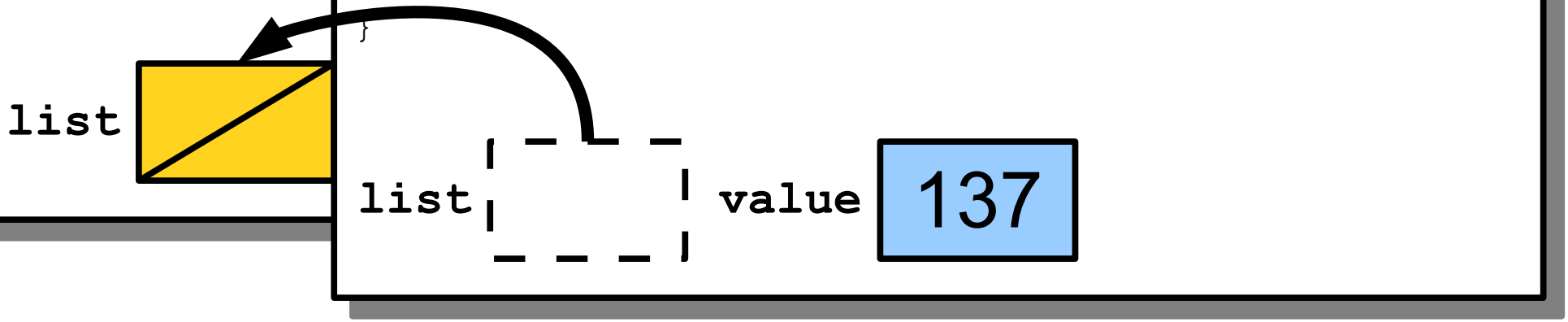
```
int main() {
    Cell* list = N
    listInsert(lis
    listInsert(lis
    listInsert(lis
}
```

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



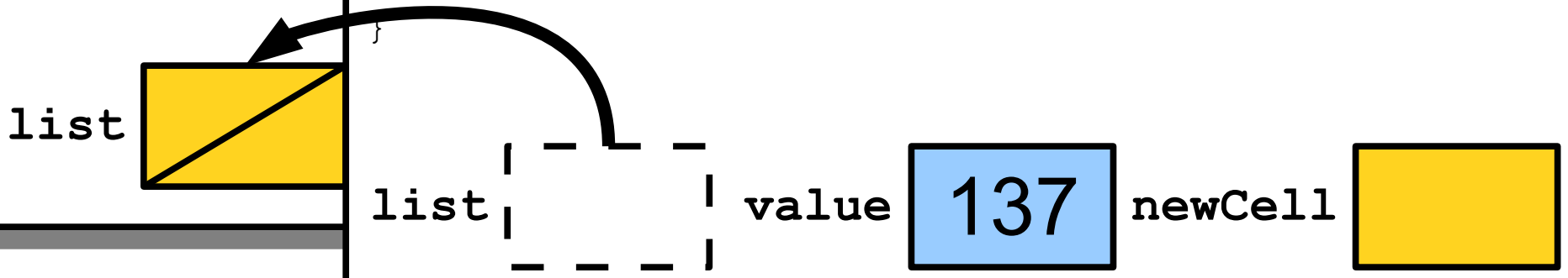
```
int main() {
    Cell* list = N
    listInsert(lis
    listInsert(lis
    listInsert(lis
}
```

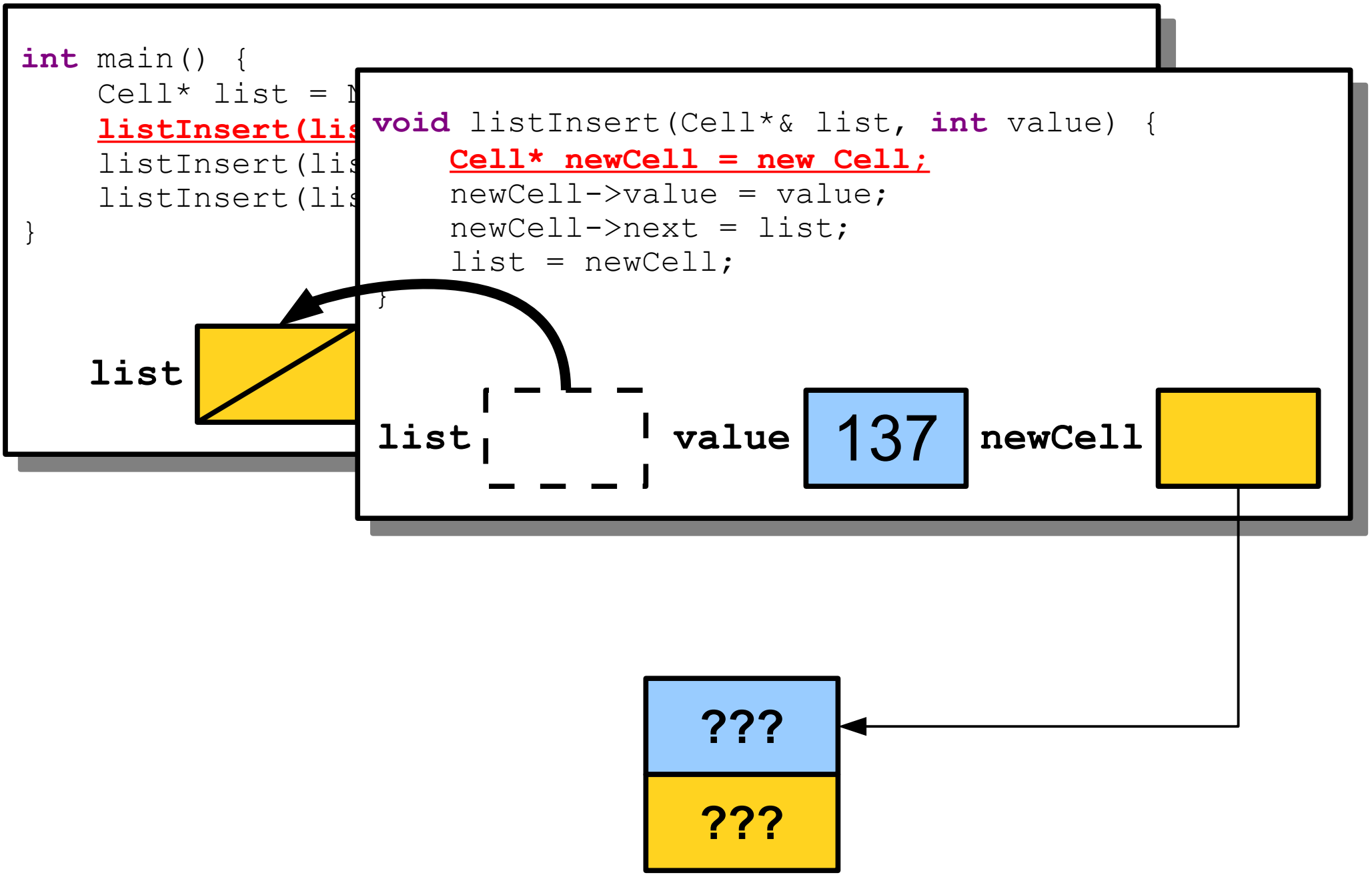
```
void listInsert(Cell*& list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```

