Suppose we want to be able to insert efficiently at either the head or the tail of a list. Why is one operation more efficient than the other?

Suppose we want to be able to insert efficiently at either the head or the tail of a list. Why is one operation more efficient than the other?

Answer: Because inserting at the head takes a constant/fixed amount of time (just a couple of assignment statements) while insertion at the tail requires traversal of (scanning through) the entire linked list until the tail node is reached. The same is true for deletion of the tail.

In other words, inserting at the head takes a constant amount of time while insertion at the tail takes time proportional to the length of the linked list, which can be problematic if the list is very long, e.g., millions of nodes.

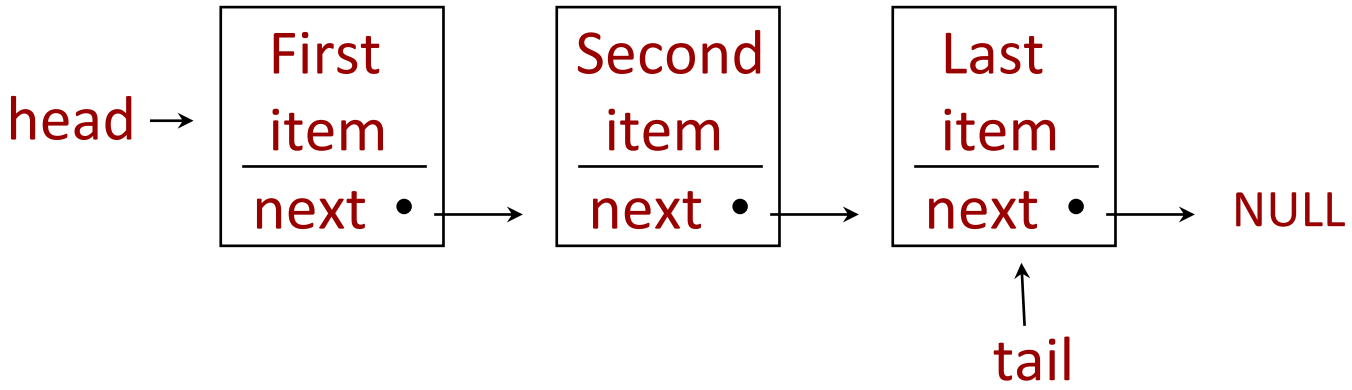
Answer: Because inserting at the head takes a constant/fixed amount of time (just a couple of assignment statements) while insertion at the tail requires traversal (scanning through) the entire linked list until the tail node is reached. The same is true for deletion of the tail.

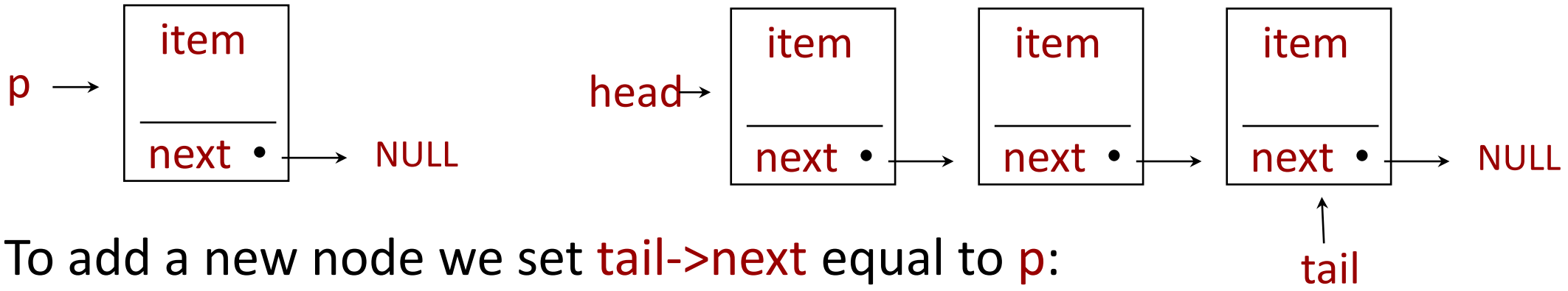
Is there a way we can make insertion at the tail of a linked list as efficient as inserting at the head?

Is there a way we can make insertion at the tail of a linked list as efficient as inserting at the head? **Yes:**

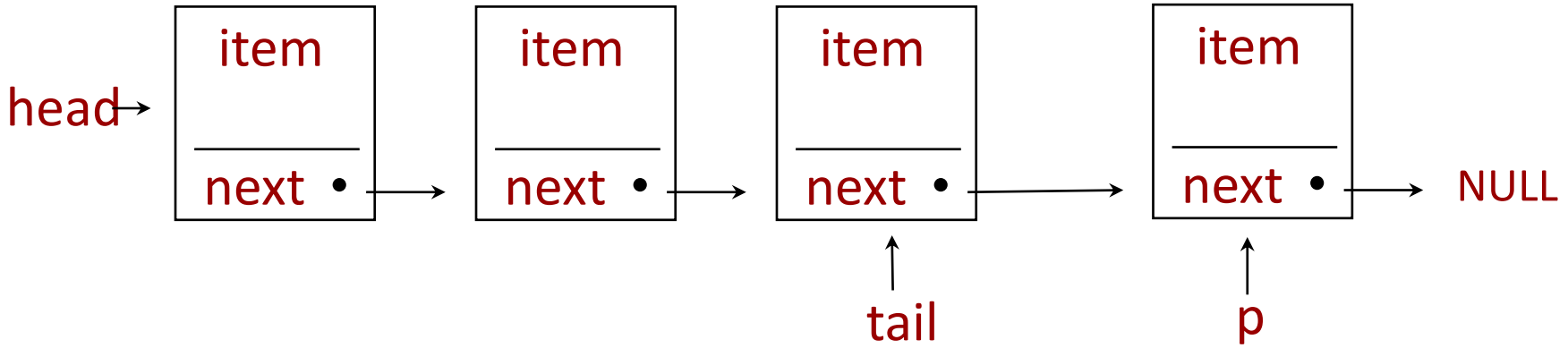
```
typedef struct {
        Node *head, *tail;
} List;
```

Now we have direct access to the last node in the linked list. The location for the next added item is pointed to by tail->next :

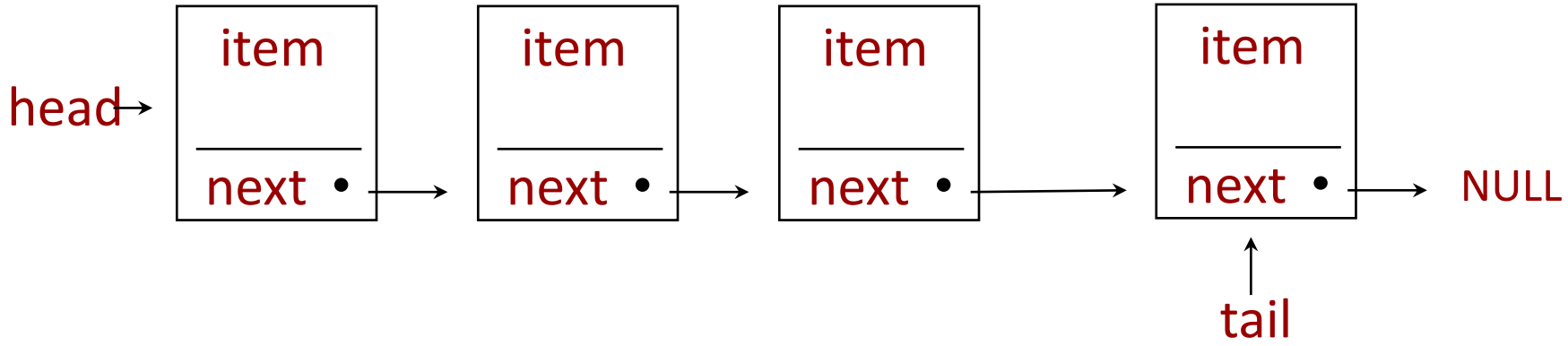




To add a new node we set tail->next equal to p:

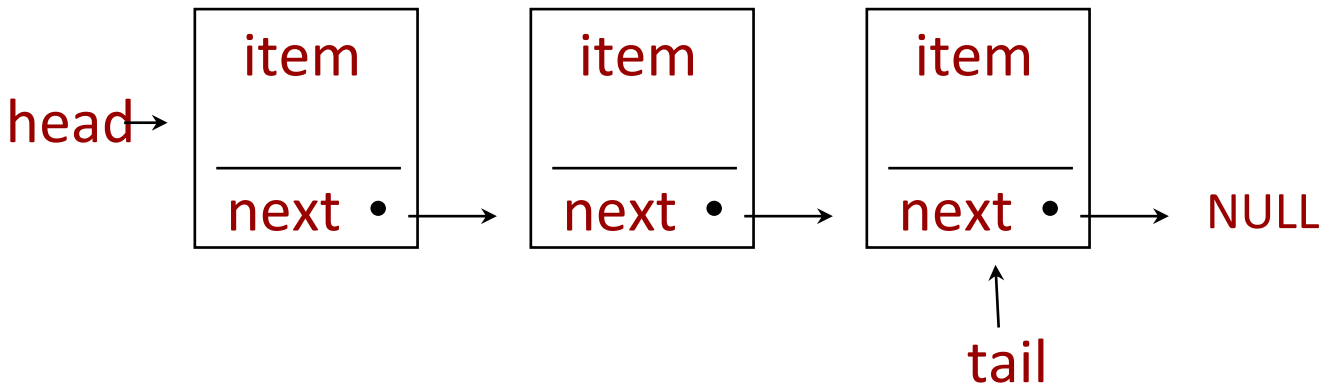


The last thing we need to do is update our tail pointer as tail=p:



```
typedef struct {
     Node *head, *tail;
} List;
```

QUESTION: Below shows the case of a list that has three items, but what would tail point to if the list is empty?



```
typedef struct {
     Node *head, *tail;
} List
```

QUESTION: What would tail point to if the list is empty?

ANSWER: It depends how we choose to design our createList function. We could initially have head and tail both equal to NULL.

```
List * createList() {
    List *p;
    if ( p = malloc(sizeof(List)) )   // Test for malloc fail
         p->head = p->tail = NULL;
    return p;
}
```

```
typedef struct {
     Node *head, *tail;
} List;
```

Below shows that addItem must treat empty lists as special cases:

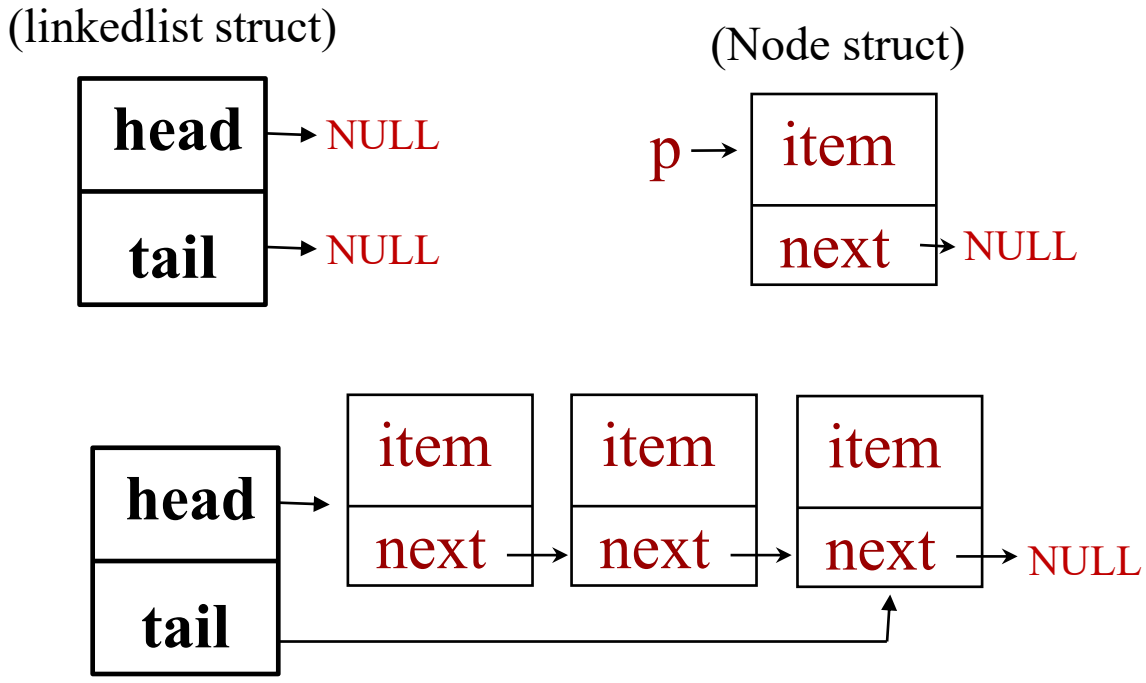
```
List * createList() {
    List *p;
    p = malloc(sizeof(List));
    p->head = p->tail = NULL;
    return p;
}
```

```
List *addItem(List *h, Key item) {
     Node *p; p = createNode(item);
     if (h->tail == NULL)
          h->head = h->tail = p;
     else {
          h->tail->next = p;
          h->tail = p;
     }
     return(h);
}
```

```
typedef struct {
        Node *head, *tail;
} List;
```

Below shows that addItem must treat empty lists as special cases:

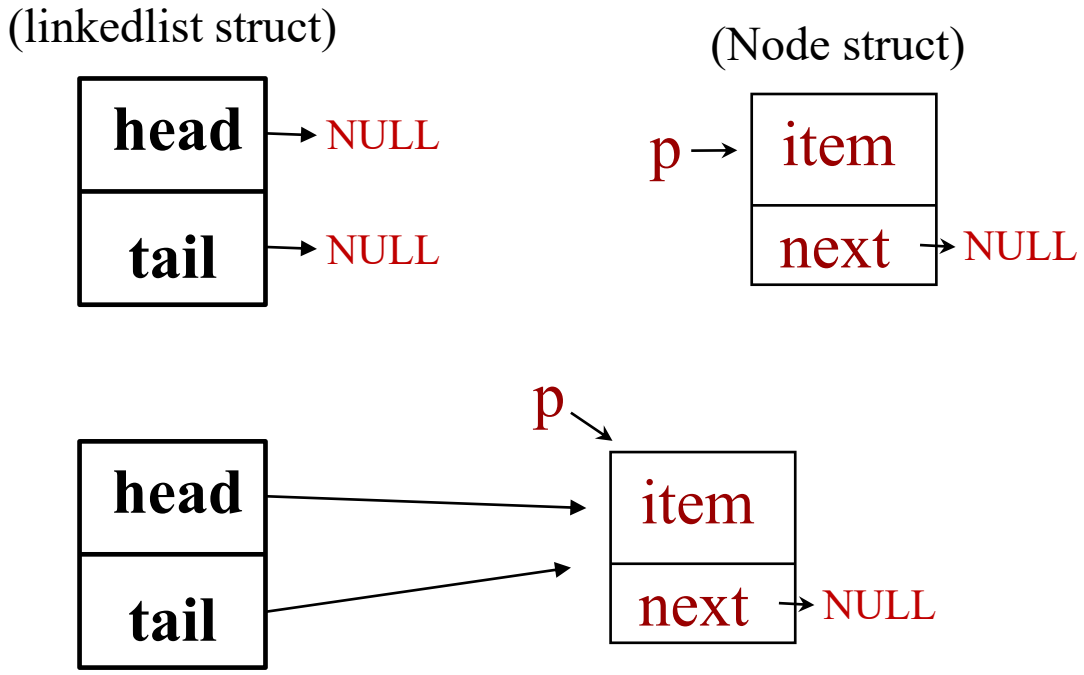
```
Node *addItem(List *h, Key item) {
  Node *p; p = createNode(item);
  if (h->tail == NULL)
      h->head = h->tail = p;
  else {
      h->tail->next = p;
      h->tail = p;
   }
   return(h);
}
```



```
typedef struct {
        Node *head, *tail;
} List;
```

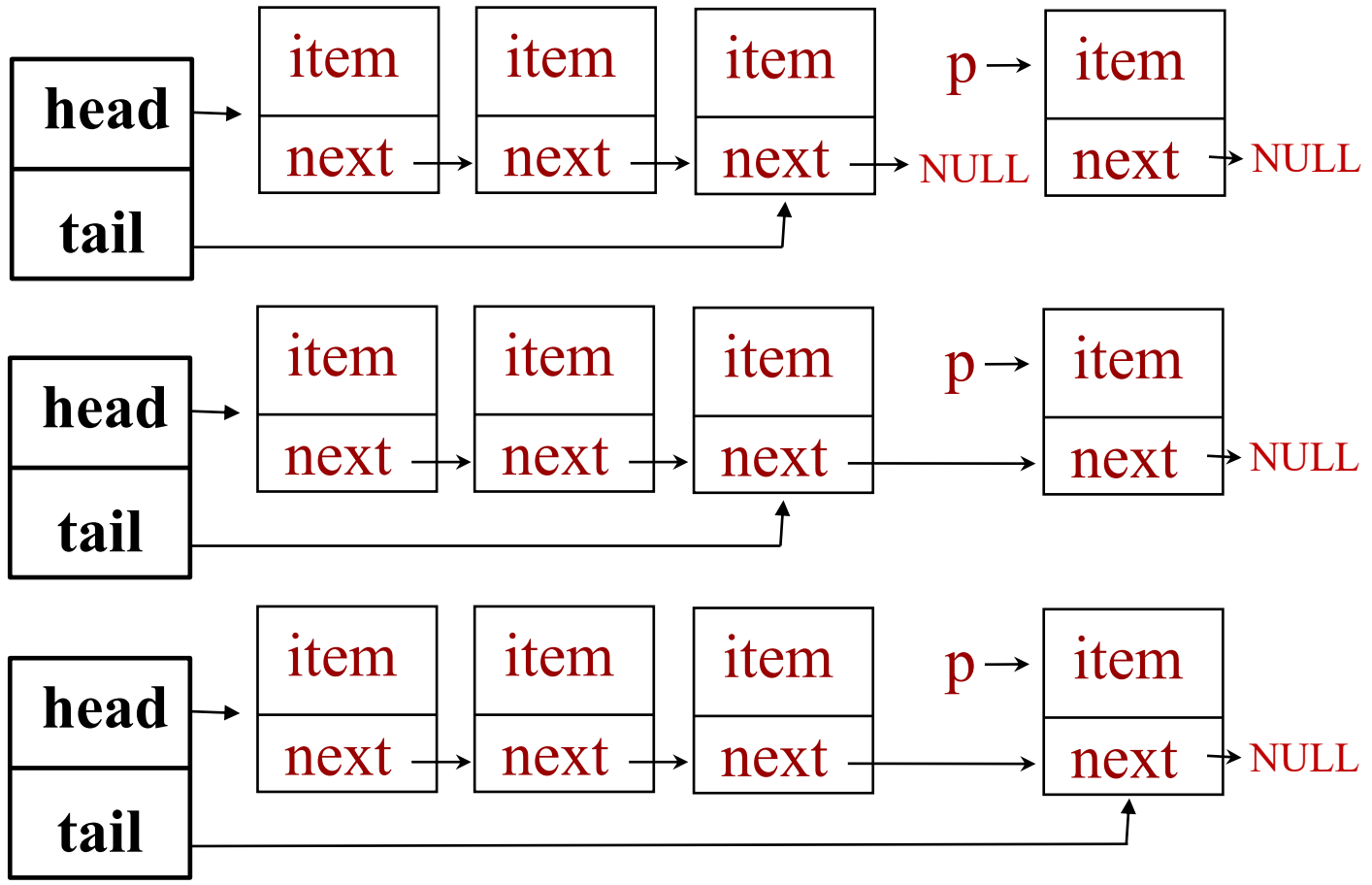
Below shows that addItem must treat empty lists as special cases:

```
if (h->tail == NULL)
    h->head = h->tail = p;
else {
    h->tail->next = p;
    h->tail = p;
 }
```