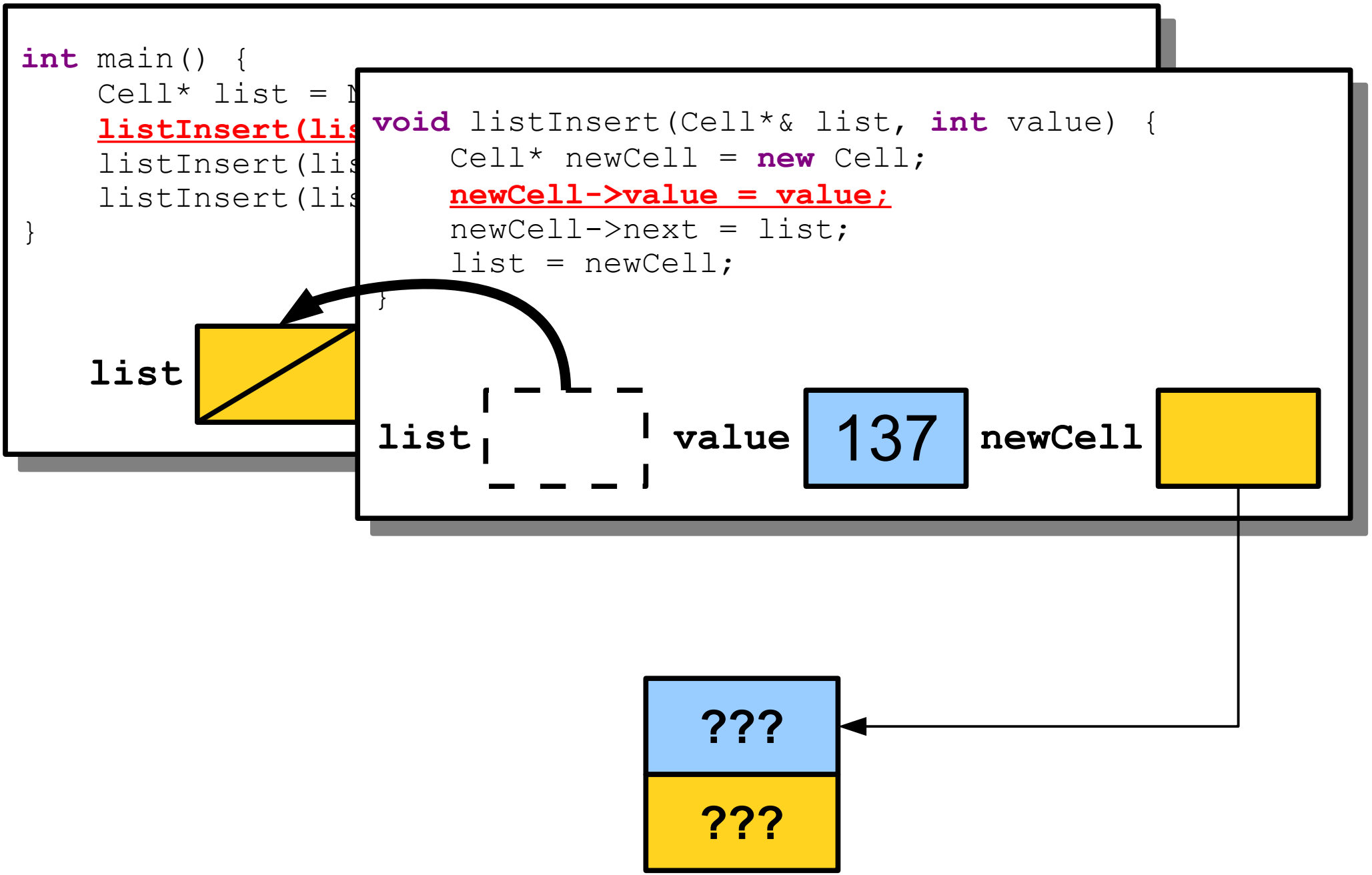


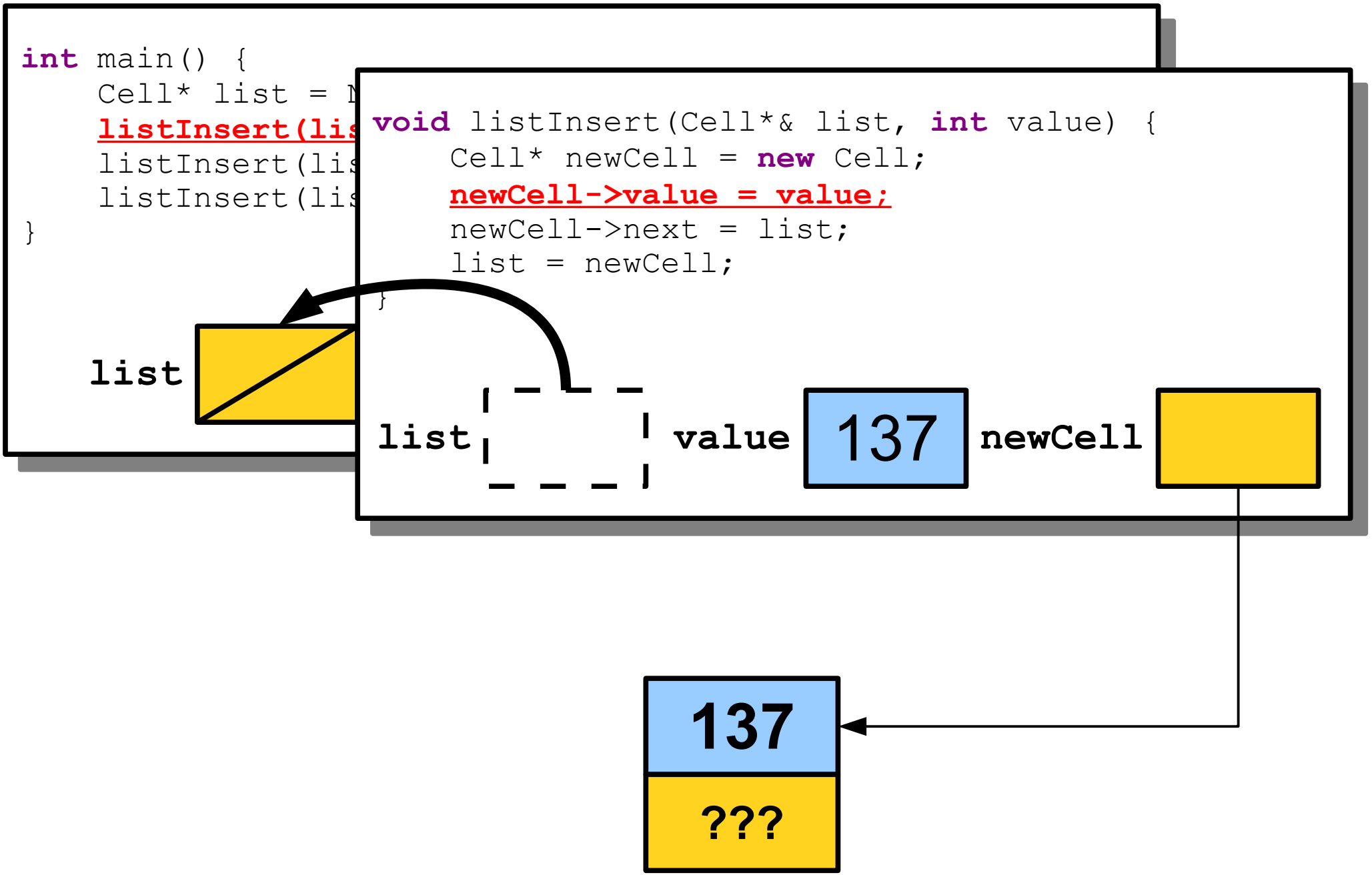
```
int main() {
    Cell* list = N
    listInsert(lis
    listInsert(lis
    listInsert(lis
}
```

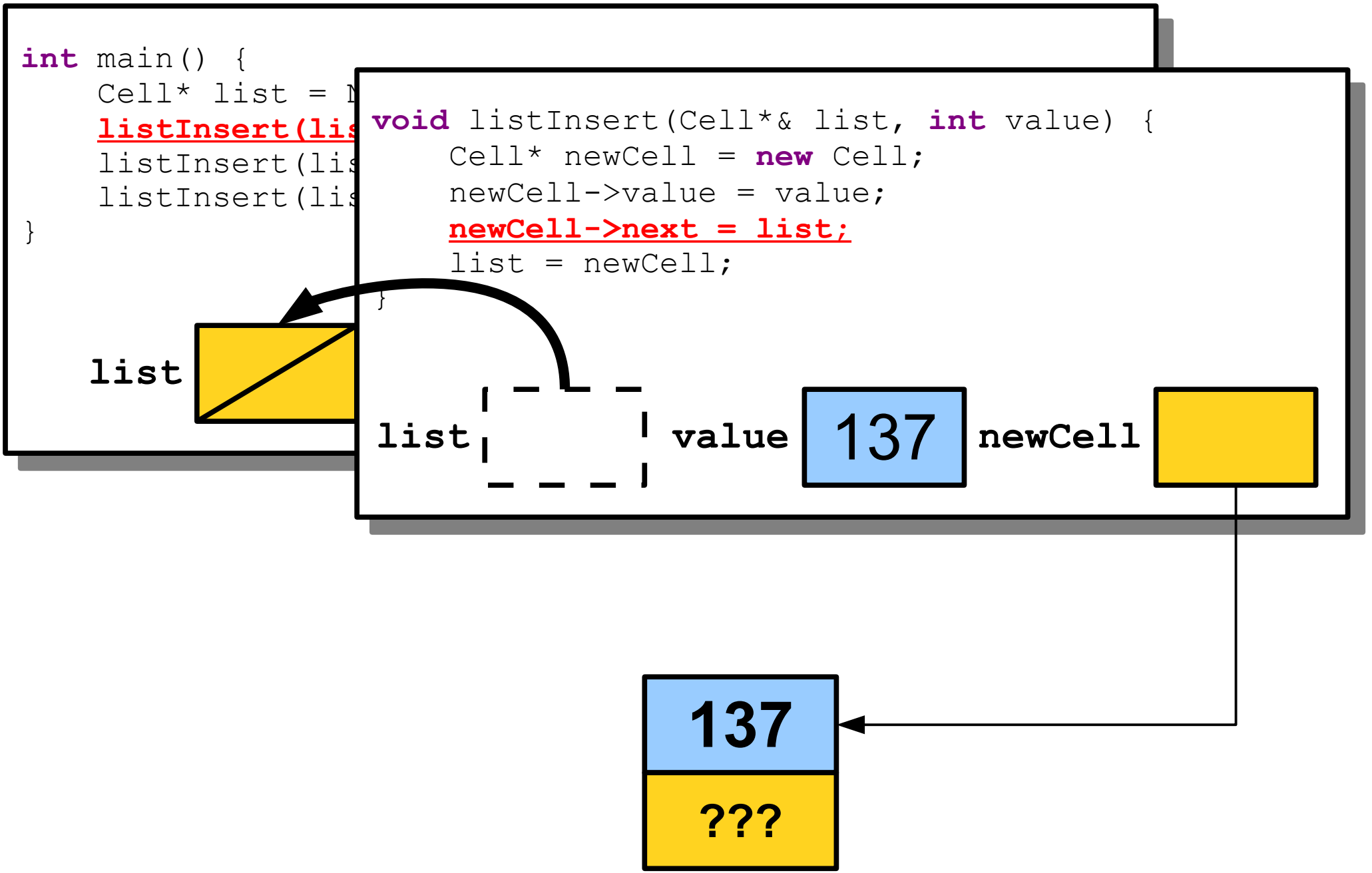
```
void listInsert(Cell*& list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```




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;
}
list
list
value
137
newCell
???
???

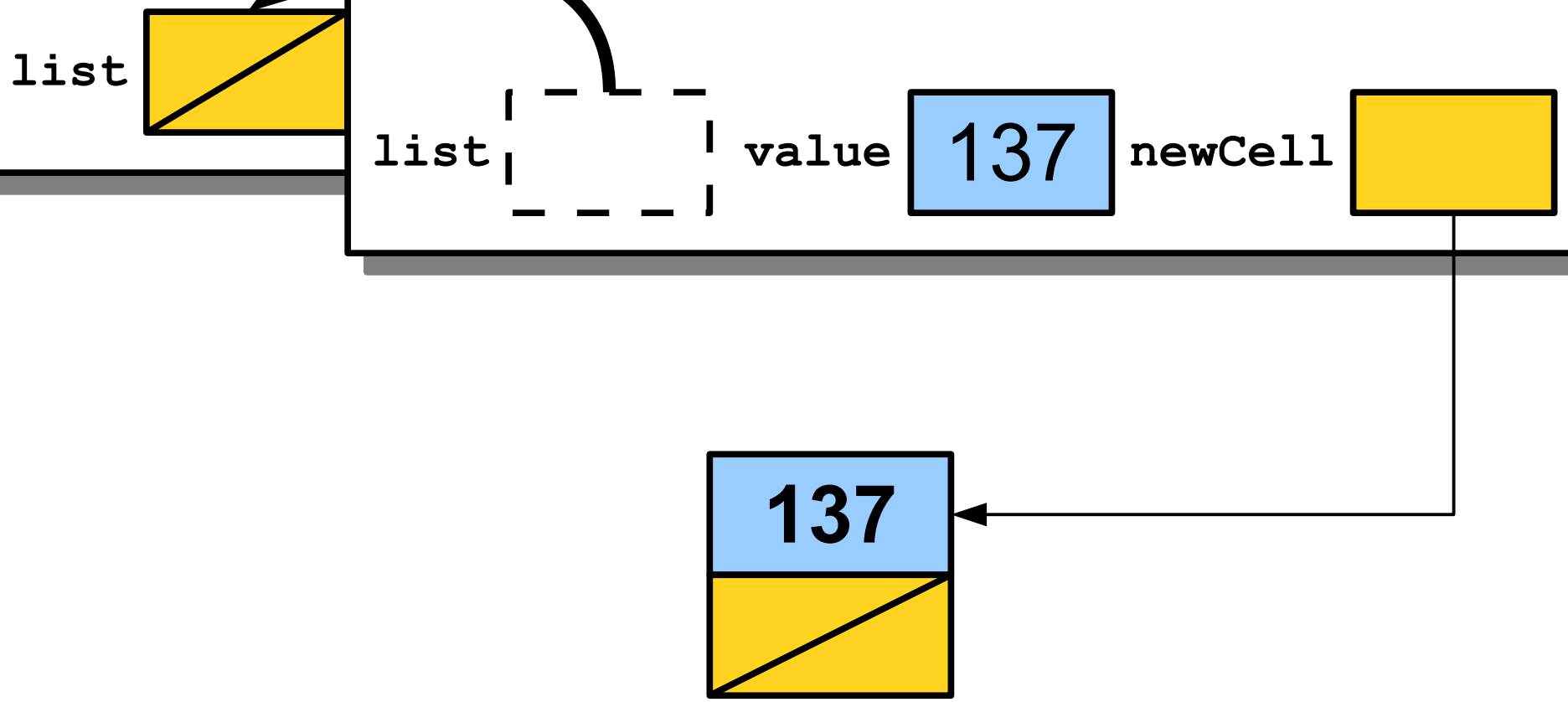

int main() {
Cell* list = N
listInsert(lis
listInsert(lis
listInsert(lis
}
void listInsert(Cell*& list, int value) {
Cell* newCell = new Cell;
newCell->value = value;
newCell->next = list;
list = newCell;
}
list
list
value
137
newCell
???
???

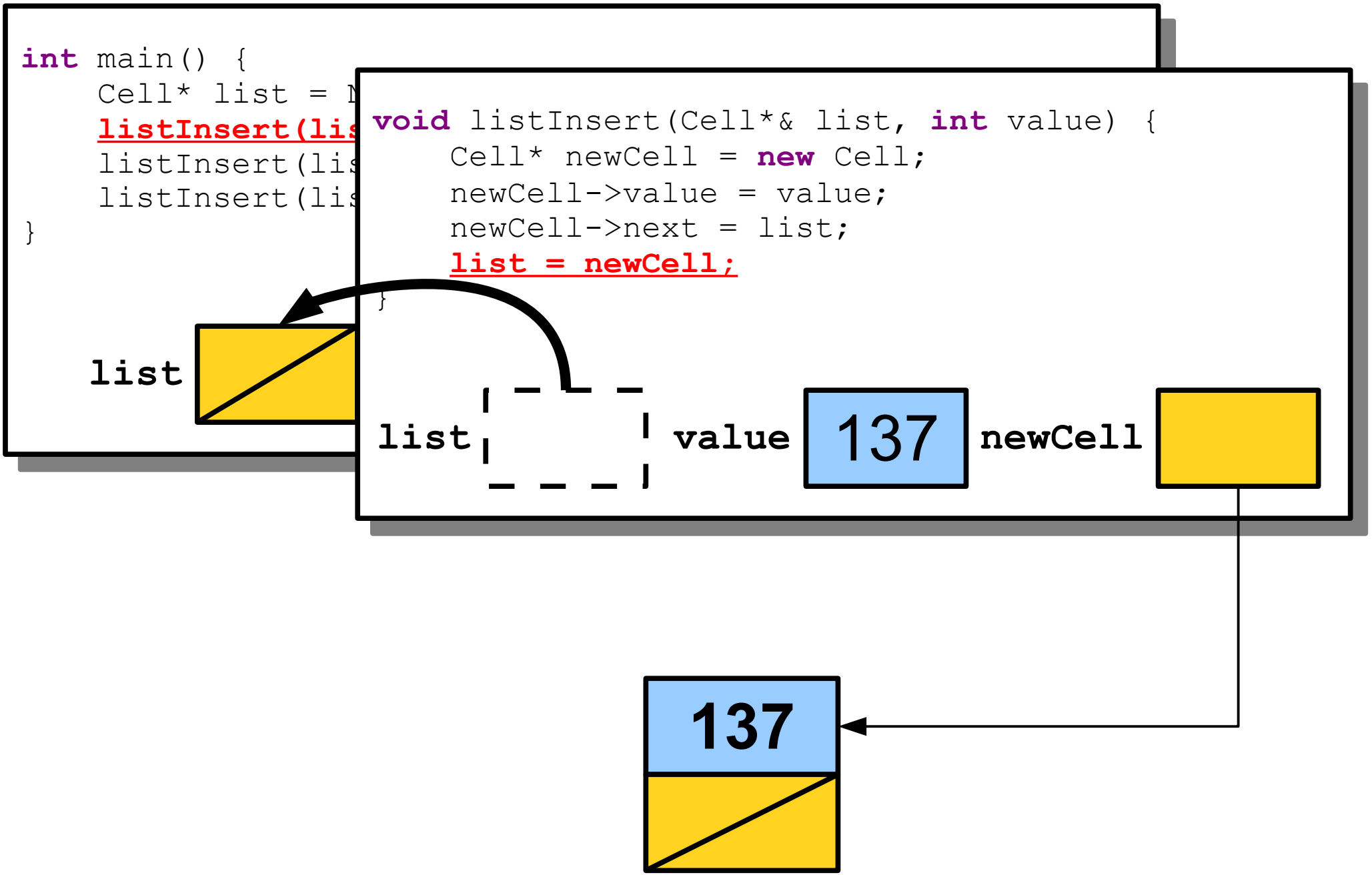

int main() {
Cell* list = N
listInsert(lis
listInsert(lis
listInsert(lis
}
void listInsert(Cell*& list, int value) {
Cell* newCell = new Cell;
newCell->value = value;
newCell->next = list;
list = newCell;
}
list
list
value
137
newCell
137
???