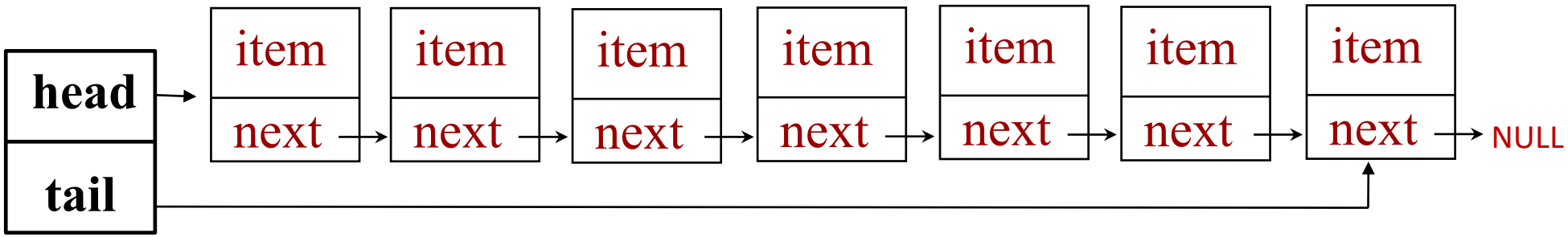
```
typedef struct {
        Node *head, *tail;
} List
```

```
if (h->tail == NULL)
    h->head = h->tail = p;
else {
    h->tail->next = p;
    h->tail = p;
 }
```



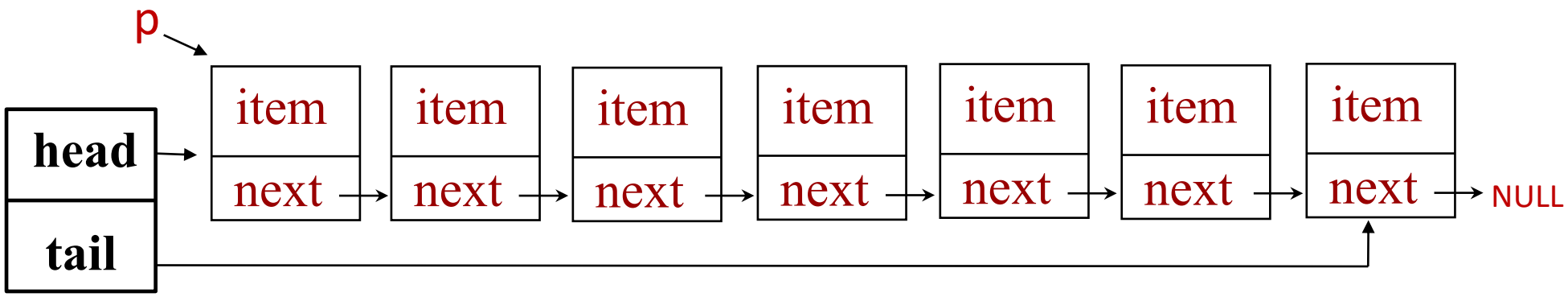
```
typedef struct {
        Node *head, *tail;
} List;
```

QUESTION: How would we implement a function to return the *k*th item from a linked list?



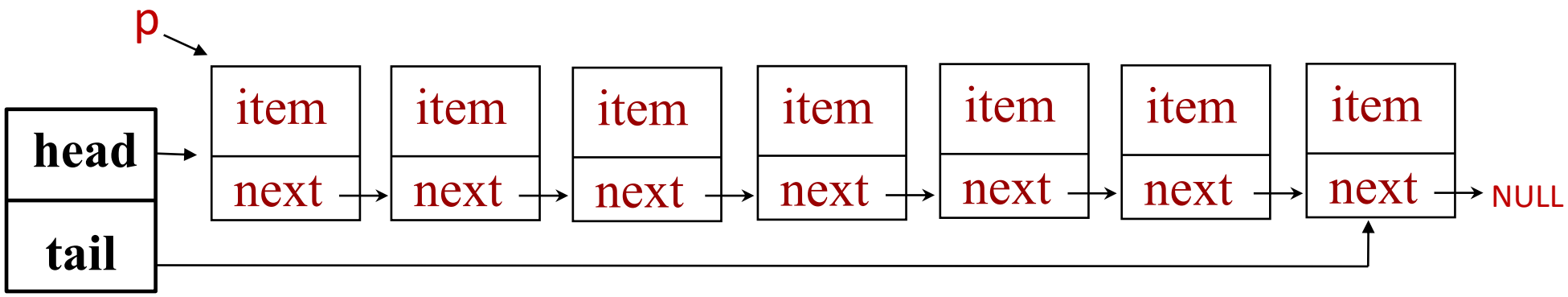
```
typedef struct {
        Node *head, *tail;
} List;
```

We can use a for loop or a while loop and count the number of nodes we visit until we reach the *k*th node. But we need to be sure that p is pointing to the node *before* the *k*th node.



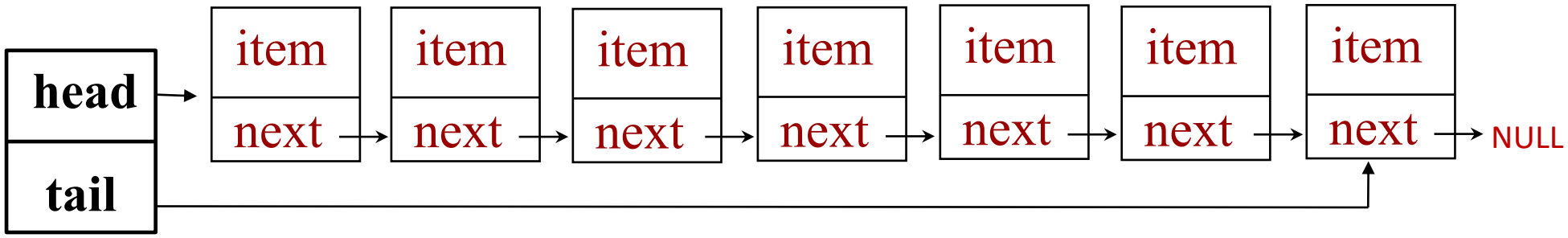
```
typedef struct {
        Node *head, *tail;
} List;
```

NOTE: Don’t forget the case in which the kth node is the tail node. The deletion code will still work correctly – *but the tail pointer will have to be updated.*



```
typedef struct {
        Node *head, *tail;
} List;
```

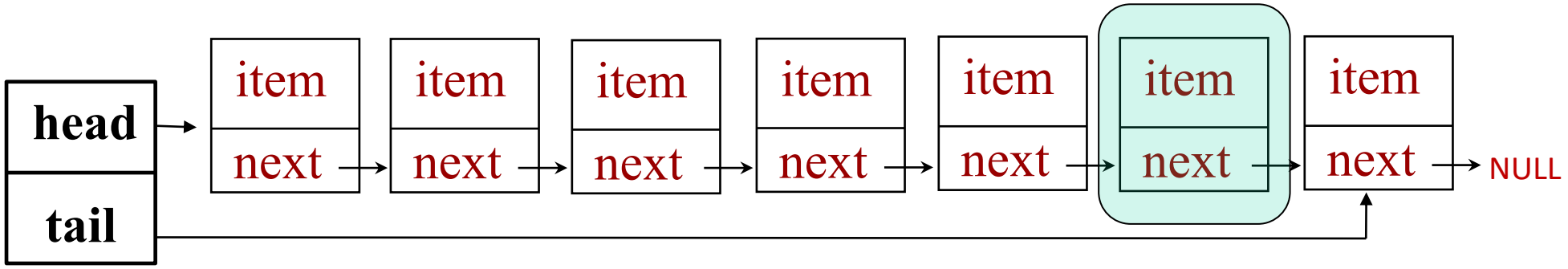
QUESTION: Can the below data structure permit the head to be removed efficiently? How about the tail?



```
typedef struct {
        Node *head, *tail;
} List;
```

QUESTION: Can the below data structure permit the head to be removed efficiently? How about the tail?

ANSWER: The head can be removed efficiently, but note that we don't have direct access to the node *before* the tail.



```
typedef struct {
    Node *head, *tail;
} List;
```

An interesting thing to note is that the above struct can allow us to define our ADT so that the user declares variables to be of type `List` rather than `List*`. That way from a user's perspective it *looks like* a standard data type and not a pointer:

```
List myList, yourList, temp;
```

The downside is that it appears we now have to implement our interface functions to either return `List` variables or require the user to pass `List` variables by reference. Why?

```
typedef struct {
    Node *head, *tail;
} List;
```

```
List myList, yourList, temp;
```

The downside is that it appears we now have to implement our interface functions to either return `List` *variables or require the user to pass* `List` *variables by reference. Why?*

A problem arises if any of our functions have to change any of the struct members. A dummy node can solve one problem by ensuring that the pointer `head` never changes, but what about `tail`?

A problem arises if any of our functions have to change any of the struct members. A dummy node can solve one problem by ensuring that the pointer `head` *never changes, but what about* `tail`*?*

Yep, `tail` is going to present a problem. Can you think of a way that the following might provide a clue to a more general solution?

```
typedef struct {
        Node *head, *tail;
        int listLength;
} ListInfo;

typedef struct {
        ListInfo *lstruct;
} List;
```

```
typedef struct {
        Node *head, *tail;
        int listLength;
} ListInfo;

typedef struct {
        ListInfo *lstruct;
} List;
```

Now a `List` variable is just a single pointer to a struct, and the address of that struct will of course never change after the user's list is created/initialized.

```
typedef struct {
      Node *head, *tail;
      int listLength;
 } ListInfo;

typedef struct {
      ListInfo *lstruct;
 } List;
```

Now a `List` variable is just a single pointer to a struct, and the address of that struct will of course never change after the user's list is created/initialized.

```
List createEmptyList() {
      List p;
      p.lstruct = malloc(sizeof(ListInfo));
      p.lstruct->head = p.lstruct->tail = NULL;
      p.lstruct->listLength = 0;
      return (p);
}
```

```
typedef struct {
      Node *head, *tail;
      int listLength;
 } ListInfo;

typedef struct {
      ListInfo *lstruct;
 } List;
```

Note that our freeList function will need to free all of the memory we have allocated for the list. That includes all nodes – and the ListInfo struct.

Stack & Queue Collection ADTs

A stack is an Abstract Data Type that supports Last-In-First-Out (LIFO) access to data items. The Push() operation puts a data item onto the *top* of the stack, and the Pop() operation retrieves the last data item put on the stack.

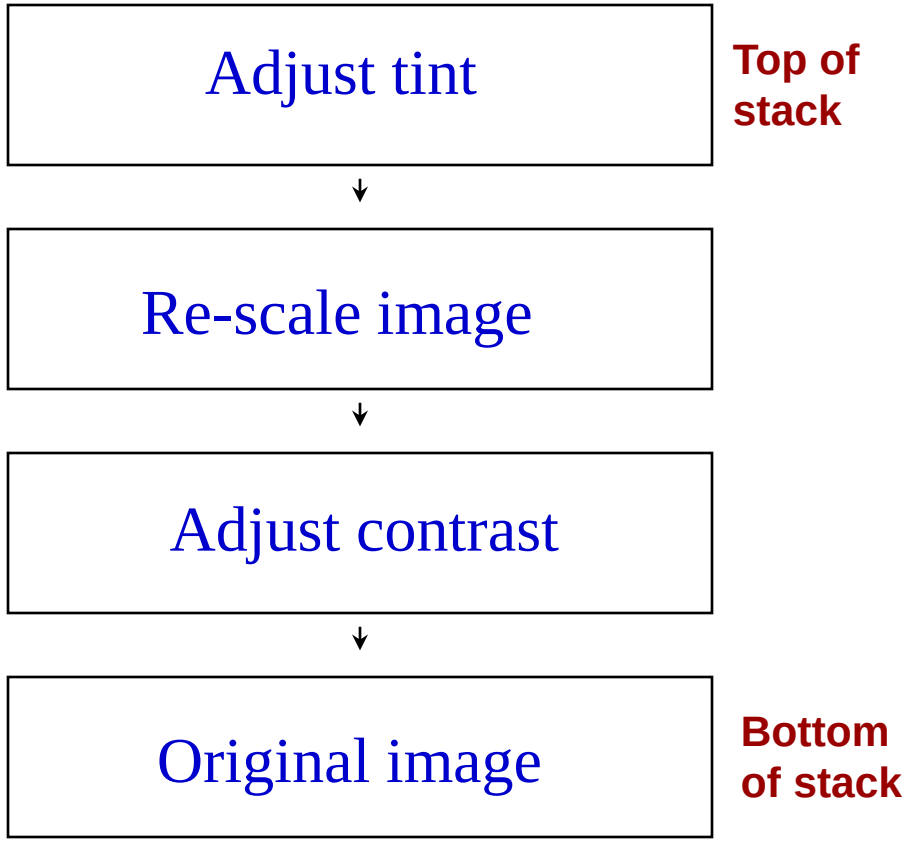
The stack ADT behaves conceptually just like the ordinary notion of “a stack of things” where something is placed on top of the stack or removed from the top of the stack.

A familiar example of the use of a stack is the “undo” facility offered in many software products such as Photoshop and Word.

When the user changes something in one of these programs, a new copy of the current environment is created with the change and Pushed onto a stack. If the user presses an undo button, the current environment is Popped from the stack, thus returning the user to the previous state. Each subsequent undo returns the user to an earlier state.

Use of a stack in Photoshop:

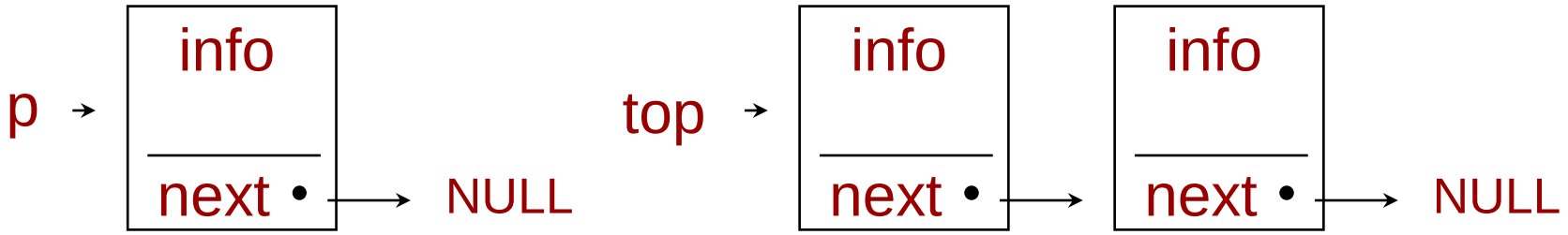
Suppose you start with an image and then apply an operation that adjusts its contrast. In Photoshop this has the effect of creating a new version of the image with adjusted contrast that is conceptually stacked on top of the original image. The same thing occurs after the application of subsequent operations. The "undo" operation simply pops the top version of the image from the stack and thus reverts to the form of the image as it existed prior to the application of the most recent operation.



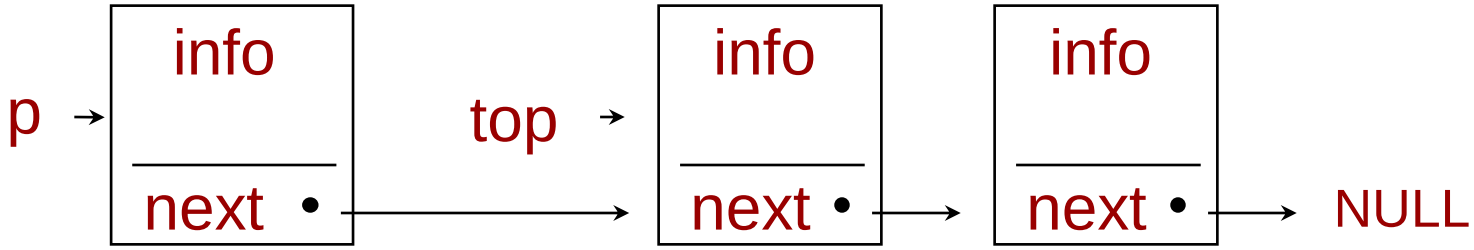
Implementing a Stack

Once we understand the Stack ADT, the question is how it can be implemented so that Push and Pop are as efficient as possible.

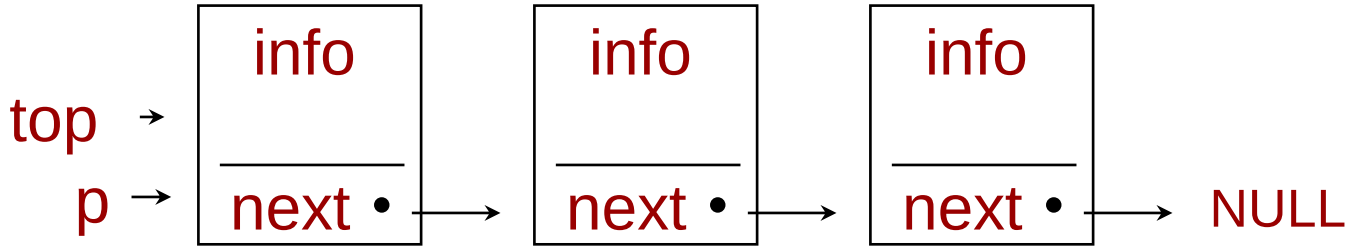
It turns out that Push and Pop can both be supported so that their running times are constant, i.e., independent of the number of items on the stack. Said another way, their running times do not change as we scale up (increase) the number of items on the stack.