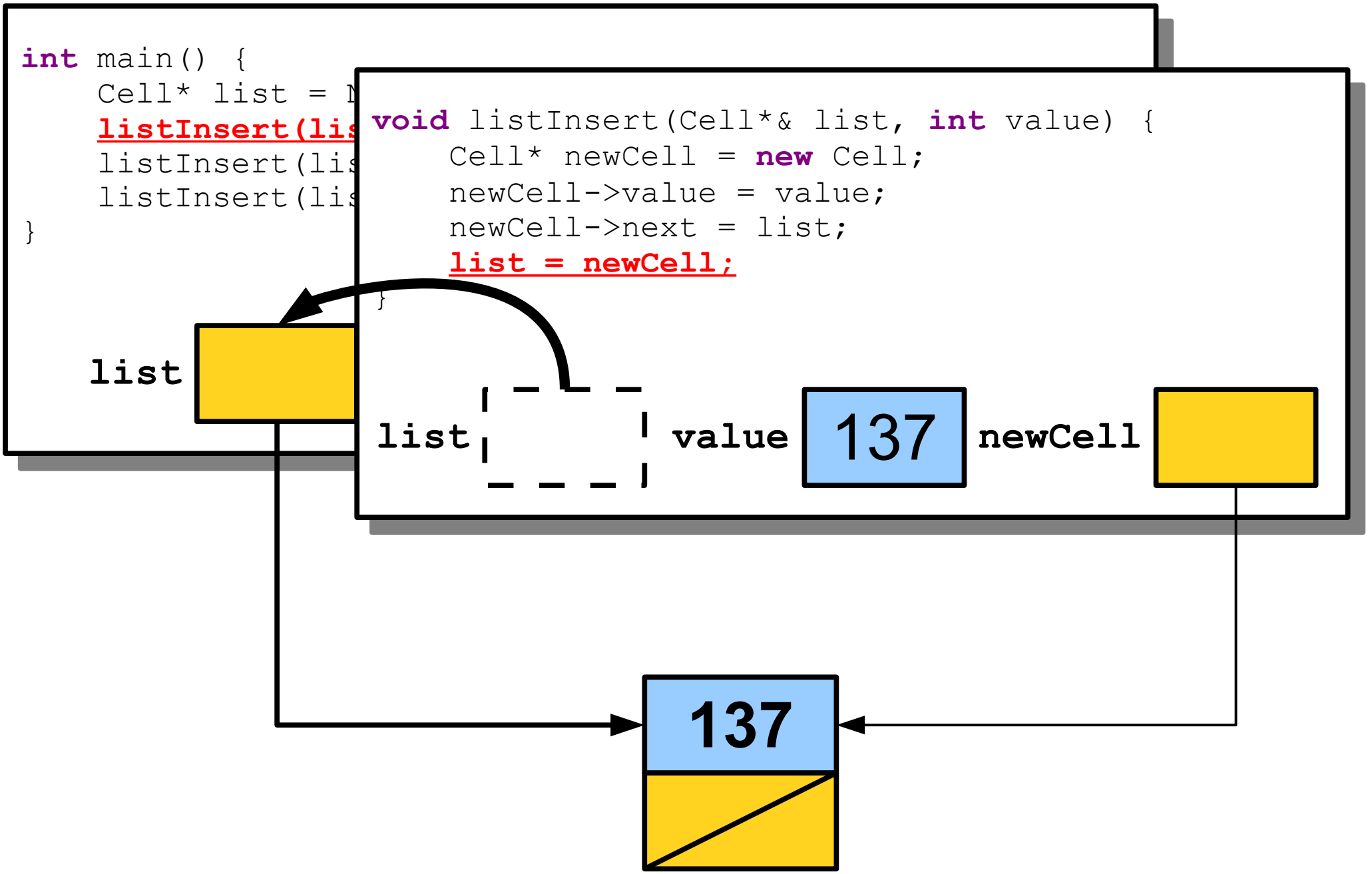

int main() {
Cell* list = N
listInsert(lis
listInsert(lis
listInsert(lis
}
void listInsert(Cell*& list, int value) {
Cell* newCell = new Cell;
newCell->value = value;
newCell->next = list;
list = newCell;
}
list
list
value
137
newCell
137
???

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

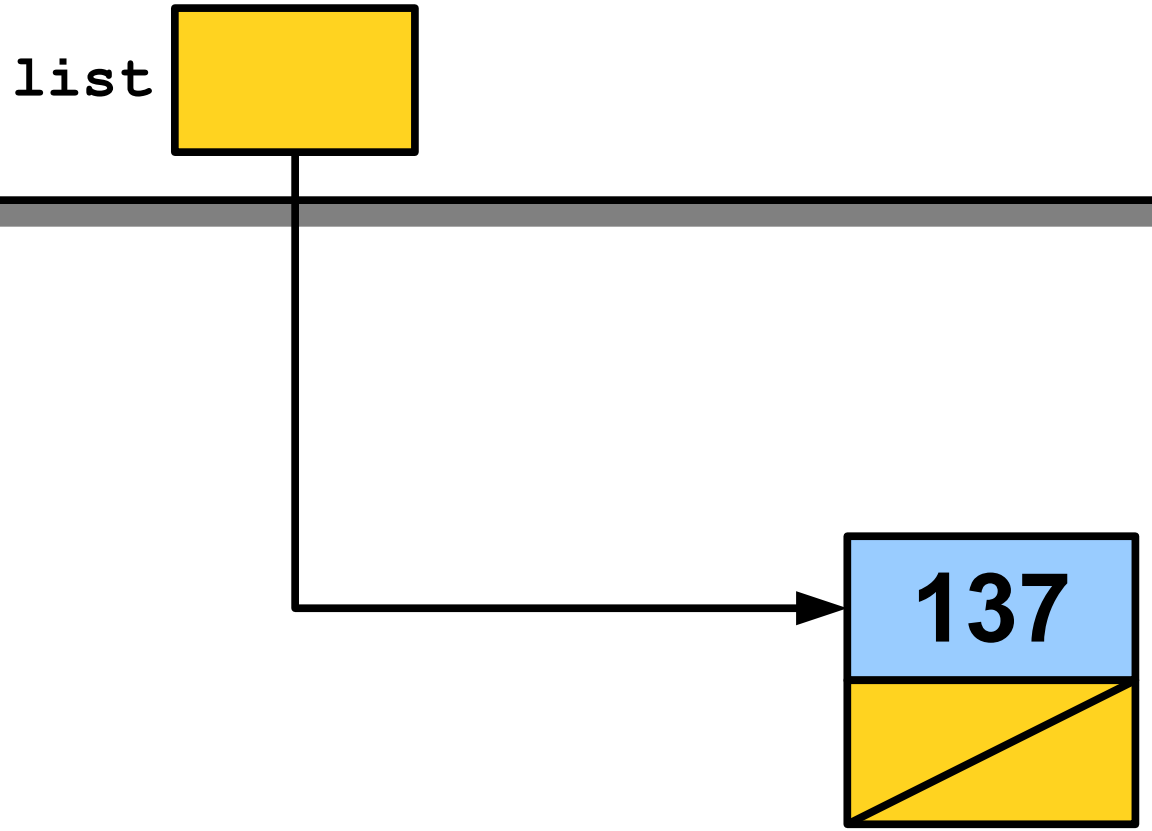
```
void listInsert(Cell*& list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```



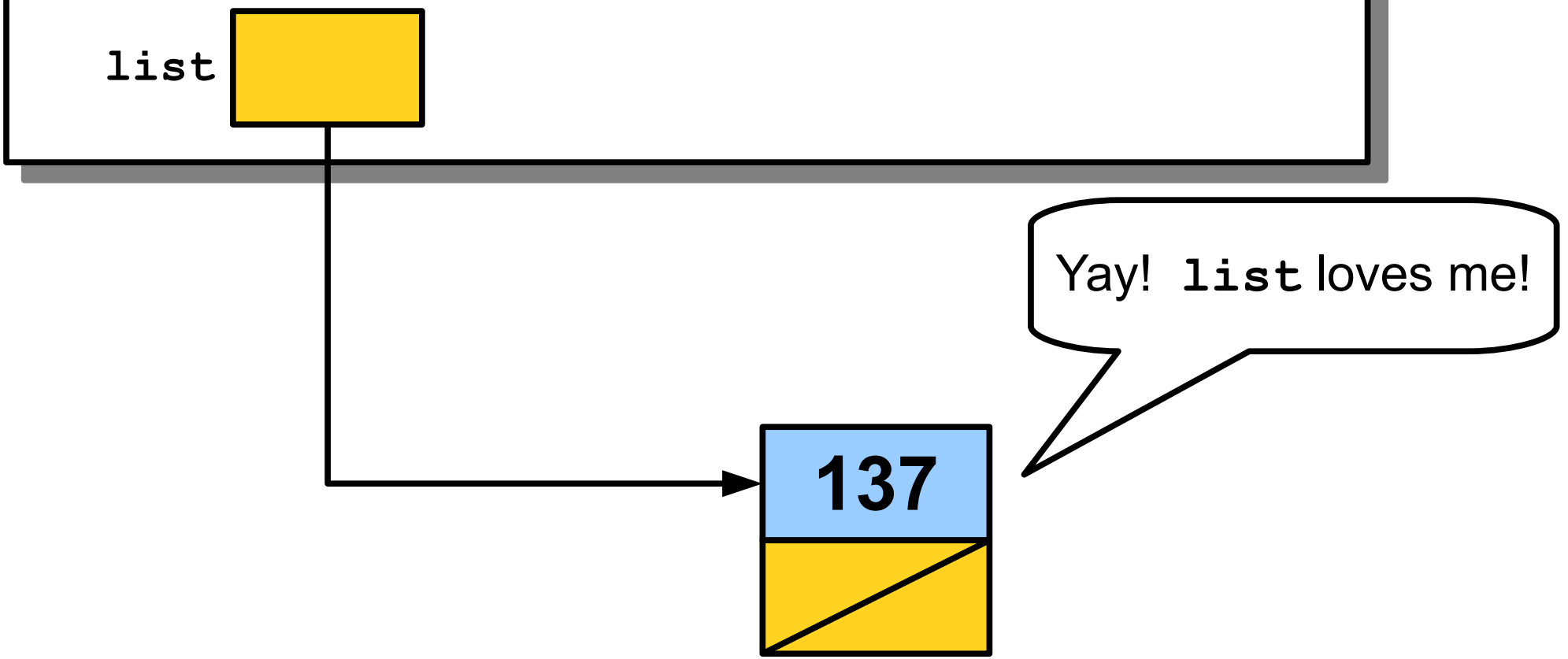

int main() {
Cell* list = N
listInsert(lis
listInsert(lis
listInsert(lis
}
void listInsert(Cell*& list, int value) {
Cell* newCell = new Cell;
newCell->value = value;
newCell->next = list;
list = newCell;
}
list
list
value
137
newCell
137


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;
}
list
list
value
137
newCell
137

```
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);
}
```



Pointers by Reference

- If you pass a pointer into a function *by value,* you can change the contents at the object you point at, but not *which* object you point at.
- If you pass a pointer into a function *by reference,* you can *also* change *which* object is pointed at.

Implementing `Queue`

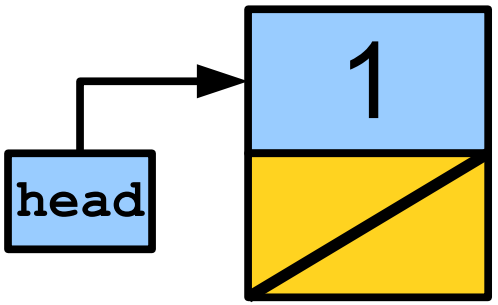
- Earlier, we implemented a queue using two stacks.
- The implementation supported enqueue and dequeue in average-case O(1).
- We can also implement a queue using linked lists!
- Idea:
  - To **enqueue**, append a new cell to the end of the list.
  - To **dequeue**, remove the first cell from the list.

Implementing `Queue`

- Earlier, we implemented a queue using two stacks.
- The implementation supported enqueue and dequeue in average-case O(1).
- We can also implement a queue using linked lists!
- Idea:
  - To **enqueue**, append a new cell to the end of the list.
  - To **dequeue**, remove the first cell from the list.

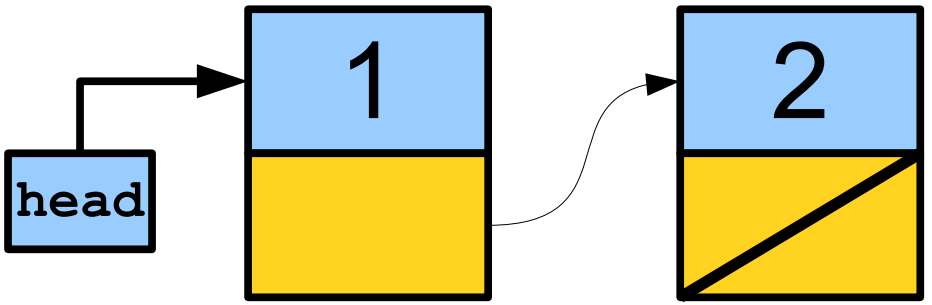
Implementing `Queue`

- Earlier, we implemented a queue using two stacks.
- The implementation supported enqueue and dequeue in average-case O(1).
- We can also implement a queue using linked lists!
- Idea:
  - To **enqueue**, append a new cell to the end of the list.
  - To **dequeue**, remove the first cell from the list.



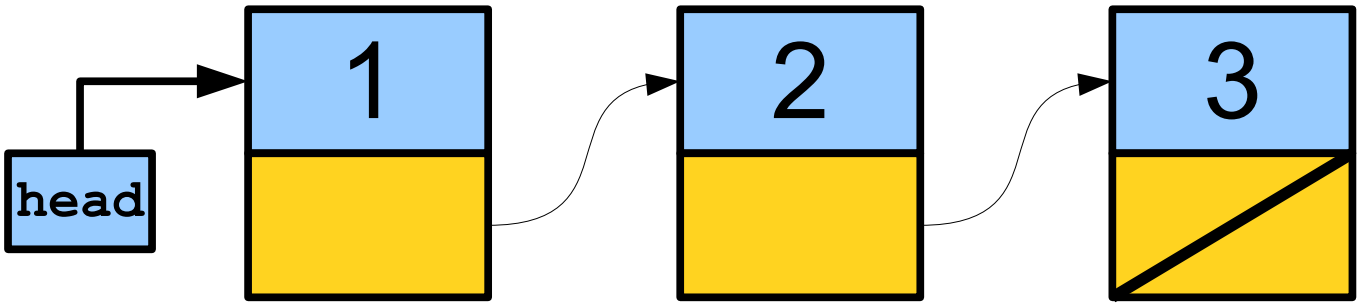
Implementing `Queue`

- Earlier, we implemented a queue using two stacks.
- The implementation supported enqueue and dequeue in average-case O(1).
- We can also implement a queue using linked lists!
- Idea:
  - To **enqueue**, append a new cell to the end of the list.
  - To **dequeue**, remove the first cell from the list.



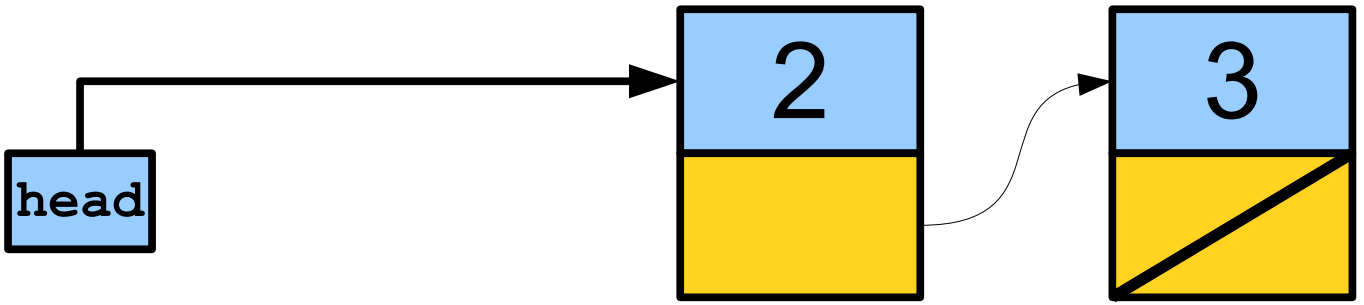
Implementing `Queue`

- Earlier, we implemented a queue using two stacks.
- The implementation supported enqueue and dequeue in average-case O(1).
- We can also implement a queue using linked lists!
- Idea:
  - To **enqueue**, append a new cell to the end of the list.
  - To **dequeue**, remove the first cell from the list.



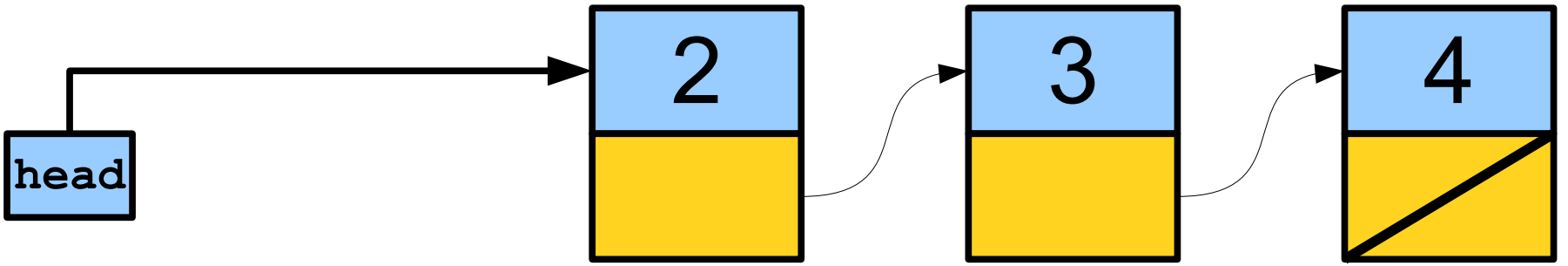
Implementing `Queue`

- Earlier, we implemented a queue using two stacks.
- The implementation supported enqueue and dequeue in average-case O(1).
- We can also implement a queue using linked lists!
- Idea:
  - To **enqueue**, append a new cell to the end of the list.
  - To **dequeue**, remove the first cell from the list.



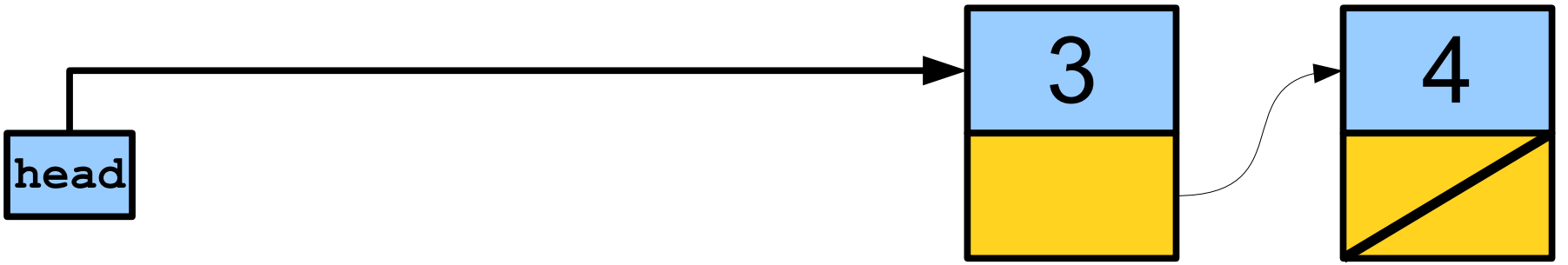
Implementing `Queue`

- Earlier, we implemented a queue using two stacks.