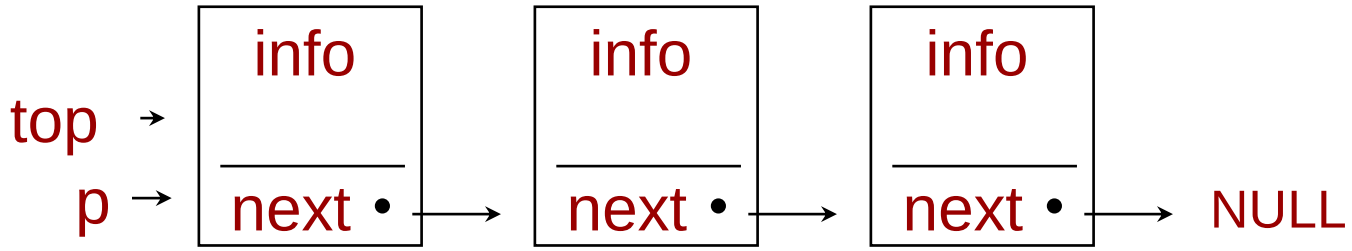


To push onto the stack, we set p->next equal to top:

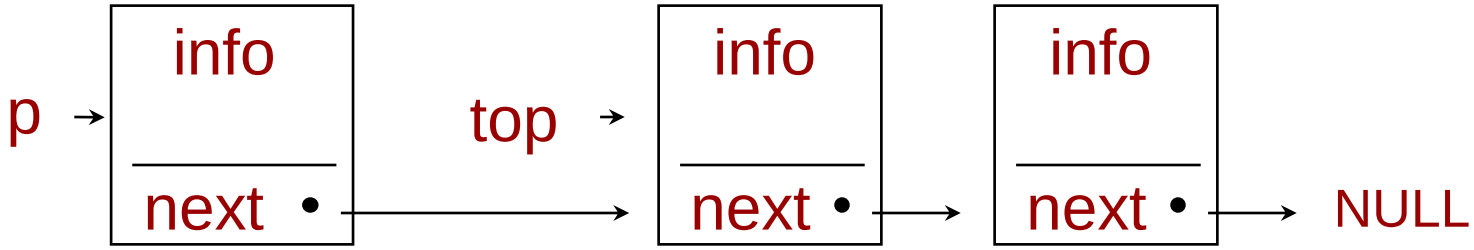


Setting top equal to p completes the Push operation:

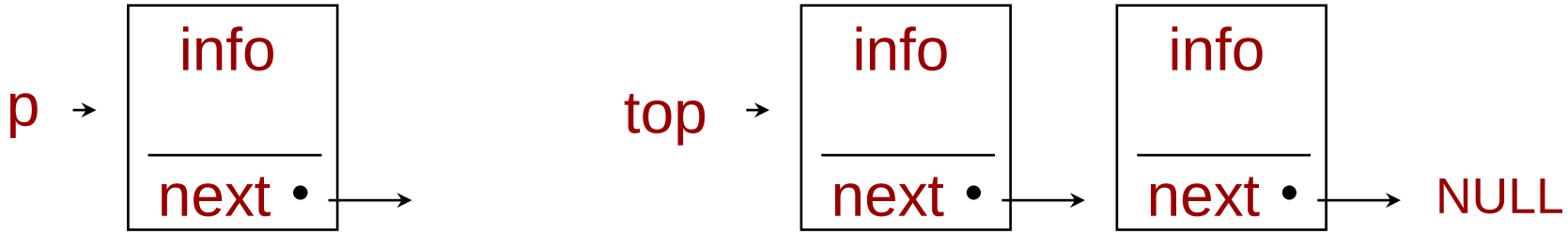




To pop from the stack, we set top equal to p->next:



Returning p completes the Pop operation:



In summary, both of the principal stack operations can be implemented so that they take a constant amount time, i.e., one unit of time, independent of the size of the stack.

The standard way to denote that an algorithm takes only one unit of time is: O(1), which is said in words as "*the algorithm takes Oh-of-1 time*". This is referred to as "Big-O" notation.

It is also common to say that the algorithm has O(1) *computational complexity*, or O(1) *time complexity*, or just O(1) *complexity*. (Big-O seems scarier than it actually is.)

It is also possible to implement the Stack ADT using an array so that Push & Pop have O(1) complexity.

Question: How long does it take to access an element of an array of size N, i.e., an array with N elements?

It is also possible to implement the Stack ADT using an array so that Push & Pop have O(1) complexity.

Question: How long does it take to access an element of an array of size N, i.e., an array with N elements?

Answer: O(1) – the time to access an element of an array of size 100 is the same as that to access an element of an array of size 10 billion. It's a constant amount of time independent of the size of the array.

Question: How long does it take to print all of the integers in an array containing N integers?

Question: How long does it take to print all of the integers in an array containing N integers?

Answer: It would take N units of time, i.e., time proportional to the number of integers printed. That means it would take O(N) time.

Question: How long does it take to access the kth node in a linked list of length N?

Question: How long does it take to access the kth node in a linked list of length N?

Answer: We have to start at the first node and progress from one node to the next until we reach the kth node. That means the time spent is proportional to k, i.e., O(k).

A **queue** is an ADT that supports First-In-First-Out (FIFO) access to data items. The enqueue() operation puts a data item onto the queue, and dequeue() operation retrieves the earliest data item put into the queue.

Unlike a stack, insertions into and deletions from a queue have to be performed at different ends of a linked list.

Queues are often used synonymously with the notion of a *buffer*. For example, virtually all operating systems have a task queue for the CPU. An application generates a task which goes into the task queue, and the tasks are executed in a first-come-first-served order (usually).

Messages sent over the internet may also be queued if the bandwidth capacity at a router is exceeded. There are many uses of queues, and in most cases they just hold things in order of arrival until whatever the things are waiting for is ready.

Prelab 9 (for March 17th)

For this prelab you are to implement six functions for a Queue ADT:

```
/* This function returns the error code from the most
   recently executed queue operation. 0 implies success,
   1 implies out-of-memory error. Some functions may
   document additional error conditions. NOTE: All
   queue functions assign an error code.   */
int getQueueErrorCode(Queue)

/* This function returns an initialized Queue variable.
   Every queue variable must be initialized before
   applying subsequent queue functions. */
Queue queueInit()

/* This function enqueues an object into the queue.
   For convenience, error code is returned directly
   (and also can be obtained via getQueueErrorCode) */
int enqueue(void *, Queue)

/* This function performs dequeue and returns
   object at front of queue. NULL is returned
   if queue is empty and error code is set to 2.
   NOTE: User should check error code if null
   objects are permitted in the queue. */
void * dequeue(Queue)

/* This function returns the number of objects
   in the queue. */
int getQueueSize(Queue)

/* This function uninitializes a queue and frees all
   memory associated with it. NOTE: value of Queue
   variable is undefined after this function is
   applied, i.e., it should not be used unless
   initialized again using queueInit. */
void freeQueue(Queue)
```

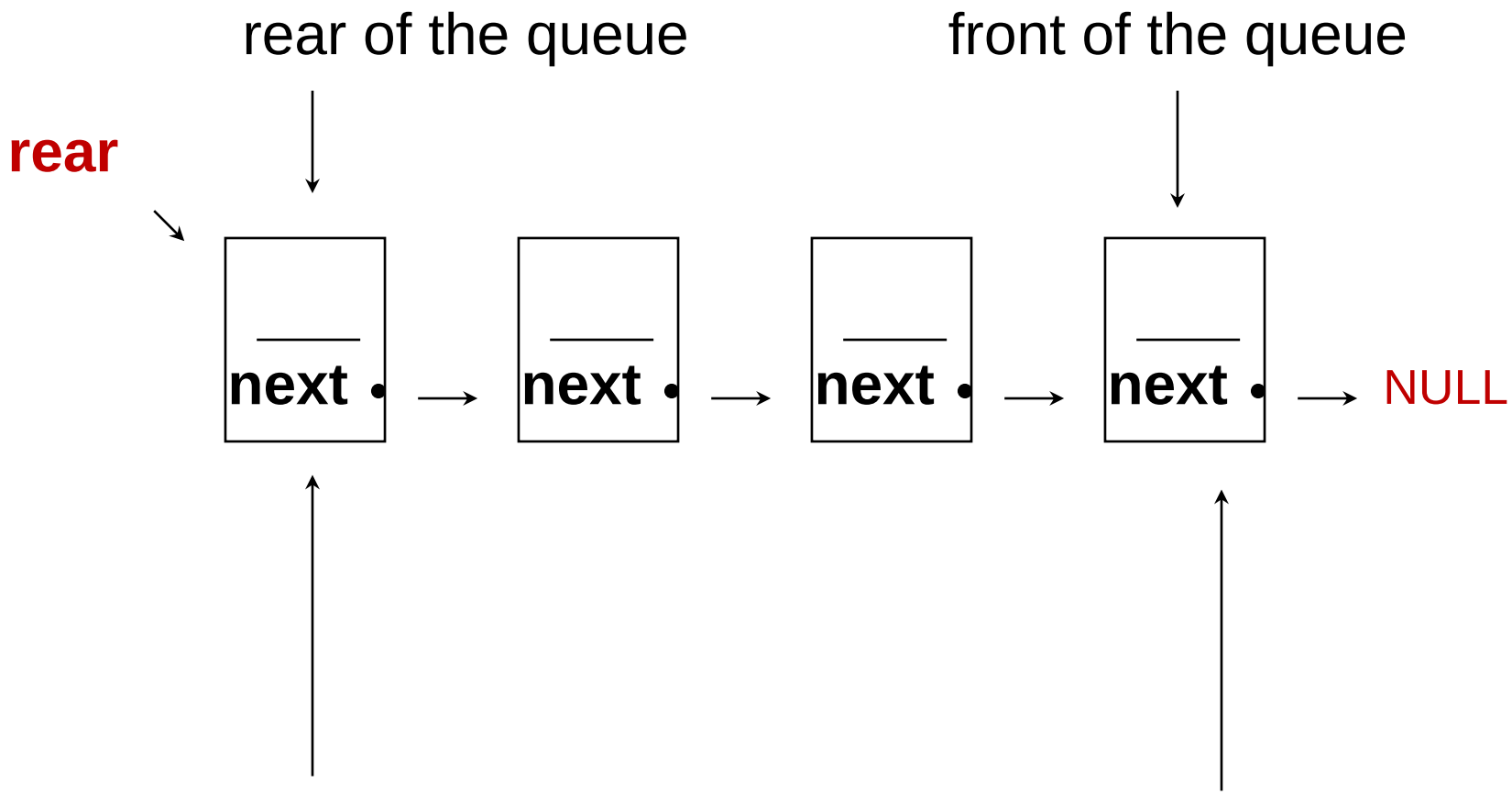
Your underlying representation should be a simple (or circular) linked list with tail pointer so that all functions take only O(1) time except for freeQueue, which obviously must take time proportional to the size of the queue. The real challenge here is designing your implementation so that the user never needs to pass anything by reference. That means that your Queue struct can't contain anything that will ever change after it is initialized.

Feel free to use dummy nodes (or not), but the specification does not allow for doubly-linked lists.

HINT 1: `queueInit` doesn't need to allocate a Queue struct, and if its pointer is NULL, then that can be used by `getQueueErrorCode` to conclude...

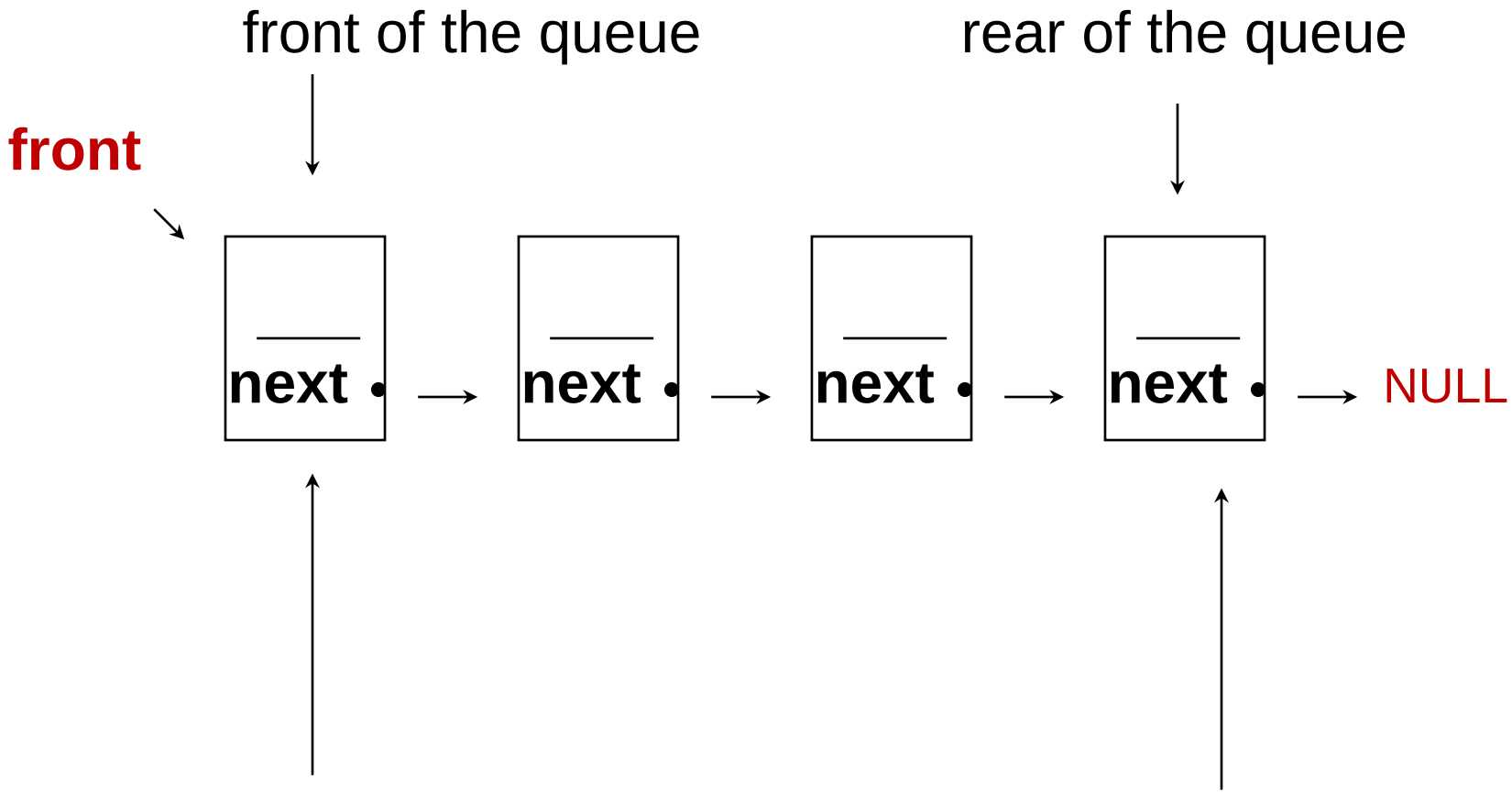
HINT 2:

```
typedef struct {
    void *p;
} PointerWrapper;

PointerWrapper x;

x.p = malloc(sizeof(Employee));

y = malloc(sizeof(PointerWrapper));
y->p = malloc(sizeof(Employee));
```


rear of the queue
front of the queue
rear
next
next
next
next
NULL
most recently added entry into the queue
earliest item put into the queue, and the next one to be removed

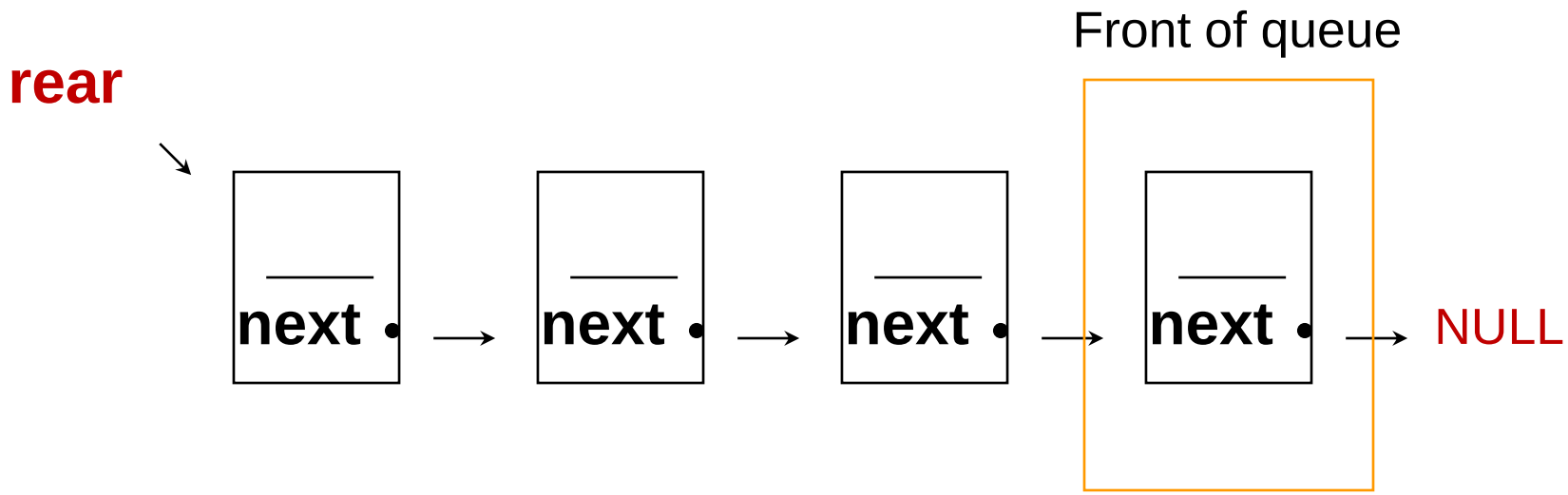
The choice of which end of the underlying linked list is treated as the front or rear of the queue doesn’t matter as long as the queue operations are implemented consistently with the choice.

Remember that the user does not need to know (and should not know) how the queue is implemented. The user only needs to know that the queue operations produce the expected results.

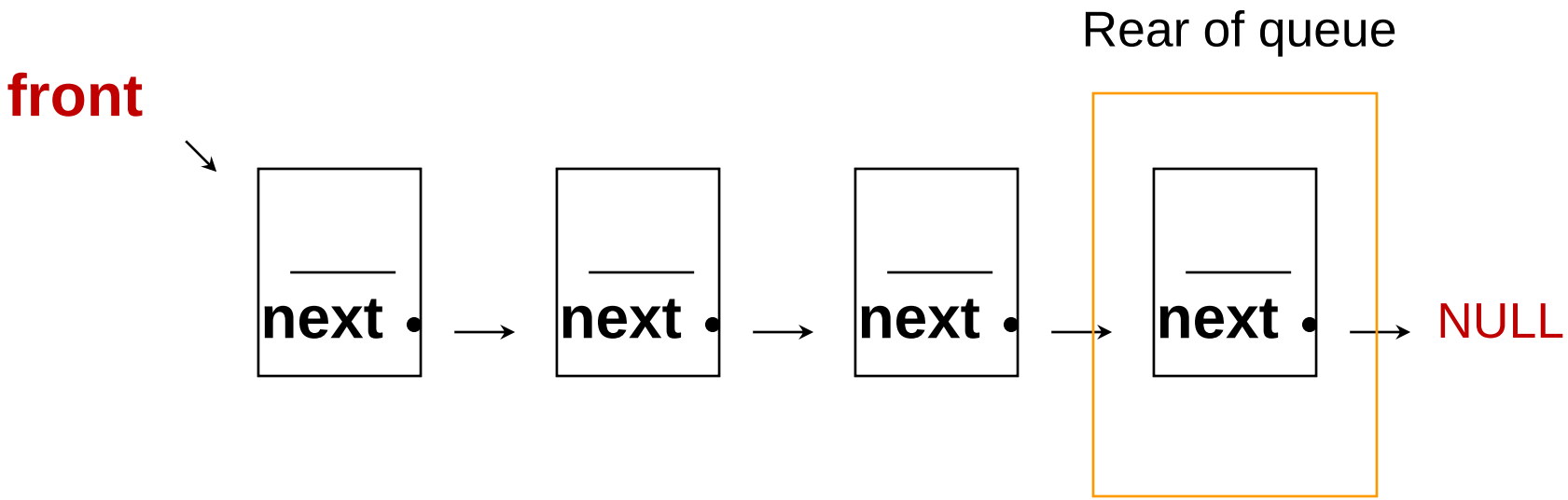


earliest item put into the queue, and the next one to be removed

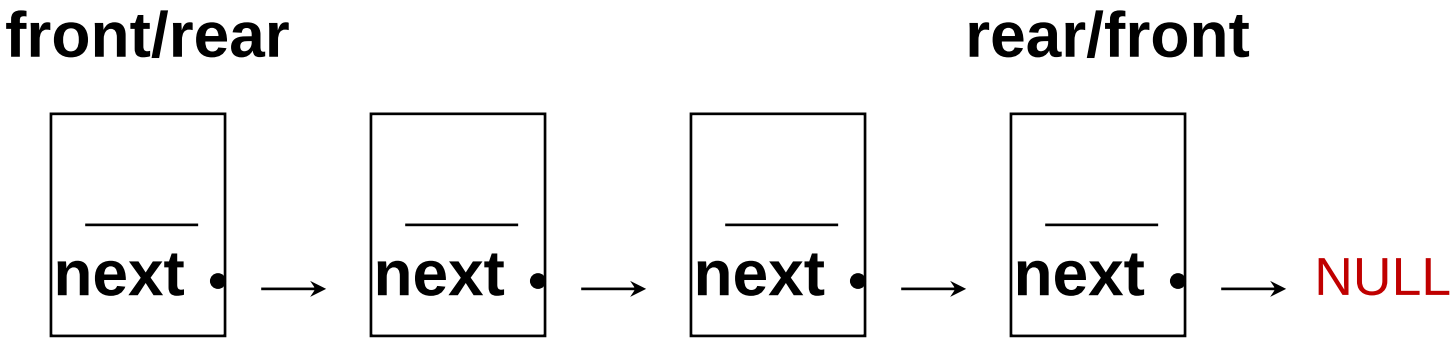
most recently added entry into the queue



In this linked list implementation, putting a node at the rear of the queue is equivalent to pushing onto a stack, but retrieving the node at the front of the queue is very different from popping from a stack.



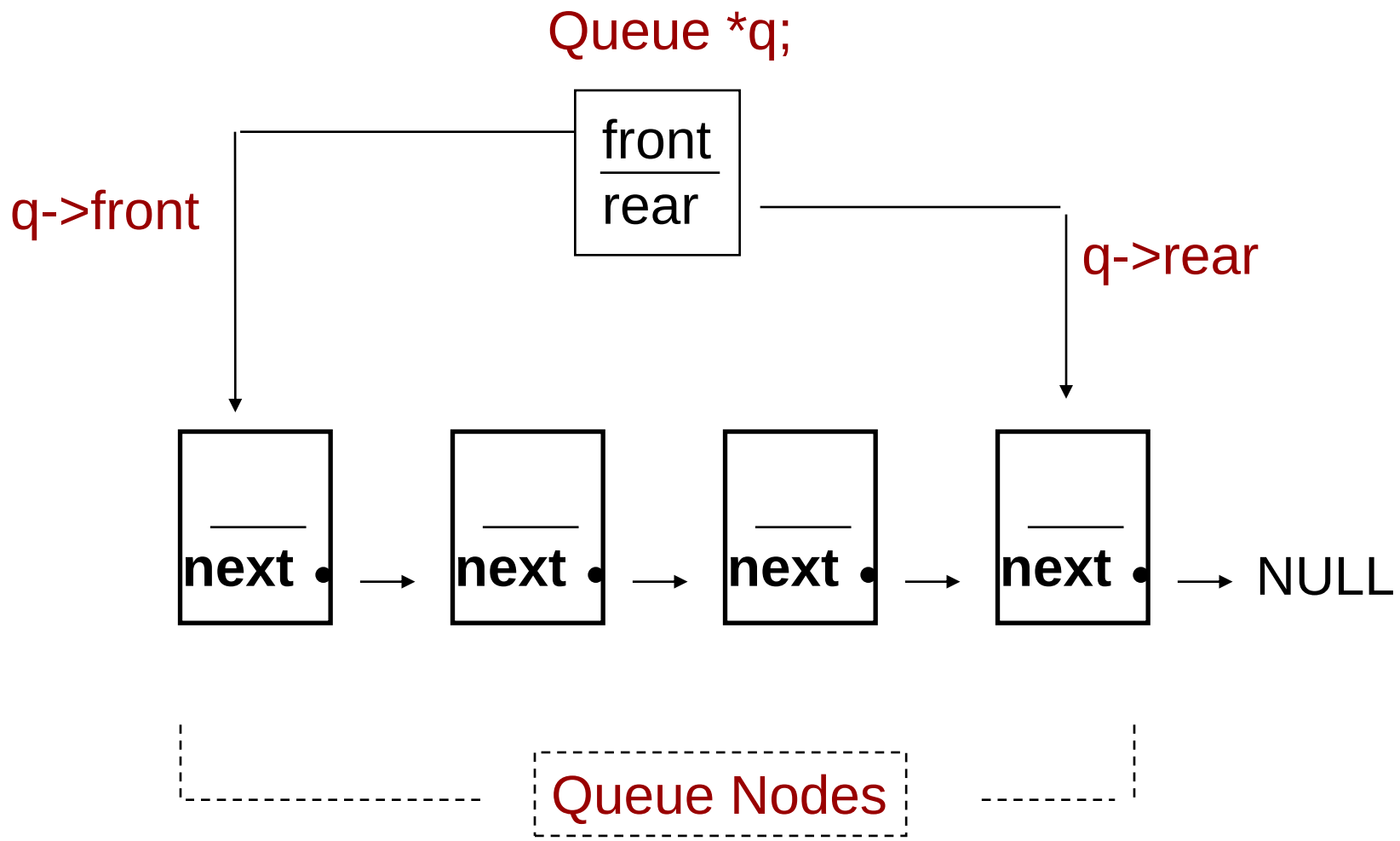
Changing which end is the front and which is the rear of the queue makes removal from the queue equivalent to popping from a stack, but adding a node to the queue now requires a traversing from the beginning of the linked list to the end of the list.



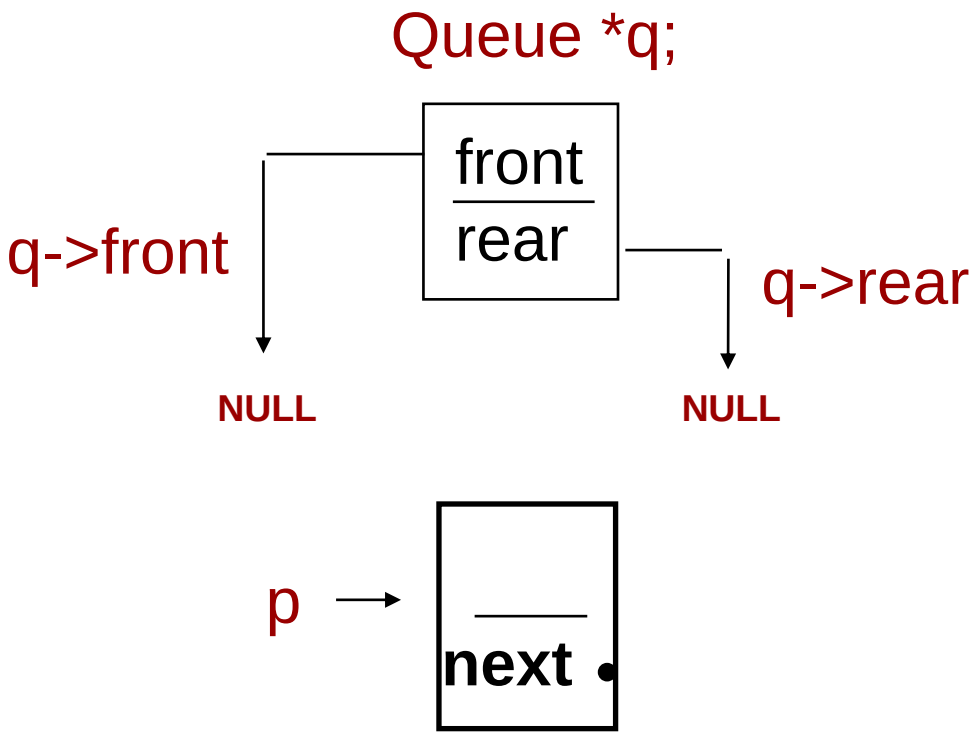
If we use a simple linked list, i.e., with a single pointer to the first node, the decision of which end is the front and which is the rear affects the complexities of the enqueue and dequeue operations. If the list has N items/nodes then one of the operations will take O(1) time and the other will take O(N) time. In other words, a sequence of O(N) operations could take $O(N^2)$ time because an operation could take O(N) time and is repeated O(N) times.

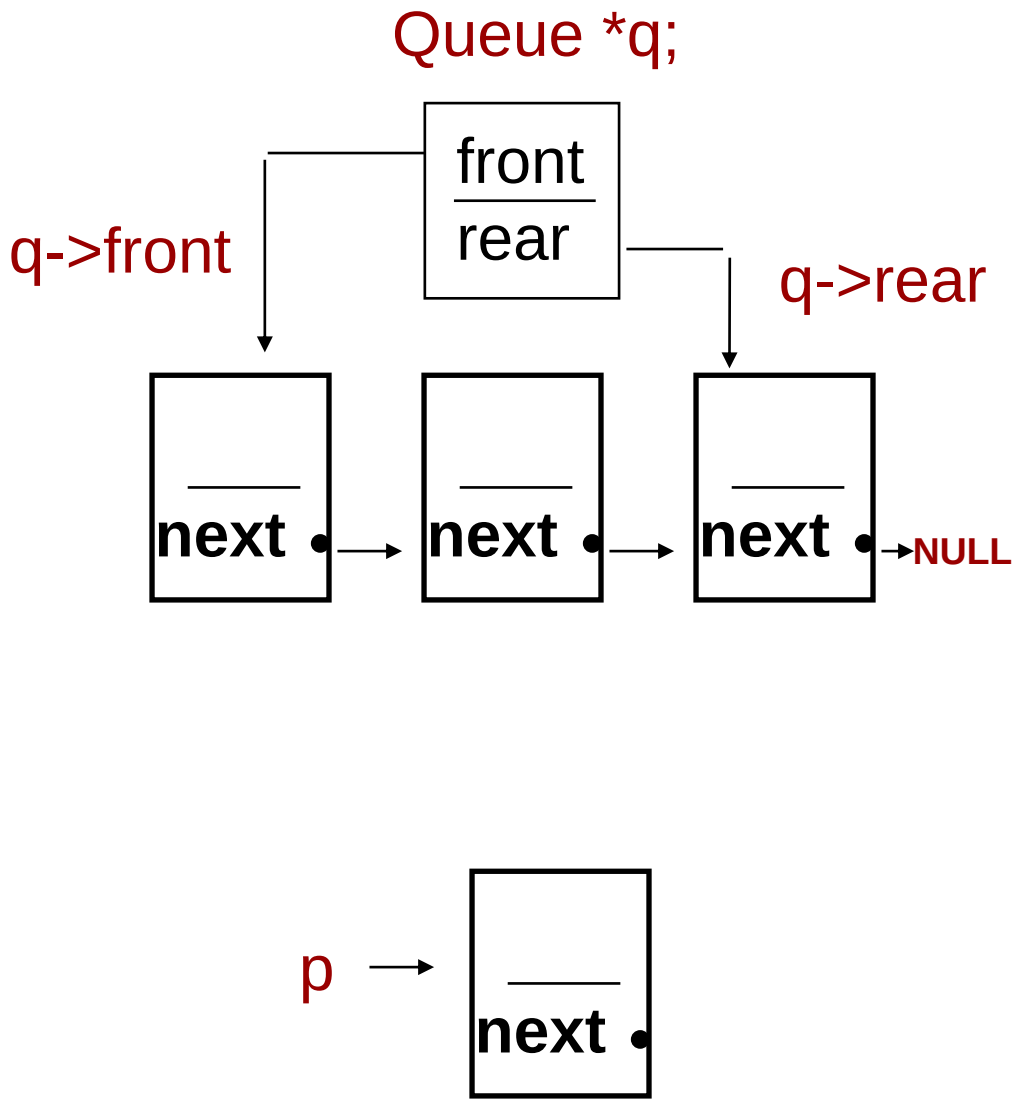
Our goal is to find a way to implement a queue so that each operation takes only O(1) time.

Below is a depiction of how a queue may be maintained using two pointers in a queue structure to provide constant-time -- i.e., O(1) time -- access to the front and rear:

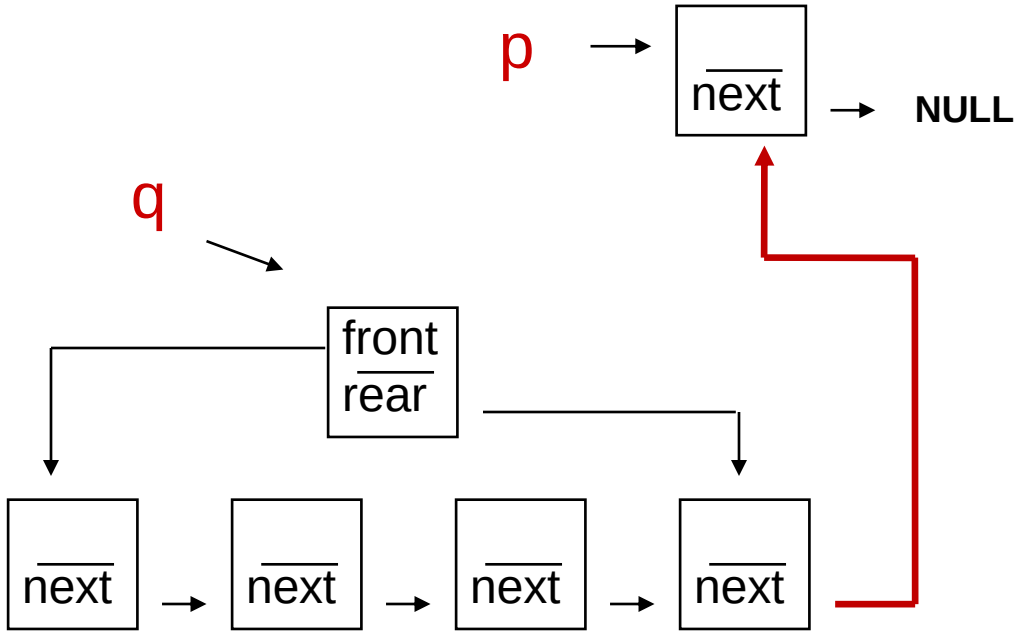


```
typedef struct {
        Node *front, *rear;
} Queue;
```

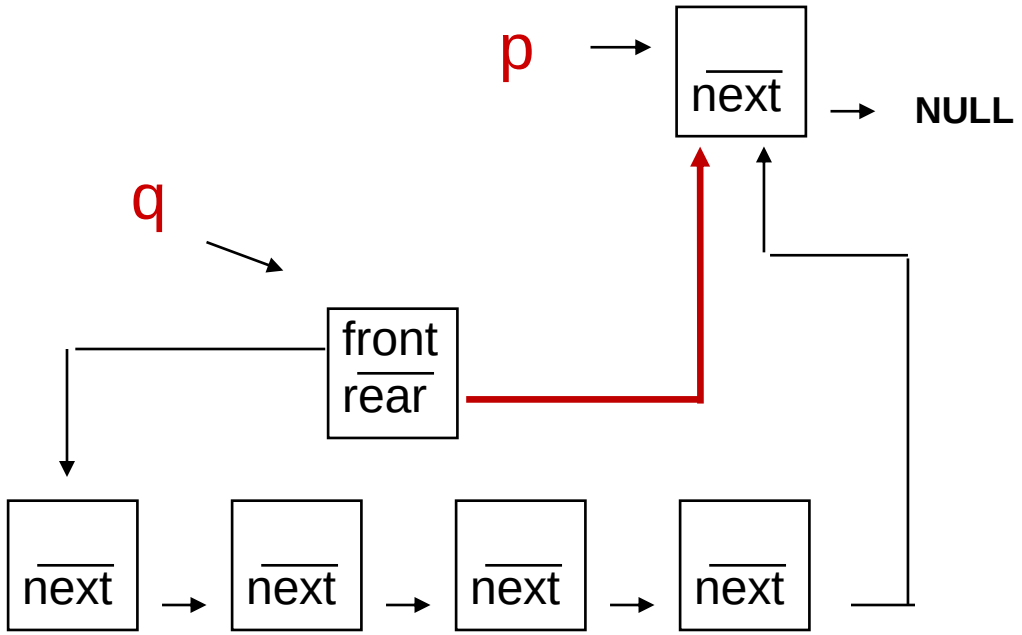




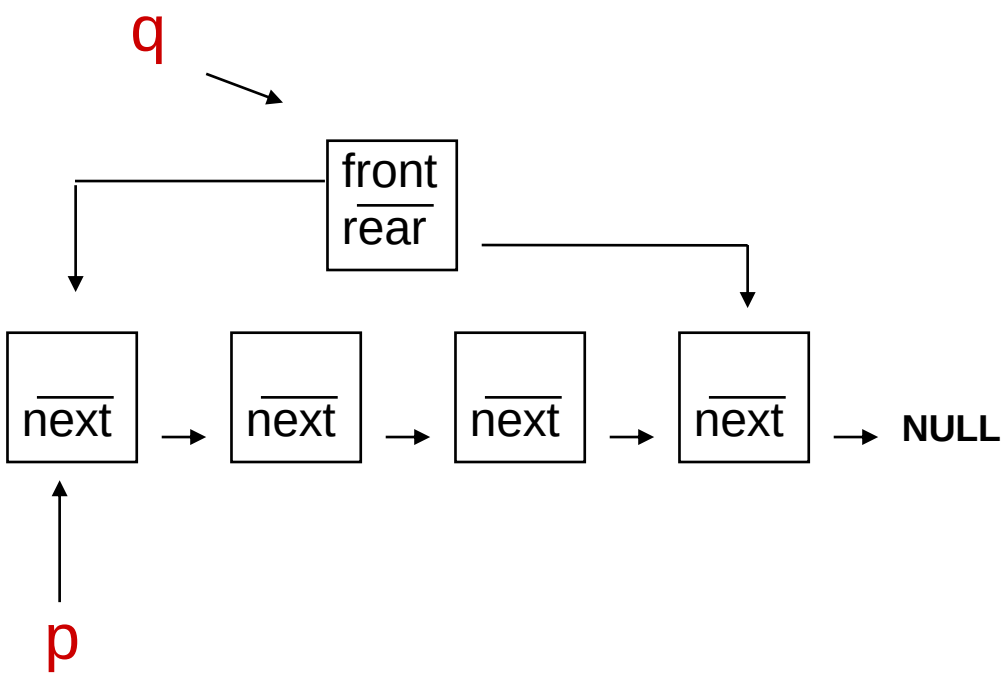
```
typedef struct {
        Node *front, *rear;
} Queue;
```