- The implementation supported enqueue and dequeue in average-case O(1).
- We can also implement a queue using linked lists!
- Idea:
  - To **enqueue**, append a new cell to the end of the list.
  - To **dequeue**, remove the first cell from the list.



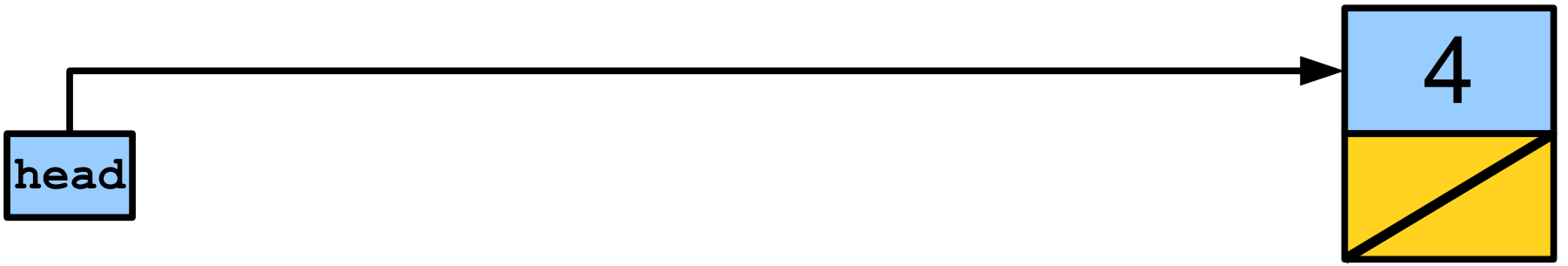
Implementing **Queue**

- Earlier, we implemented a queue using two stacks.
- The implementation supported enqueue and dequeue in average-case O(1).
- We can also implement a queue using linked lists!
- Idea:
  - To **enqueue**, append a new cell to the end of the list.
  - To **dequeue**, remove the first cell from the list.



Implementing **Queue**

- Earlier, we implemented a queue using two stacks.
- The implementation supported enqueue and dequeue in average-case O(1).
- We can also implement a queue using linked lists!
- Idea:
  - To **enqueue**, append a new cell to the end of the list.
  - To **dequeue**, remove the first cell from the list.



Implementing `Queue`

- Earlier, we implemented a queue using two stacks.
- The implementation supported enqueue and dequeue in average-case O(1).
- We can also implement a queue using linked lists!
- Idea:
  - To **enqueue**, append a new cell to the end of the list.
  - To **dequeue**, remove the first cell from the list.

`Queue Data`

`Queue::Queue()`

`Queue::~Queue()`

(Code)

`Queue::enqueue()`
`Queue::dequeue()`
(Pseudocode)

`Queue::enqueue()`
`Queue::dequeue()`
(Code)

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

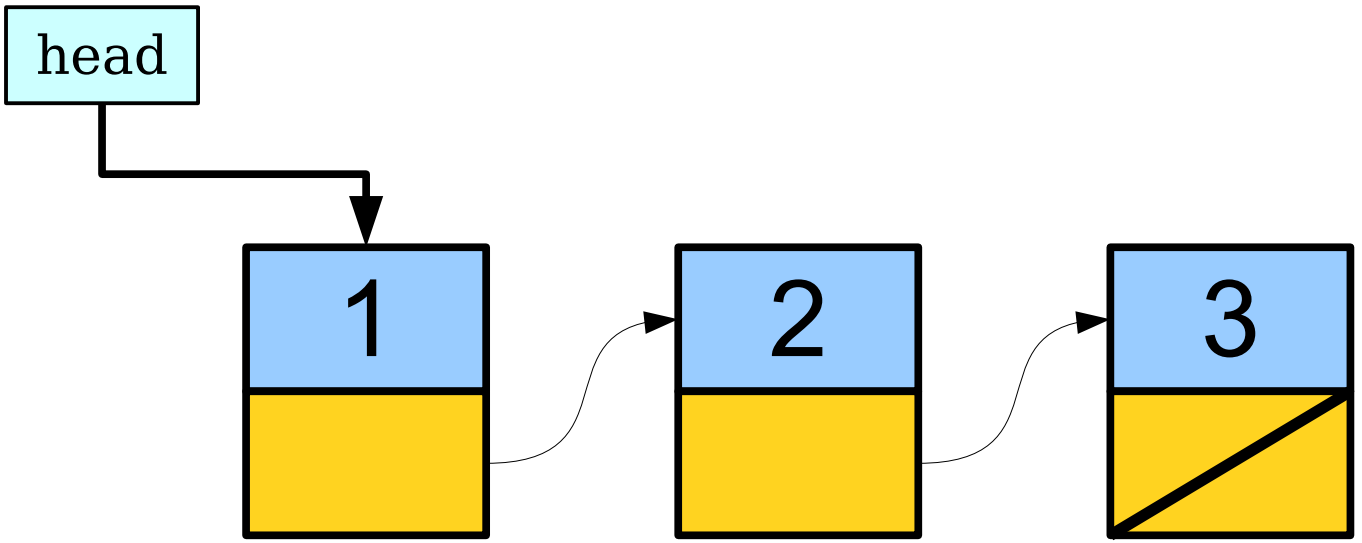
- The O(n) work in enqueue comes from scanning the list to find the end.
- **Idea**: What if we just stored a pointer to the very last cell in the list?

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.

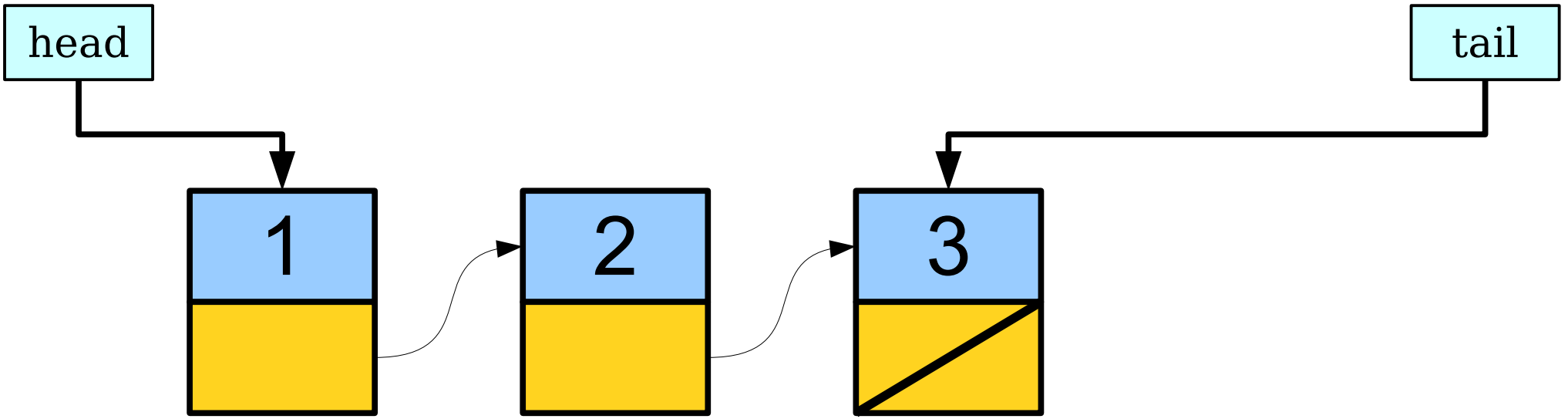
Improving Efficiency

- 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.



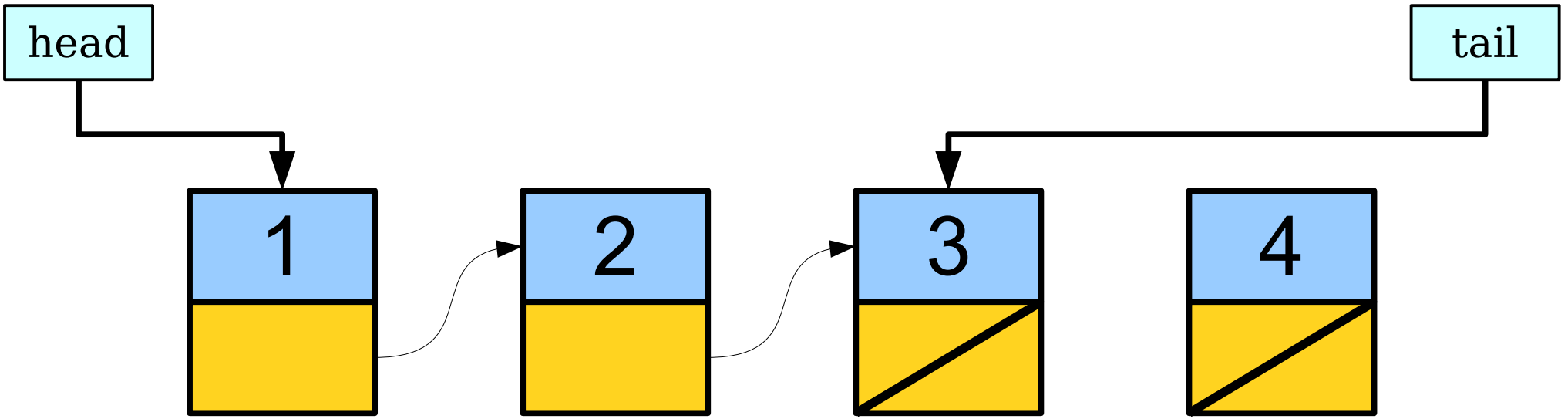
Improving Efficiency

- 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.