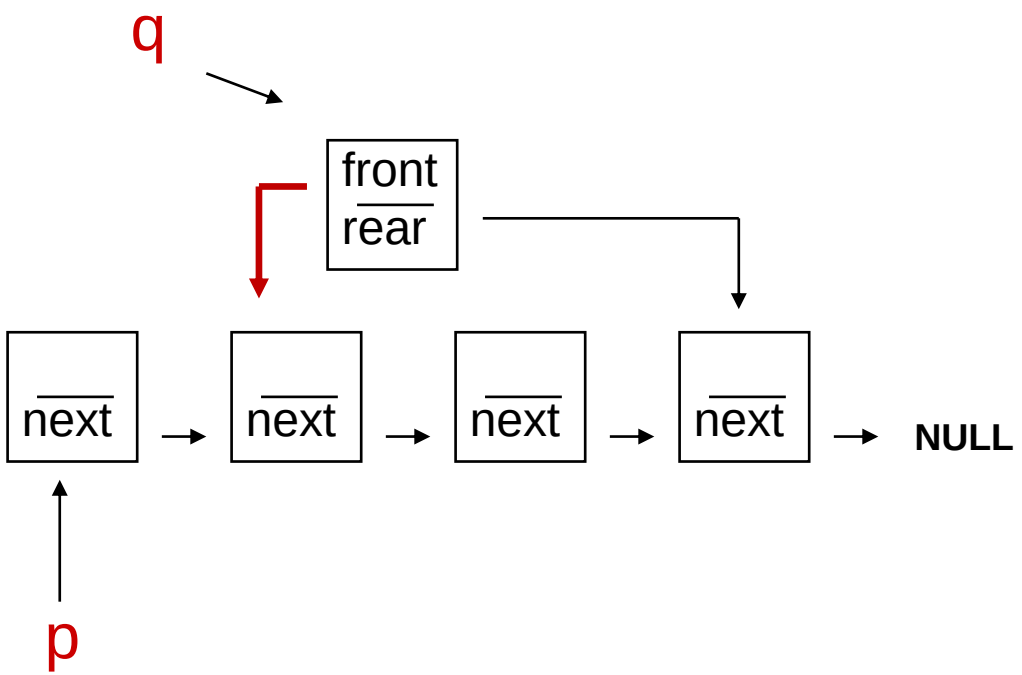
q->rear->next = p;



q->rear = p;



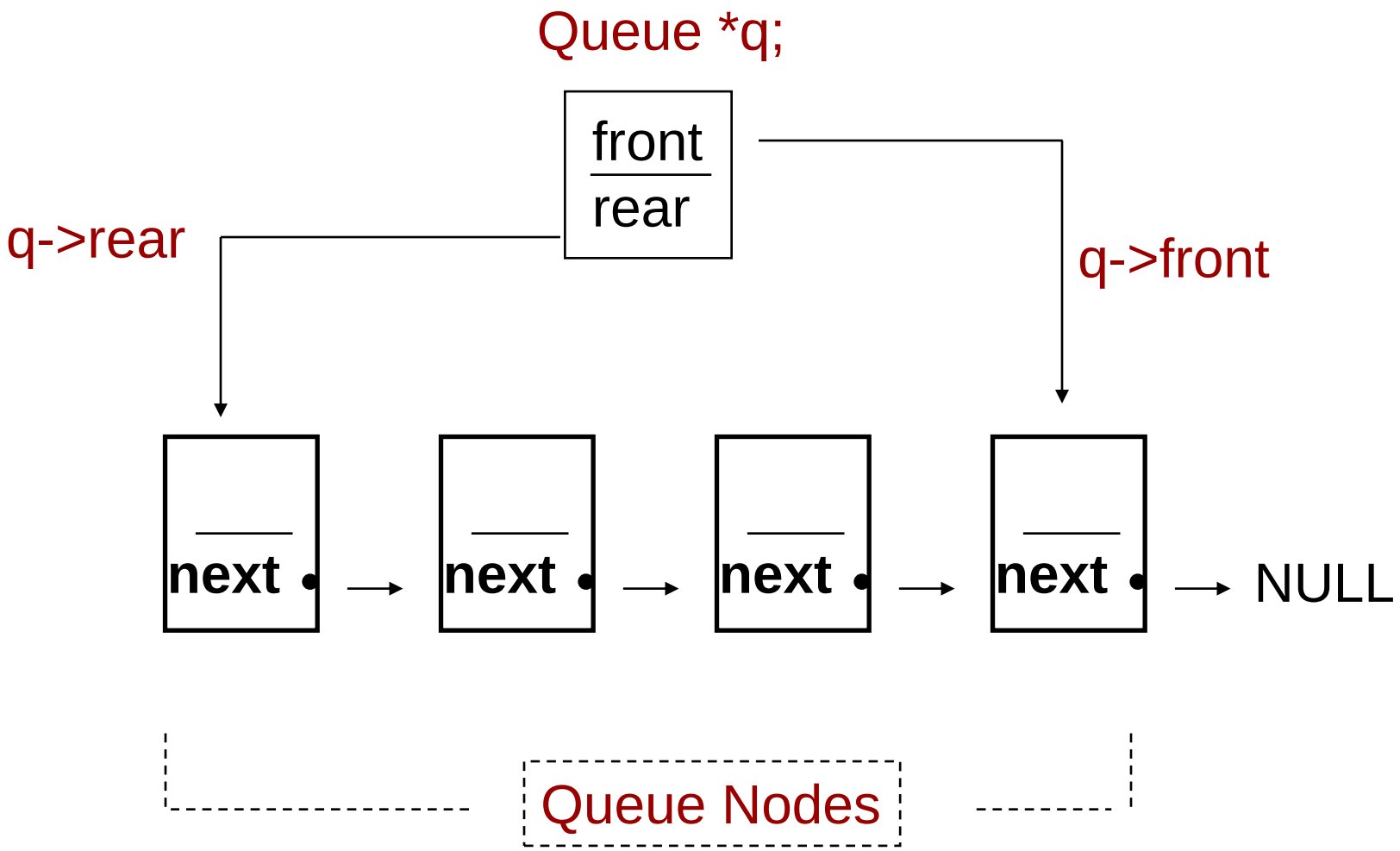
Deleting the node at the front of the queue



q->front = q->front->next;

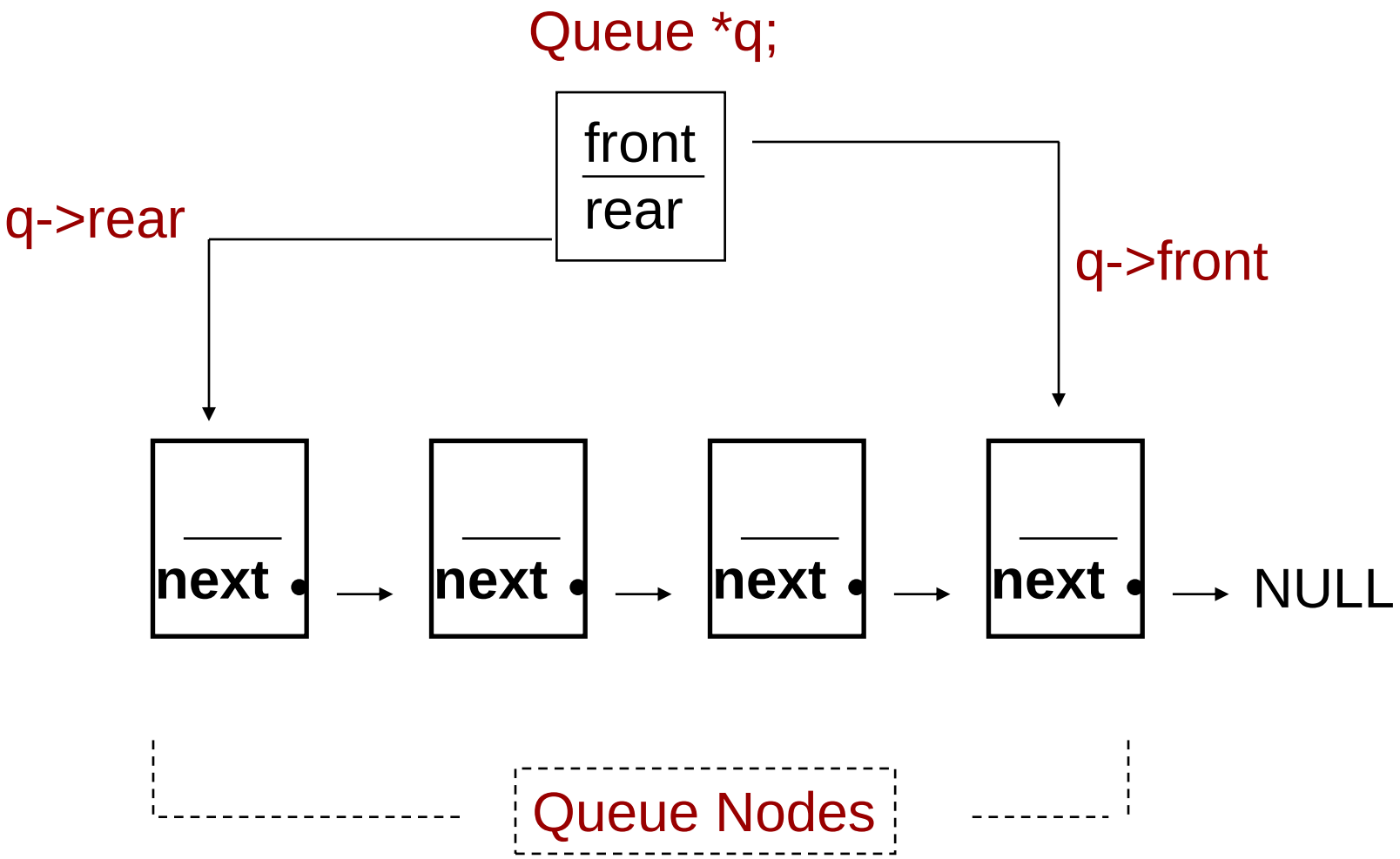
(or equivalently: q->front = p->next;)

Will changing the which end of the linked list is the rear and which is the front affect the complexity compared to what we examined before?



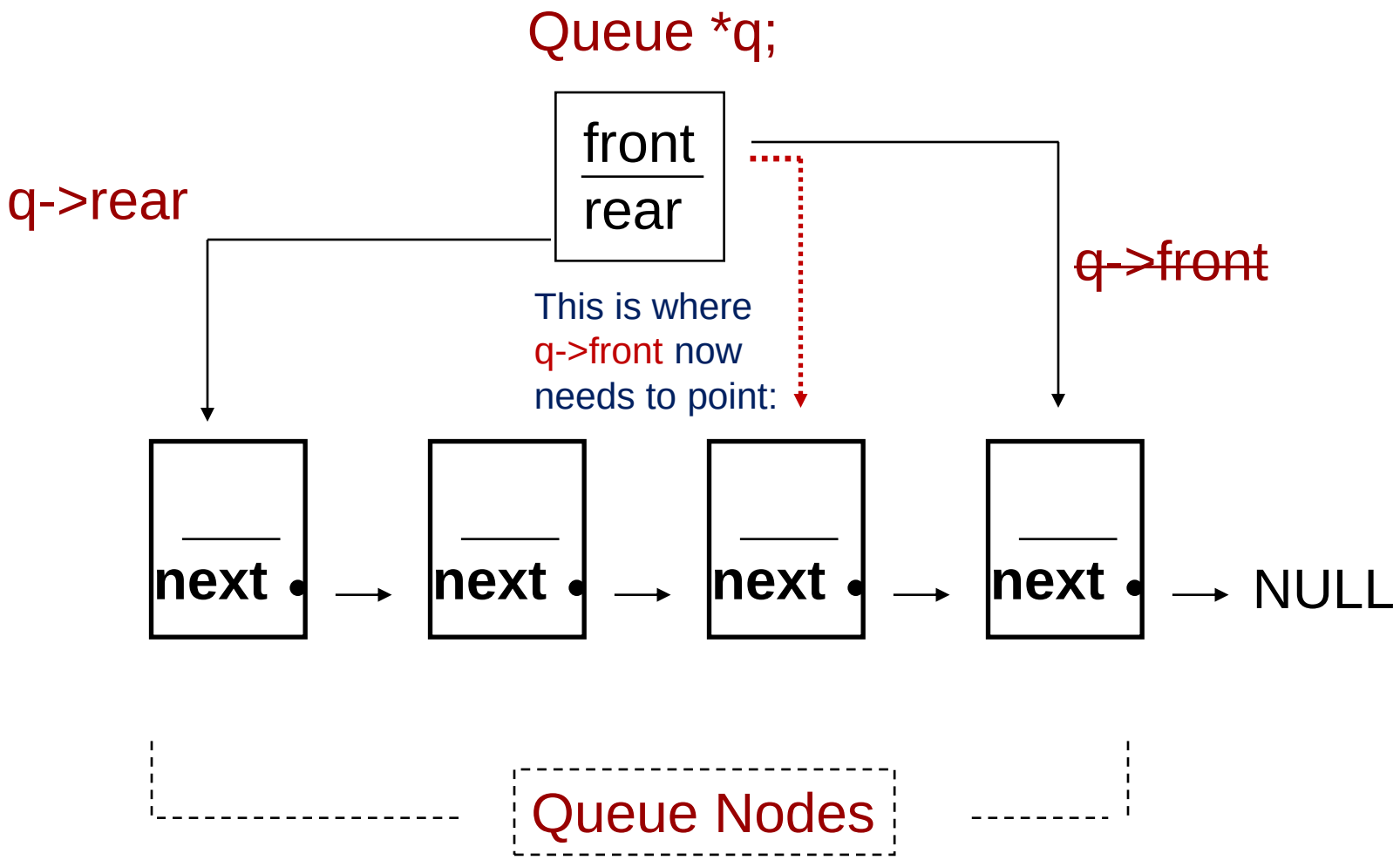
Will changing the which end of the linked list is the rear and which is the front affect the complexity compared to what we examined before?

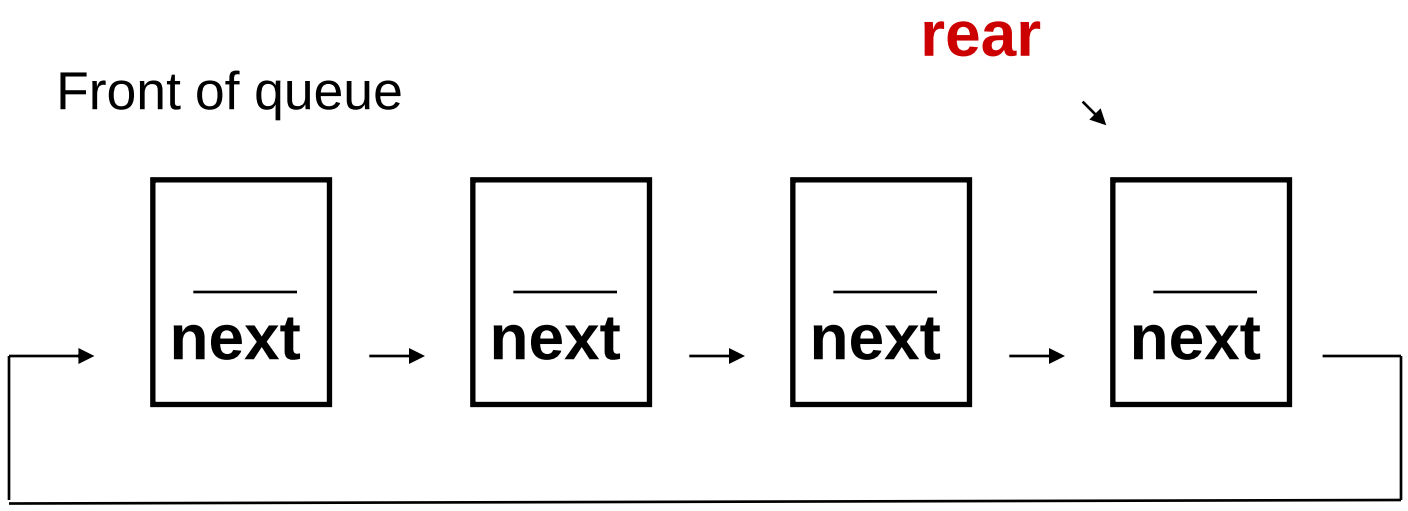
Yes, dequeue now *cannot* be performed in O(1) time. Why?



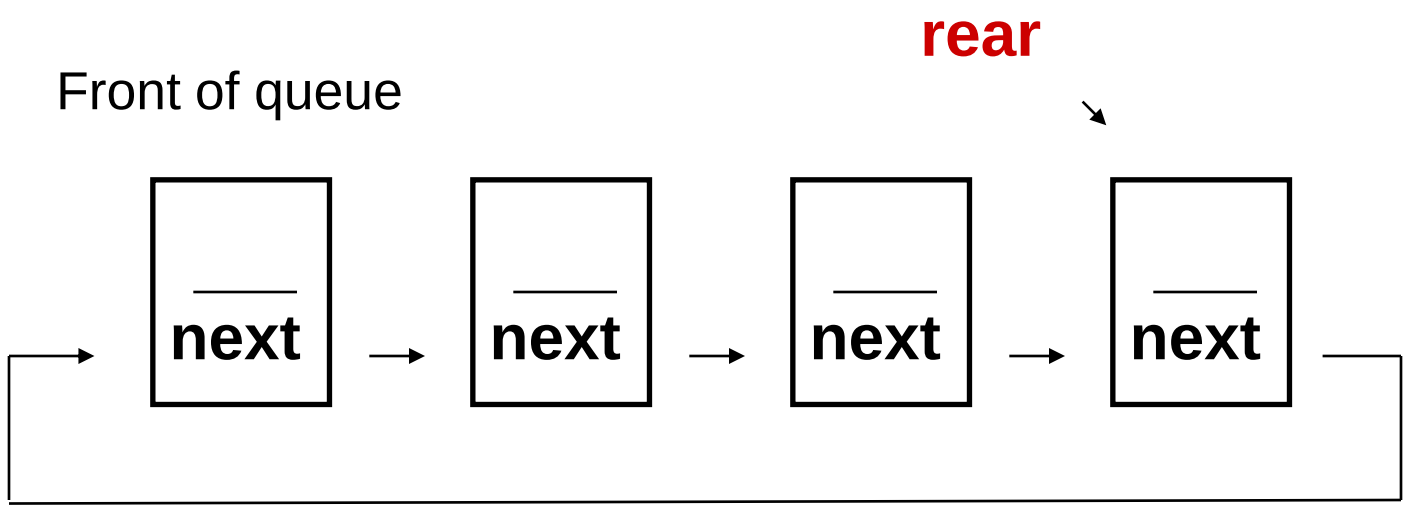
Will changing the which end of the linked list is the rear and which is the front affect the complexity compared to what we examined before?

Yes, dequeue now cannot be performed in O(1) time. Why?

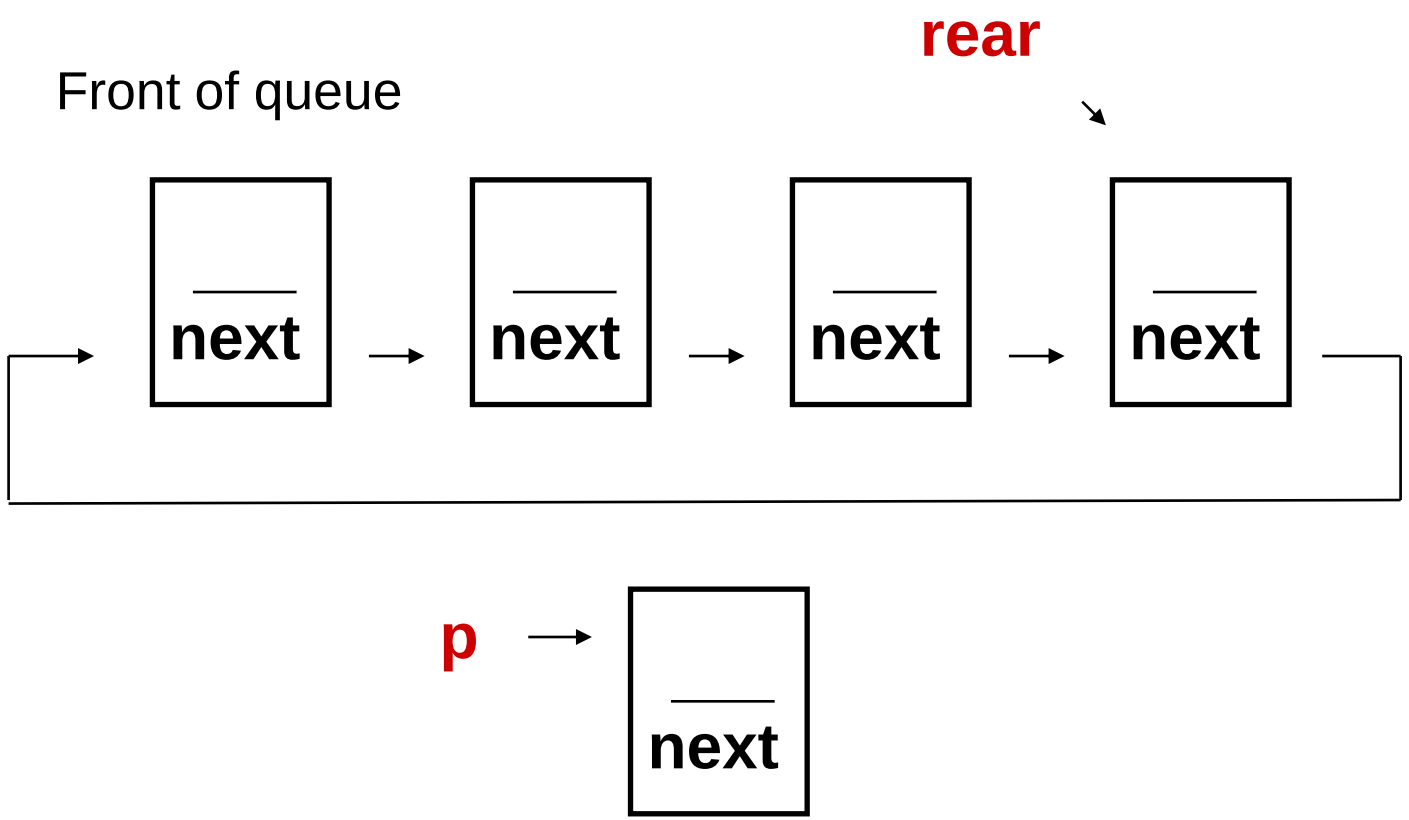




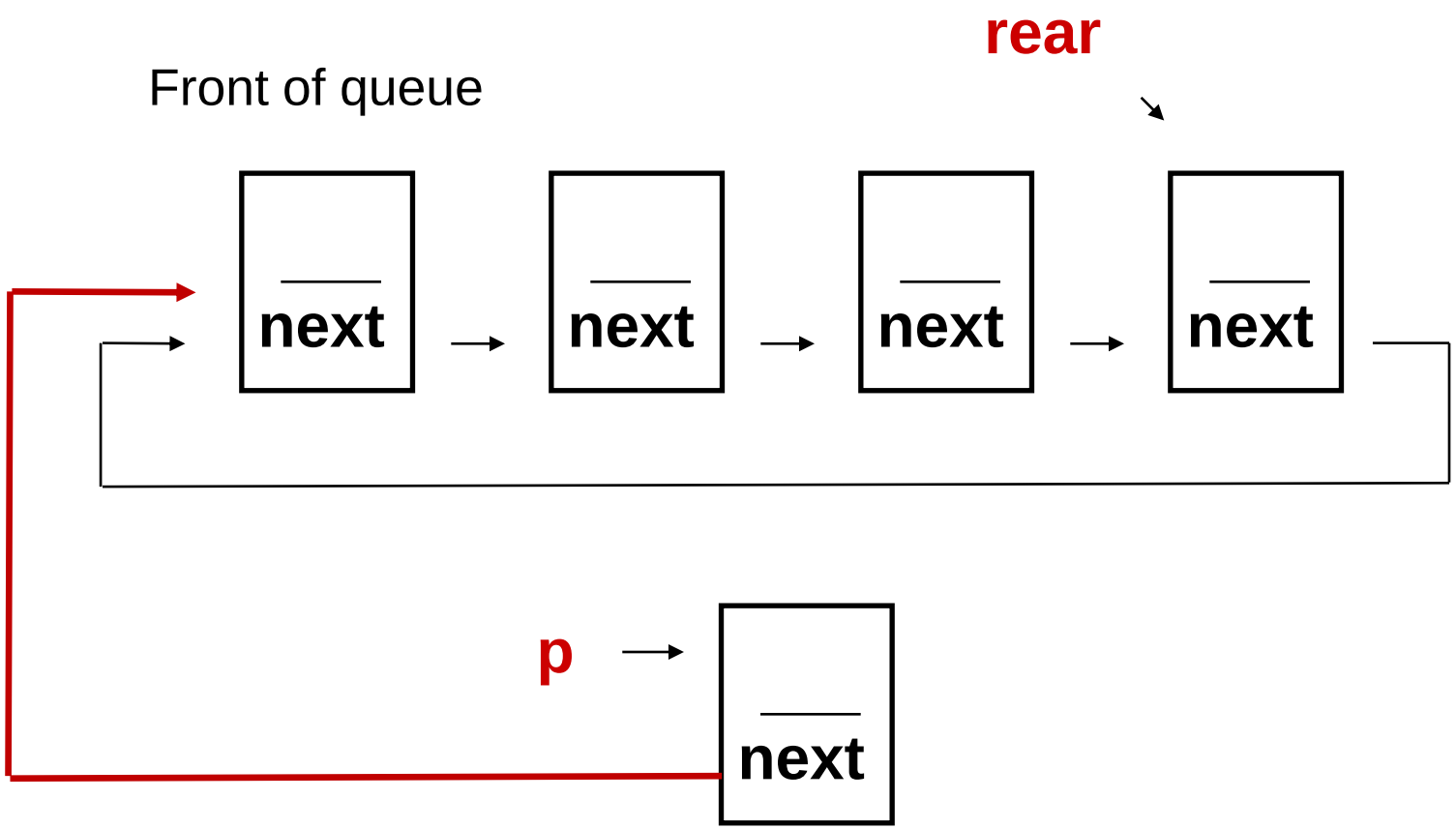
It is also possible to implement a queue with only one pointer and achieve the same efficiency as was achieved by maintaining separate front and rear pointers. However, to do so requires a different data structure: a *circular linked list*.



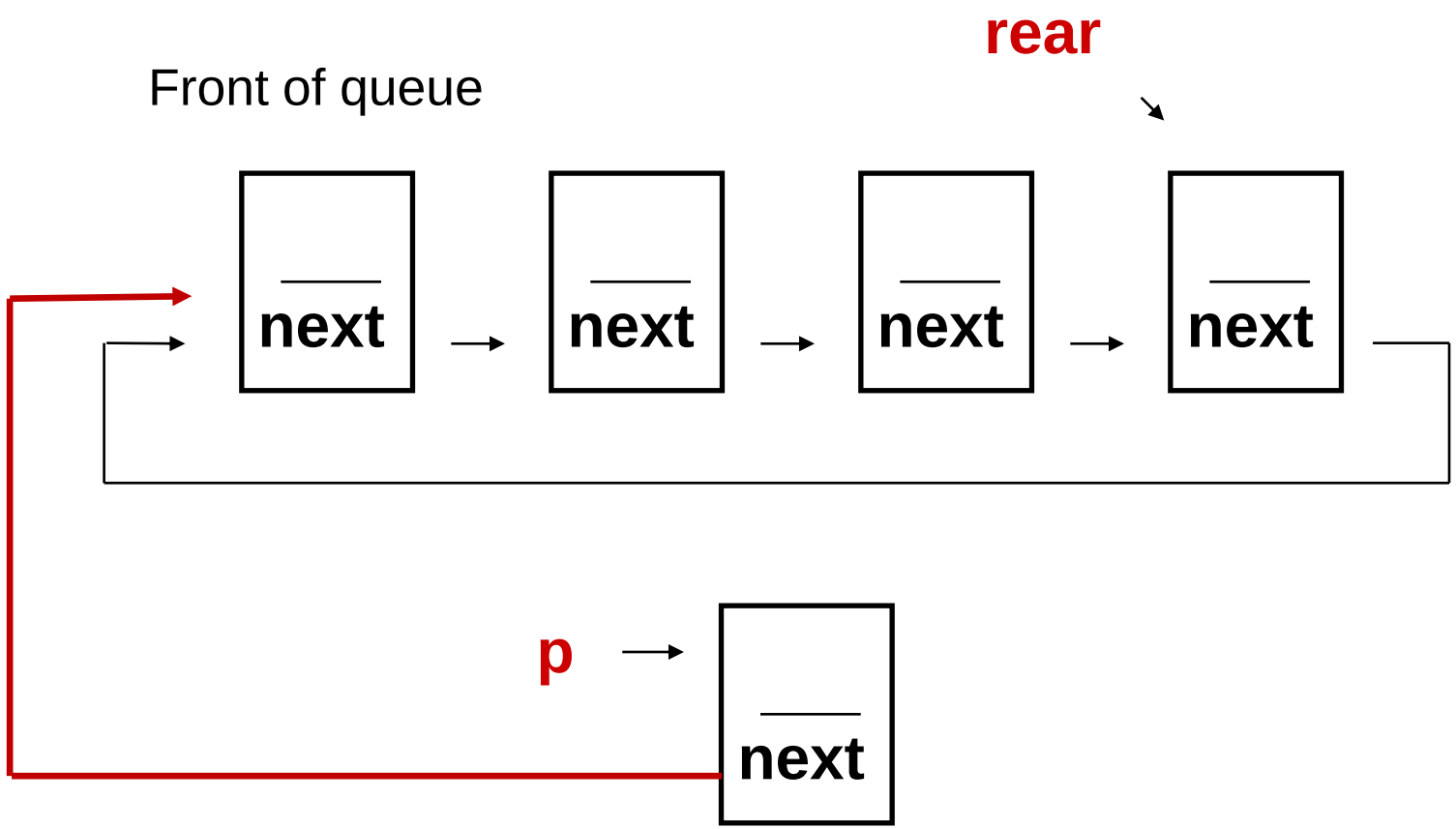
Note that the “rear” pointer provides O(1) access to the rear of the queue, and “rear->next” gives O(1) access to the front of the queue.



Now we need to figure out how to insert a node, pointed to by p, into a circularly linked list.

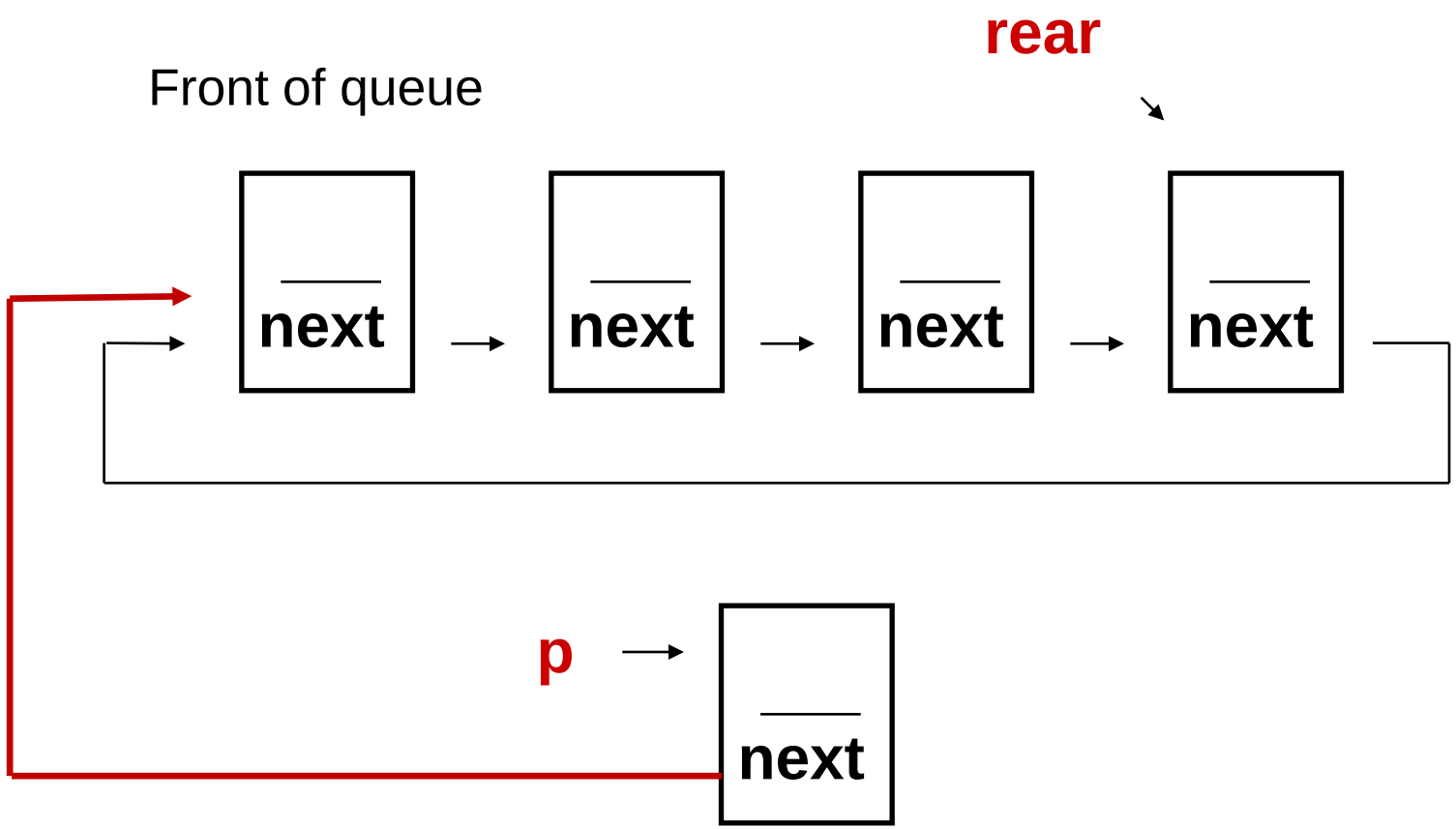


First we set p->next to rear->next. Remember, we eventually want this node to be at the rear of the queue.



First we set p->next to rear->next.

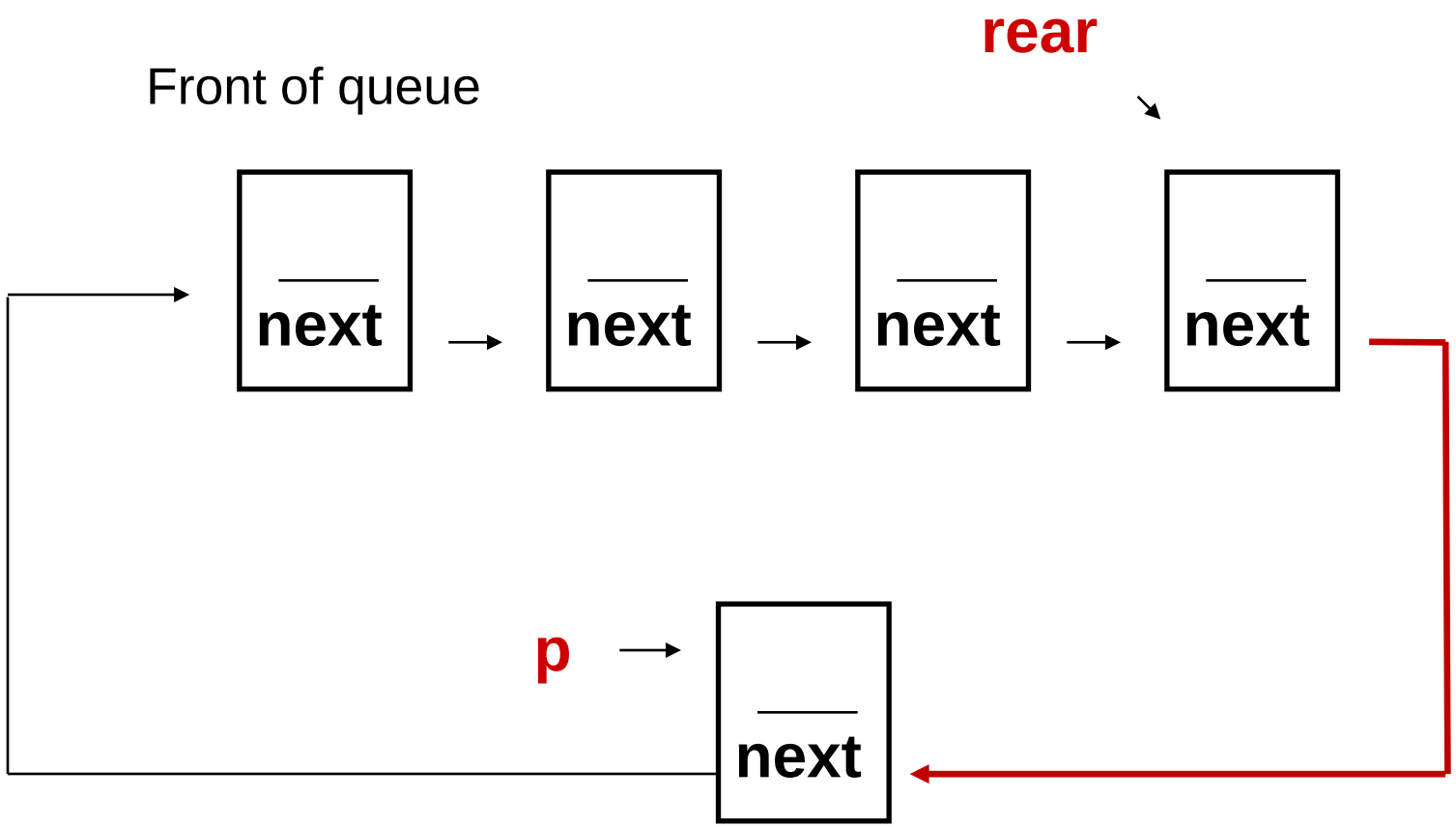
QUESTION: Under what circumstance will the above assignment not be the correct first step?