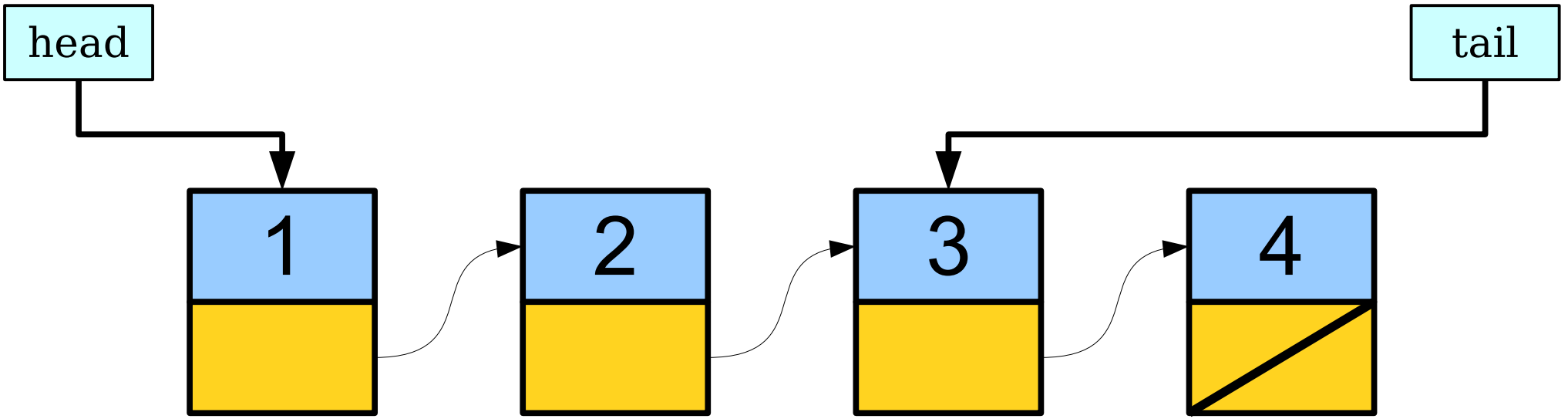
Improving Efficiency

- 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.



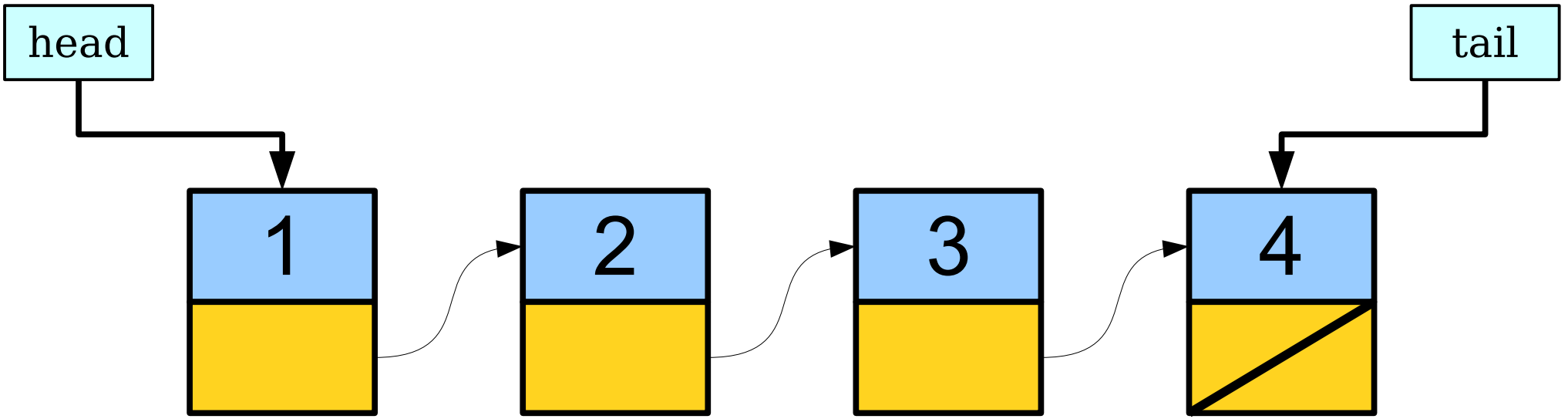
Improving Efficiency

- 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.



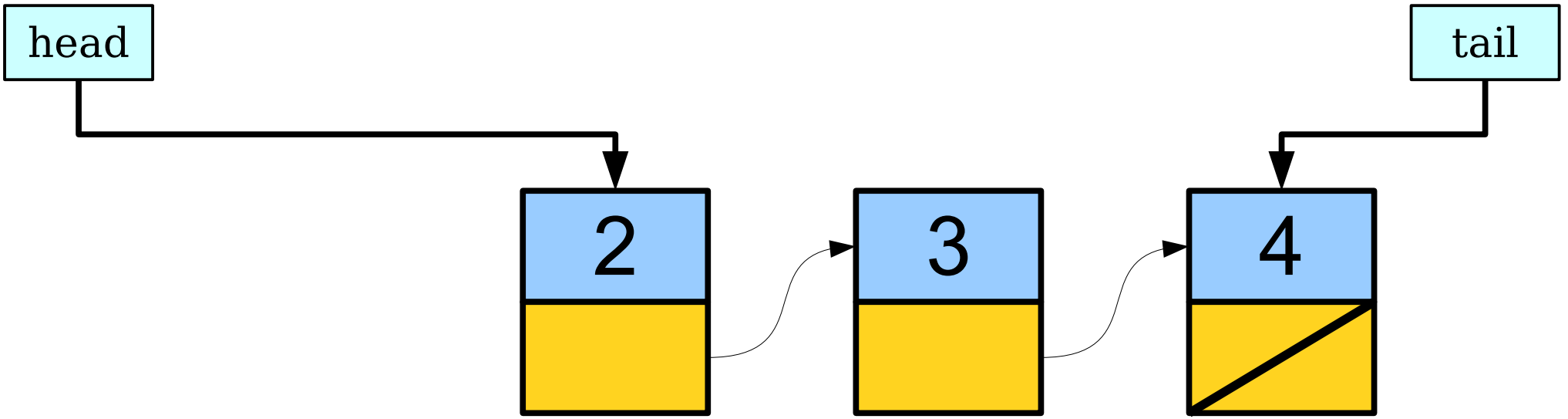
Improving Efficiency

- 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.



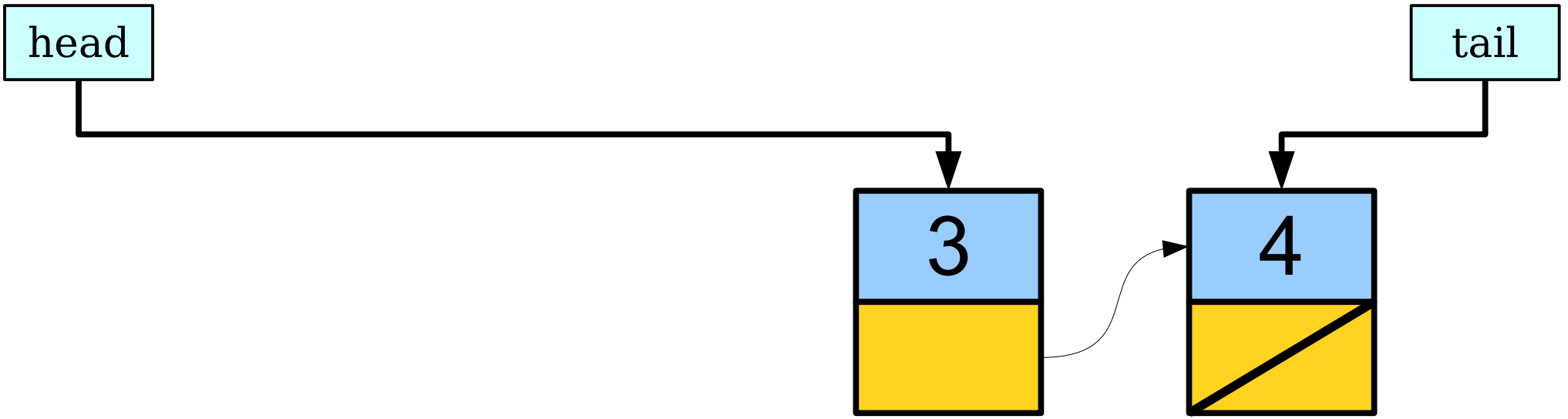
Improving Efficiency

- 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.



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.



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.

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.