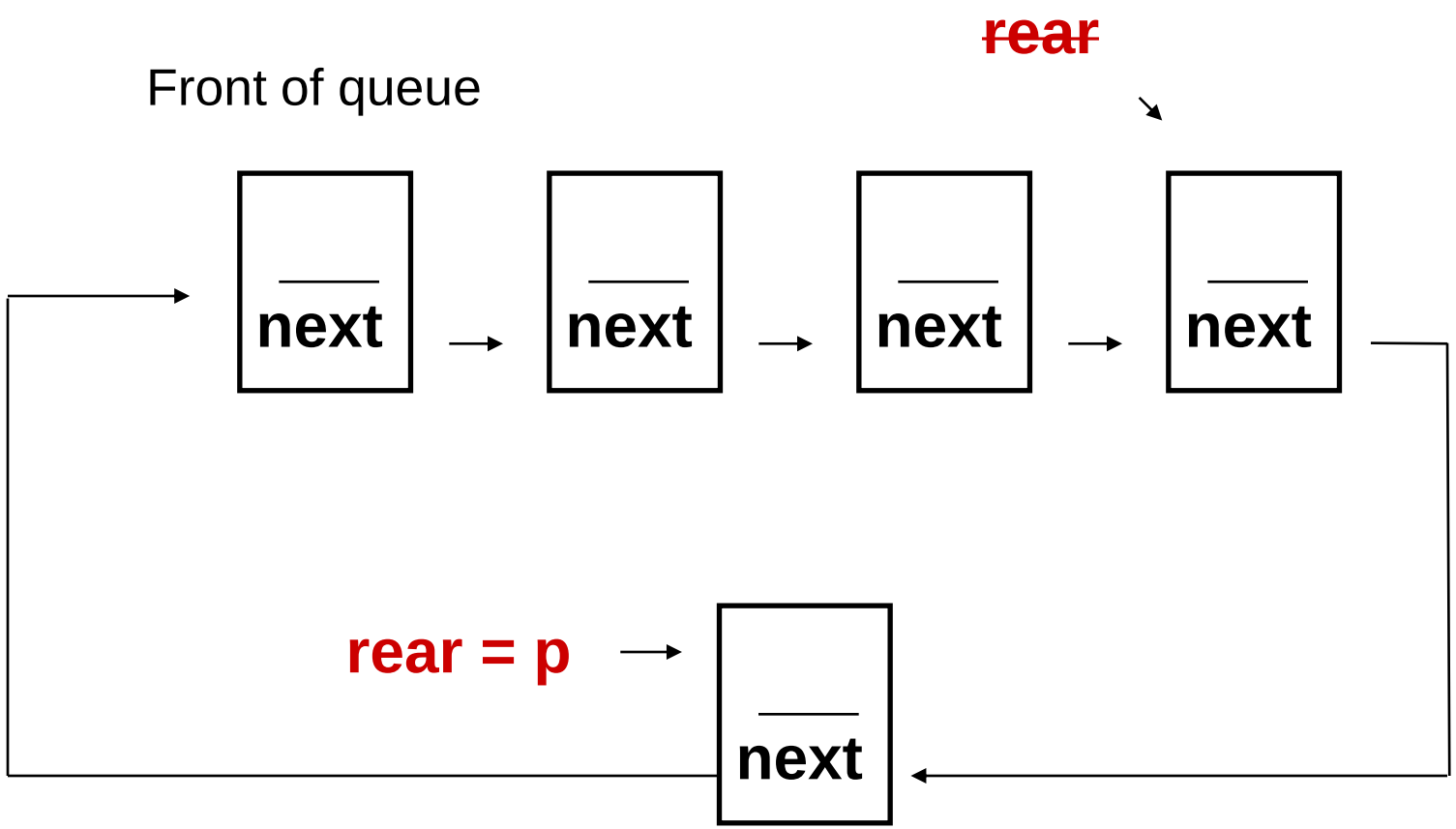
First we set p->next to rear->next.

The above line will fail (seg fault) if rear is NULL.

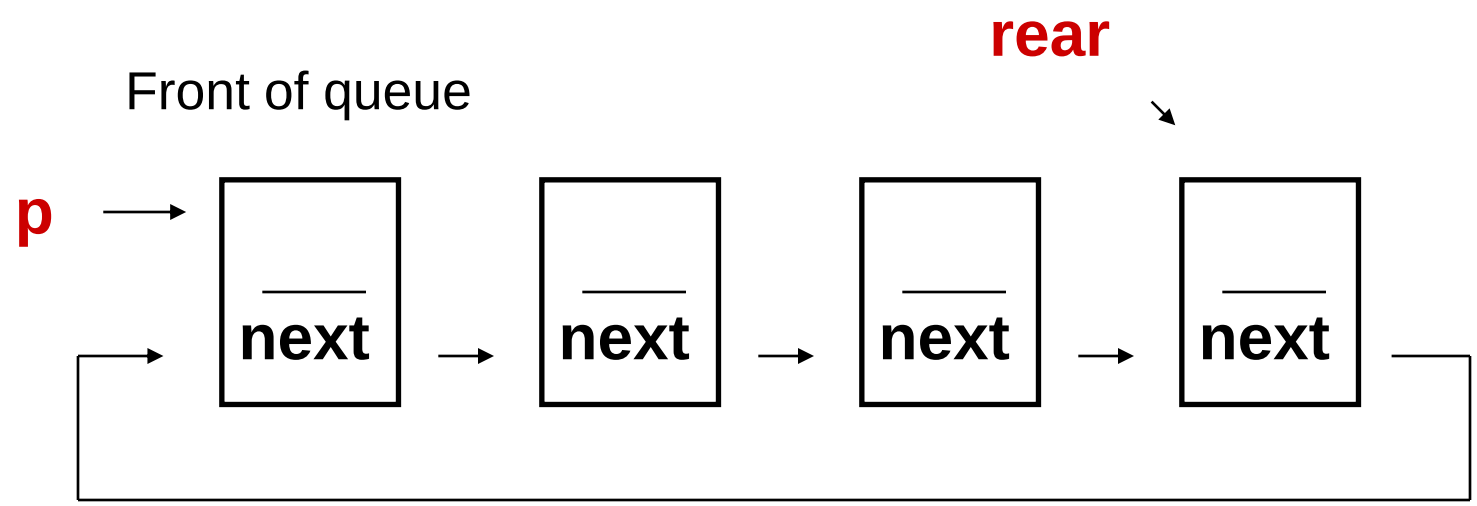
(Use of a dummy node simplifies by eliminating this special case.)



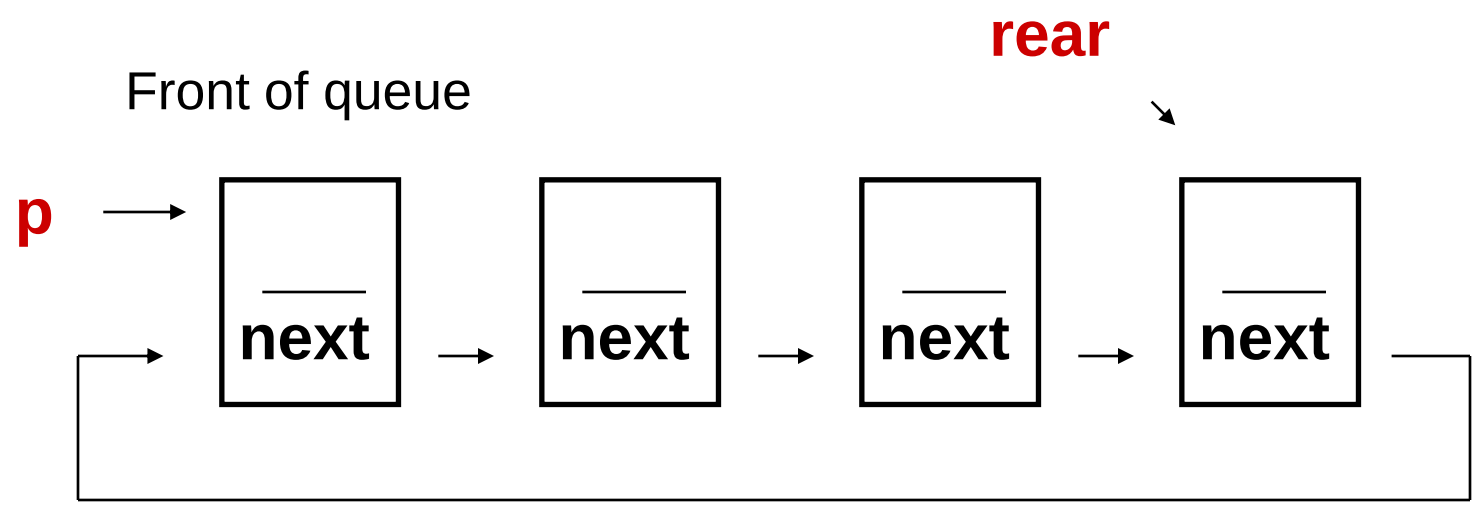
Then we set rear->next to p. Note that right now p points to the node at the *front* of the queue.



Once we set rear equal to p, the new node is now at the rear location in the queue.

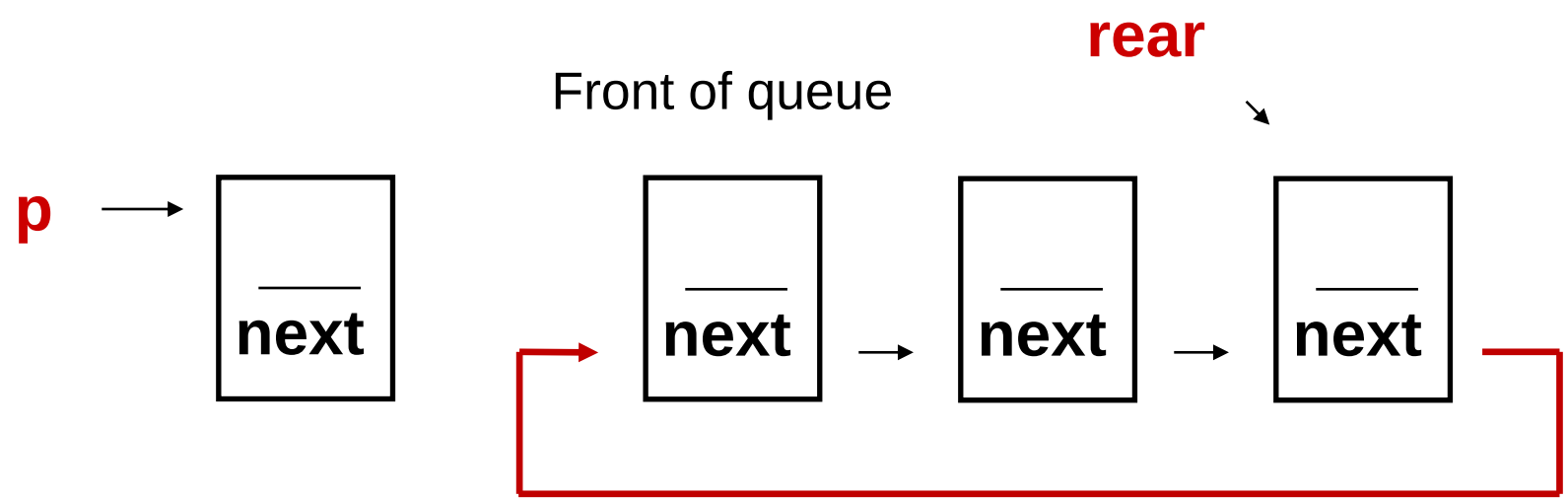


To remove (dequeue) the first node in the queue, we begin by setting p equal to rear->next, which is the front of the queue.

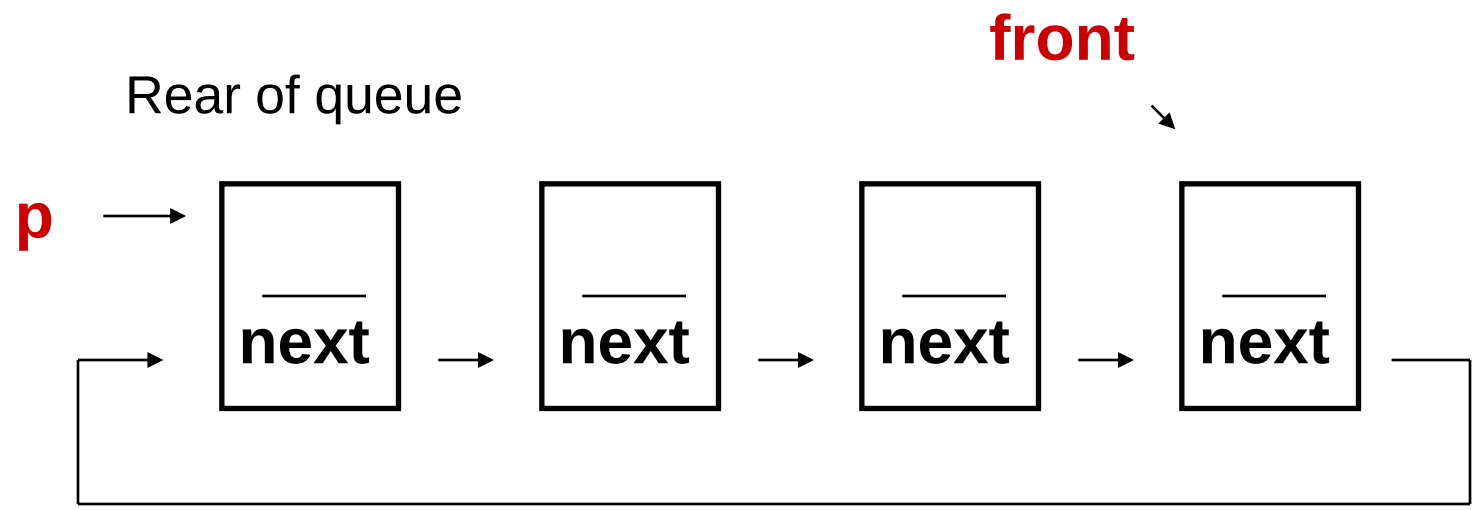


To remove the first node in the queue, we begin by setting p equal to rear->next, which is the front of the queue.

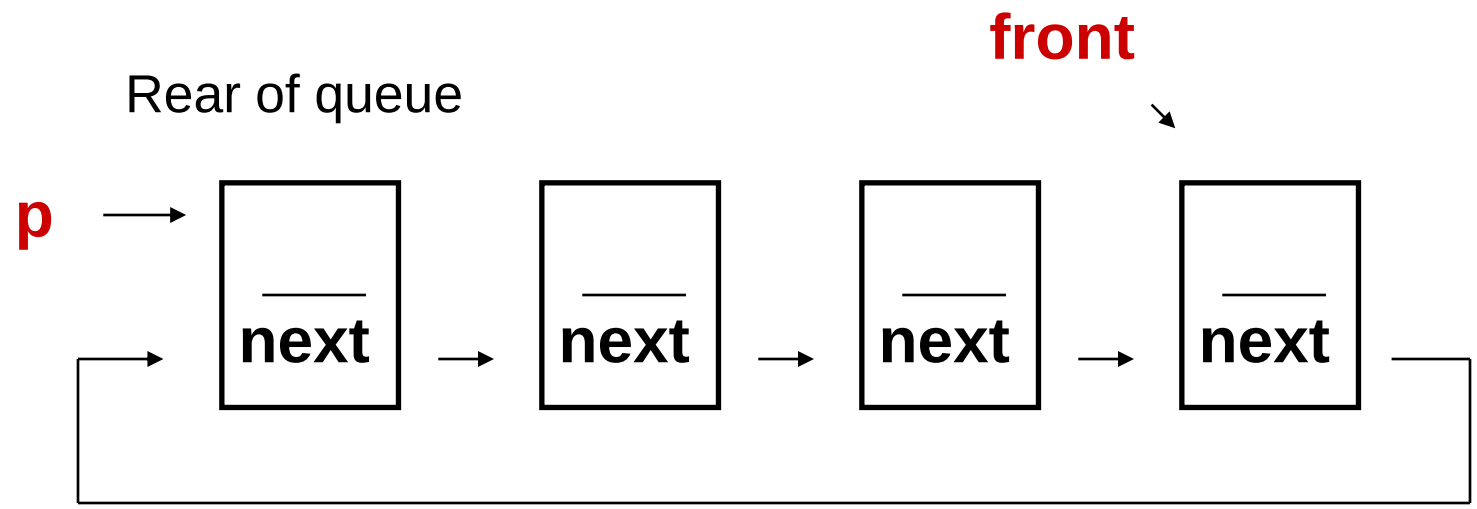
Note that the case in which the queue is empty must also be handled appropriately.



Then we set rear->next equal to p->next. This cuts the node pointed to by p out of the list.



Is there a problem with the above depiction for implementing a queue with a linked list?



Is there a problem with the above depiction for implementing a queue with a linked list?

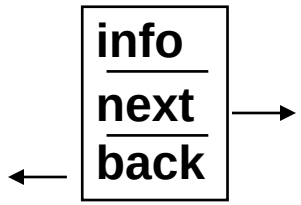
Yes, after a dequeue front will need to point to the node that precedes it. How can we access it?