New Cases to Consider

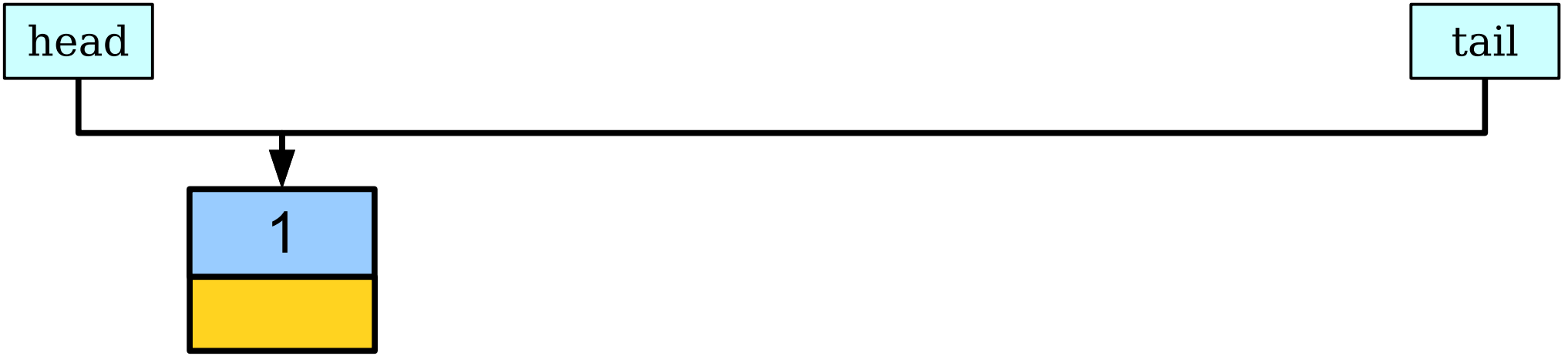
- Enqueuing into an empty queue.

New Cases to Consider

- Enqueuing into an empty queue.

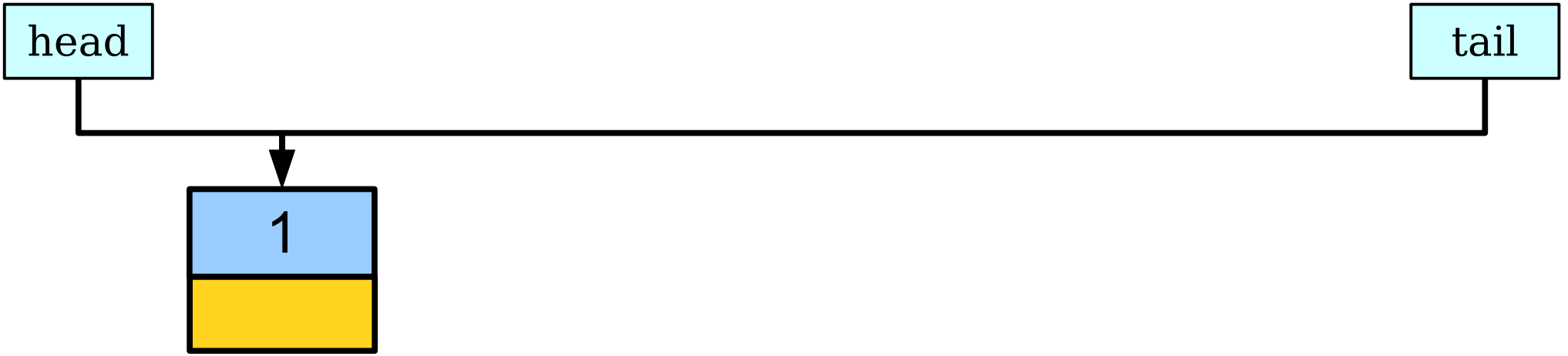
New Cases to Consider

- Enqueuing into an empty queue.

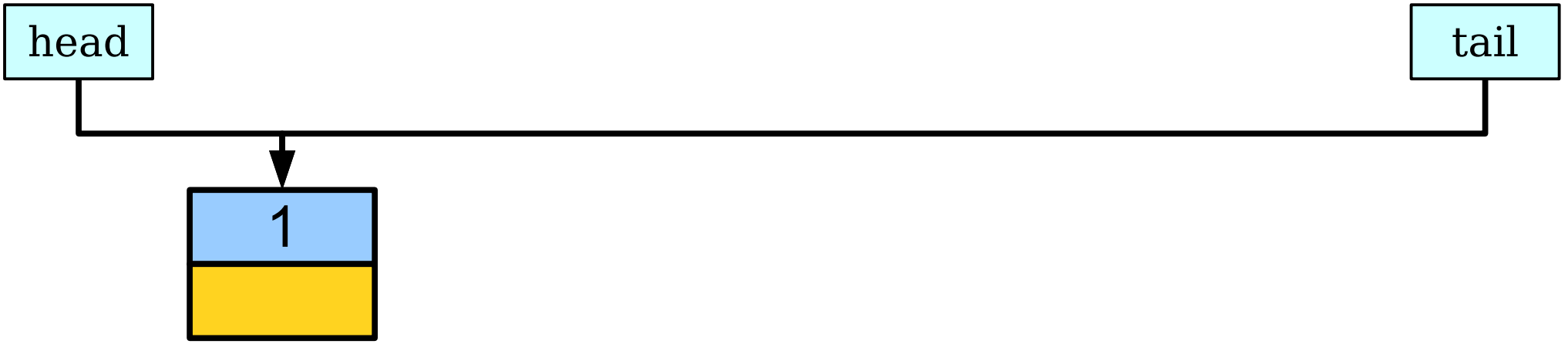


New Cases to Consider

- Enqueuing into an empty queue.

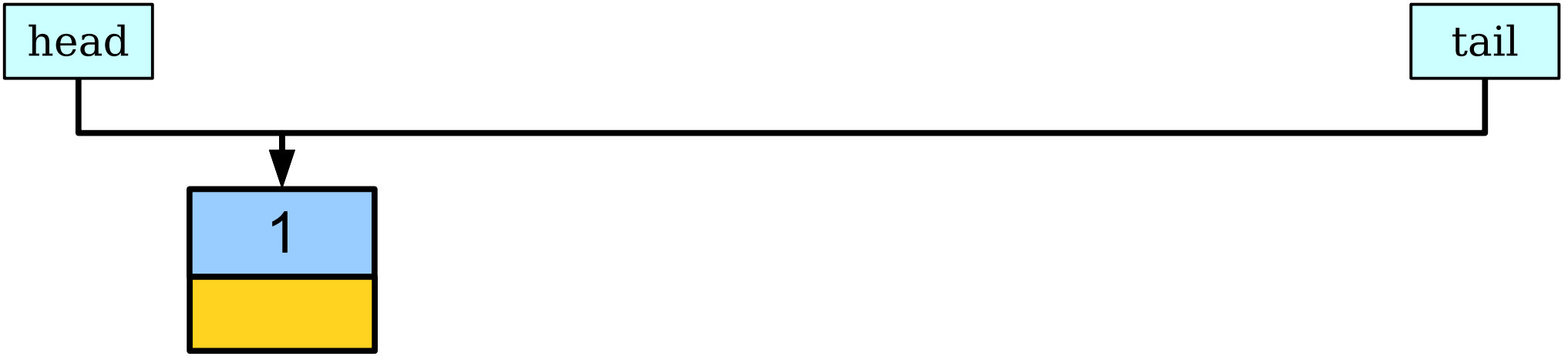


- Dequeuing the last element of a queue.



New Cases to Consider

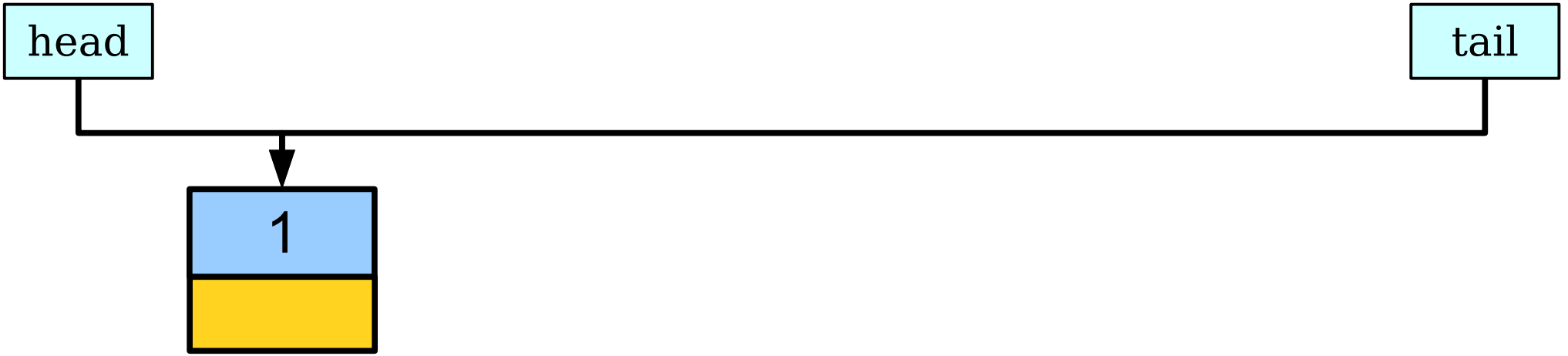
- Enqueuing into an empty queue.



- Dequeuing the last element of a queue.

New Cases to Consider

- Enqueuing into an empty queue.



- Dequeuing the last element of a queue.

`Queue::enqueue()`
`Queue::dequeue()`
(Code)

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)**.

Analyzing our Queue

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

`OurQueue` (linked list)

vs.

`Queue<int>` (dynamic array)

Speed Test

Analyzing our Queue

- Enqueue and dequeue 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.

Tomorrow: Hashing!