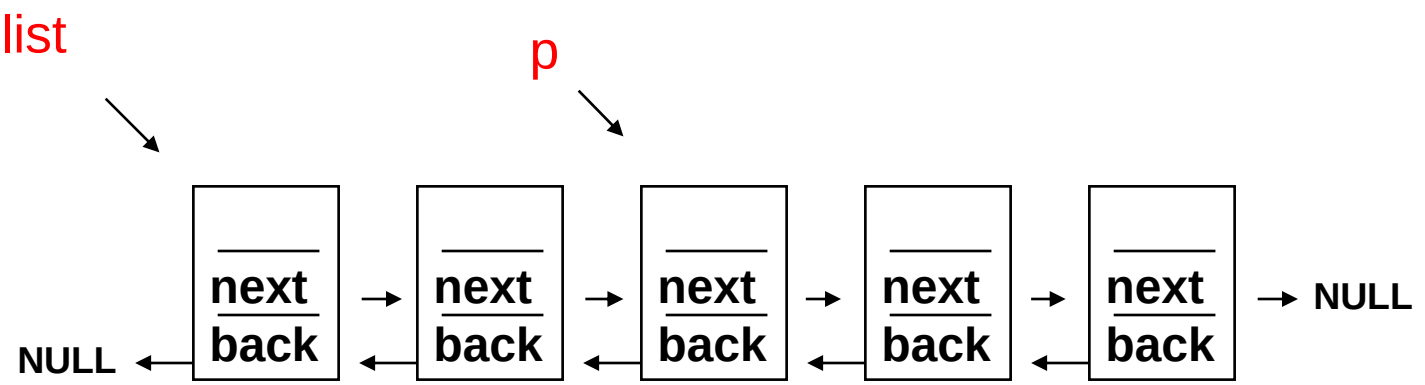
A seemingly more versatile data structure is the ***doubly-linked list***.

Each node of a doubly-linked list contains a pointer to its predecessor and a pointer to its successor in the list.

A doubly-linked list can efficiently support insertions and deletions on either end of the list.

Therefore, a doubly-linked list can simultaneously support both stack and queue operations efficiently. This is not possible with a simple linked list.

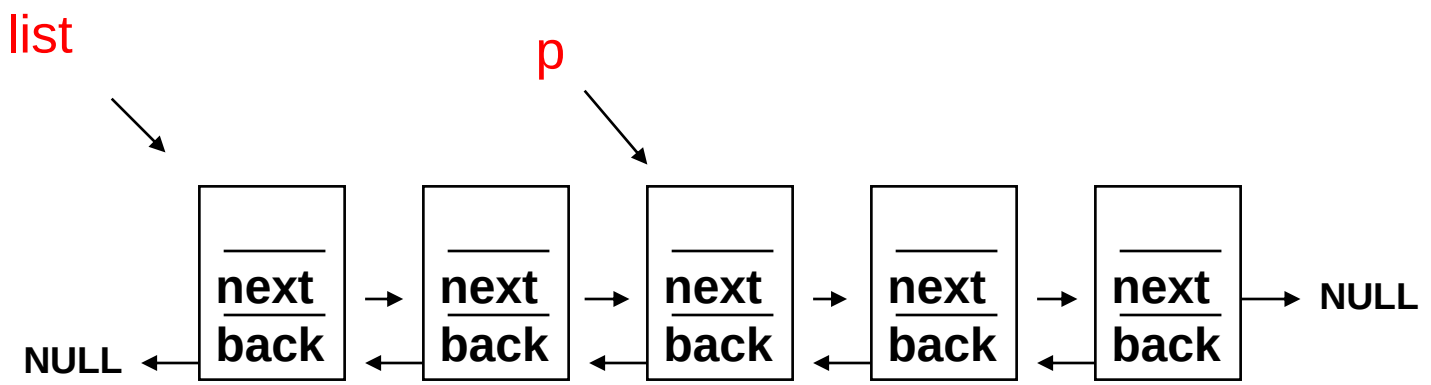




Consider the deletion of a particular node, pointed to by **p**, from a doubly-linked list.

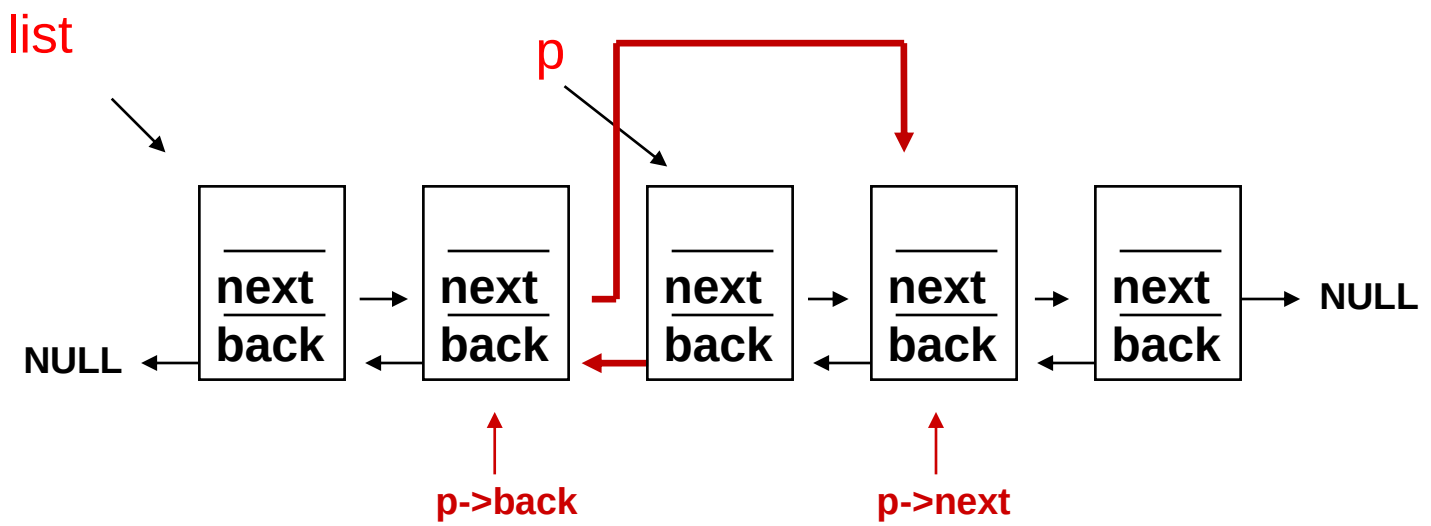
In a simple linked list we would have to have a pointer to the node *preceding* the node we want to delete.

In a doubly-linked list, we just need a pointer to the node we want to delete.



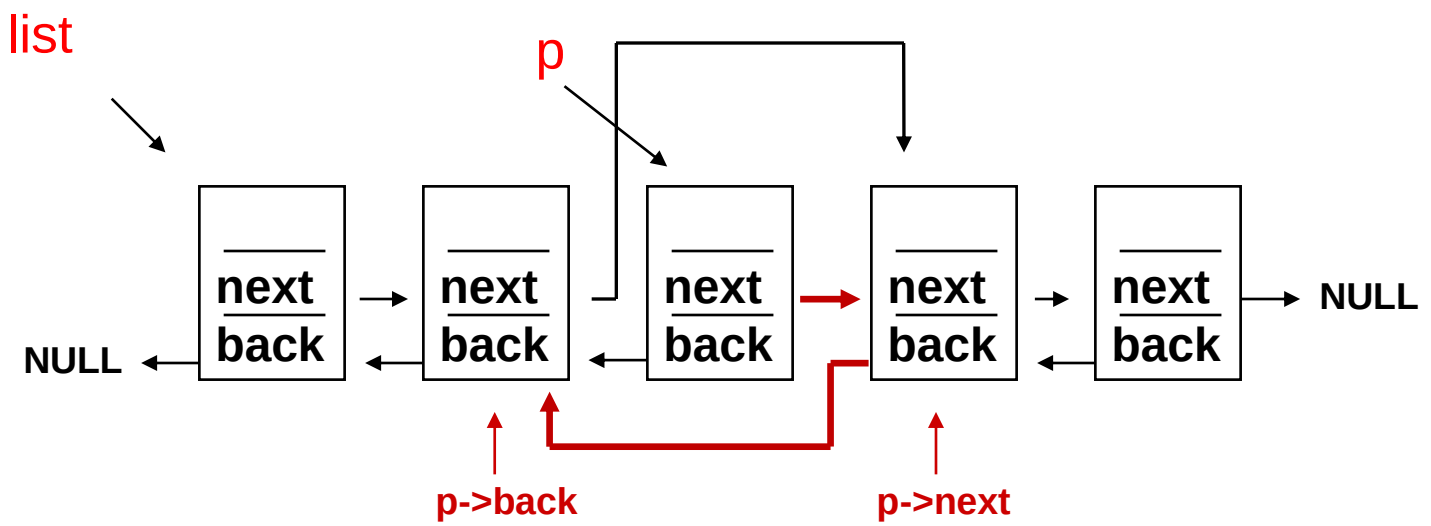
The following is all that's necessary to delete the node pointed to by p:

```
p->back->next = p->next;
p->next->back = p->back;
```



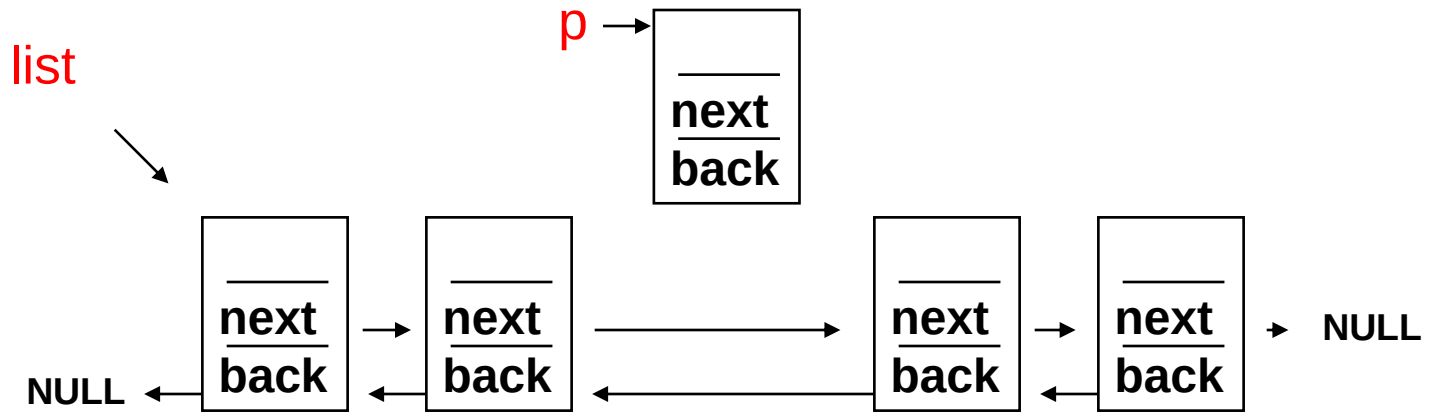
p->back->next = p->next;

p->next->back = p->back;



p->back->next = p->next;

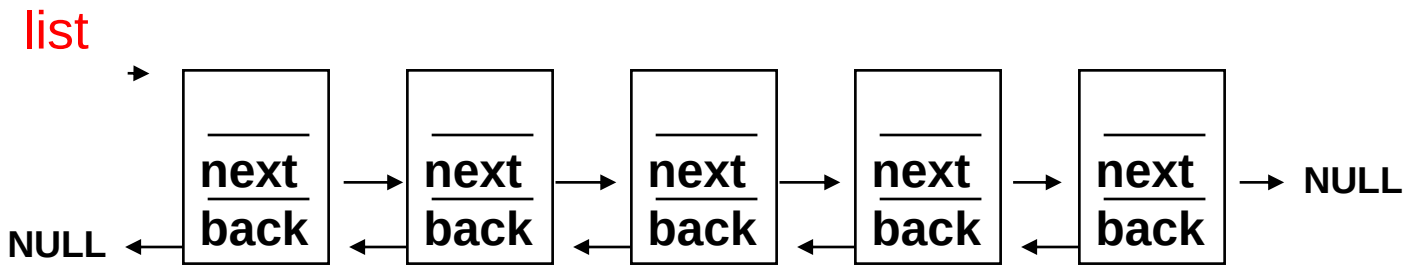
p->next->back = p->back;



p->back->next = p->next;

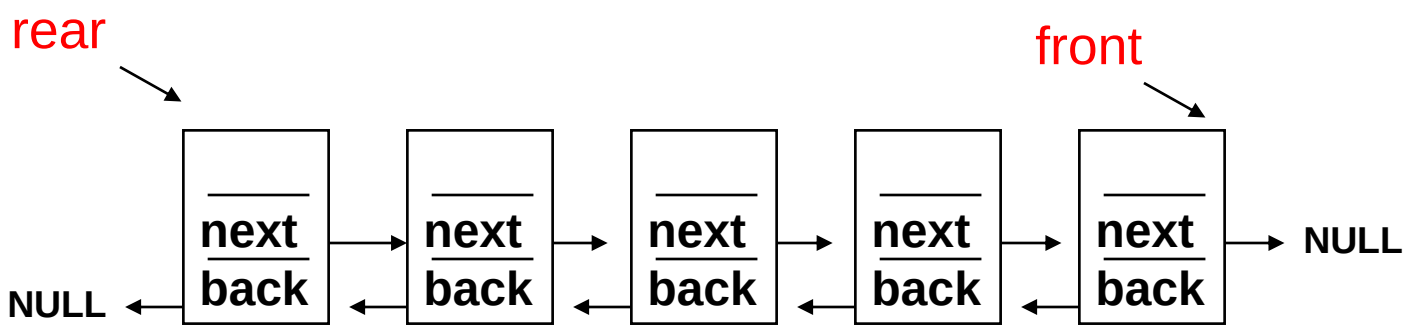
p->next->back = p->back;

Done.

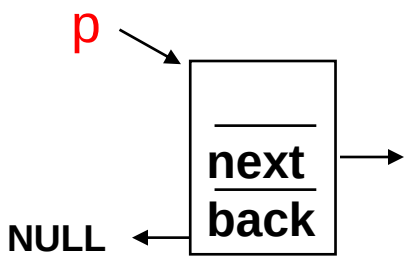


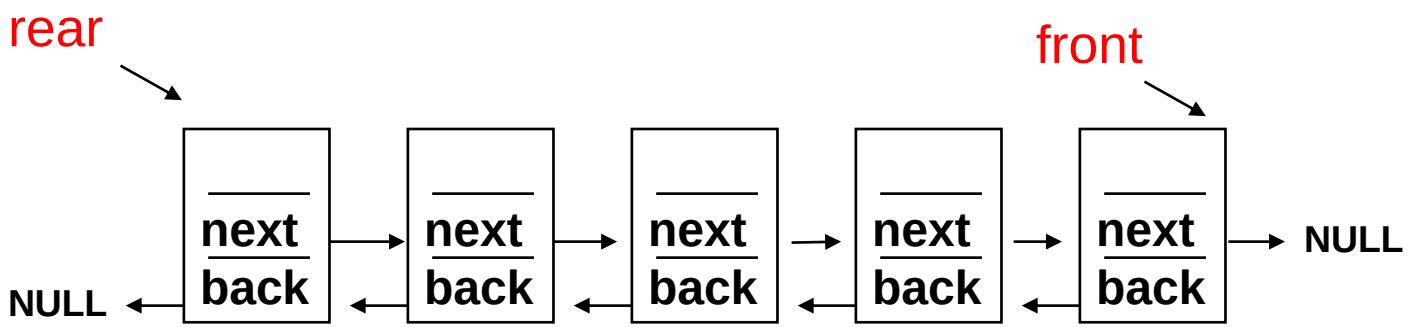
Writing a routine to delete nodes with a specified key value is straightforward with a doubly-linked list.

The only complications involve the special-case situations at the beginning and end of the list unless dummy nodes are used.

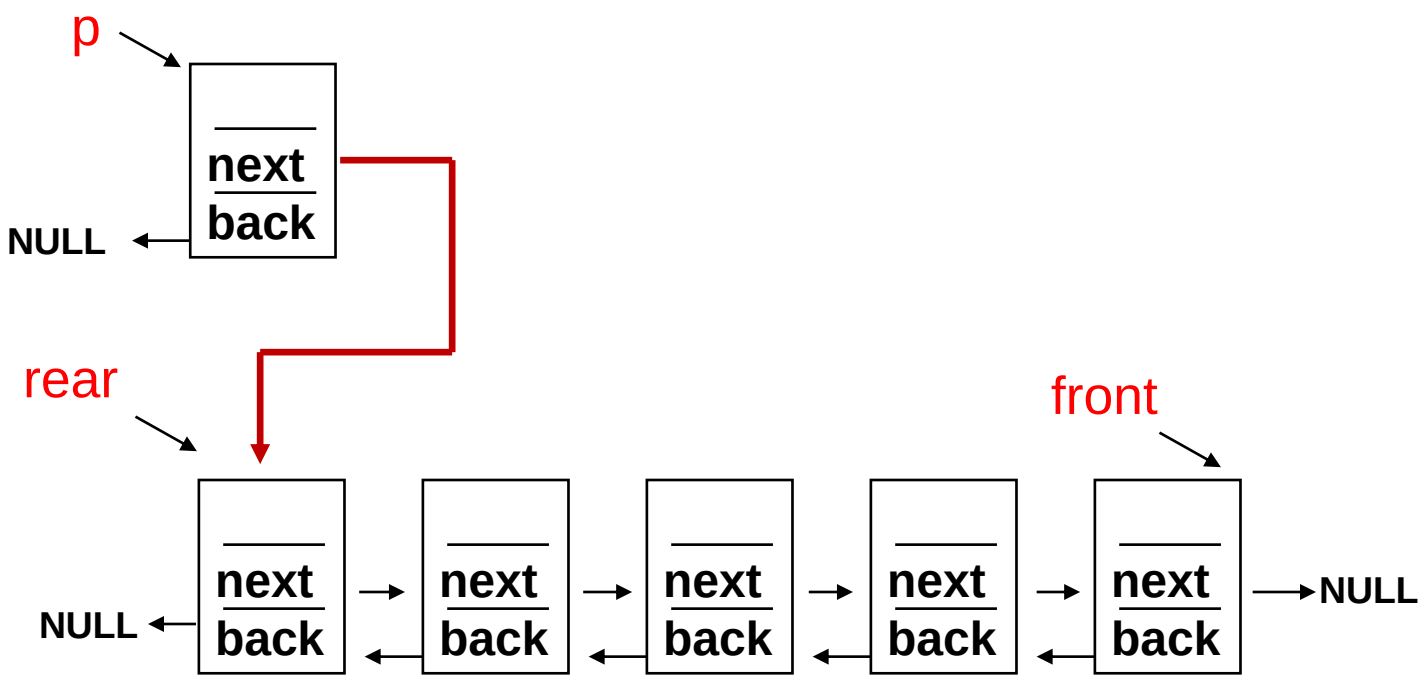


Note that with a doubly-linked list it is possible to efficiently support the enqueue and dequeue operations regardless of which end is defined as being the rear or the front.

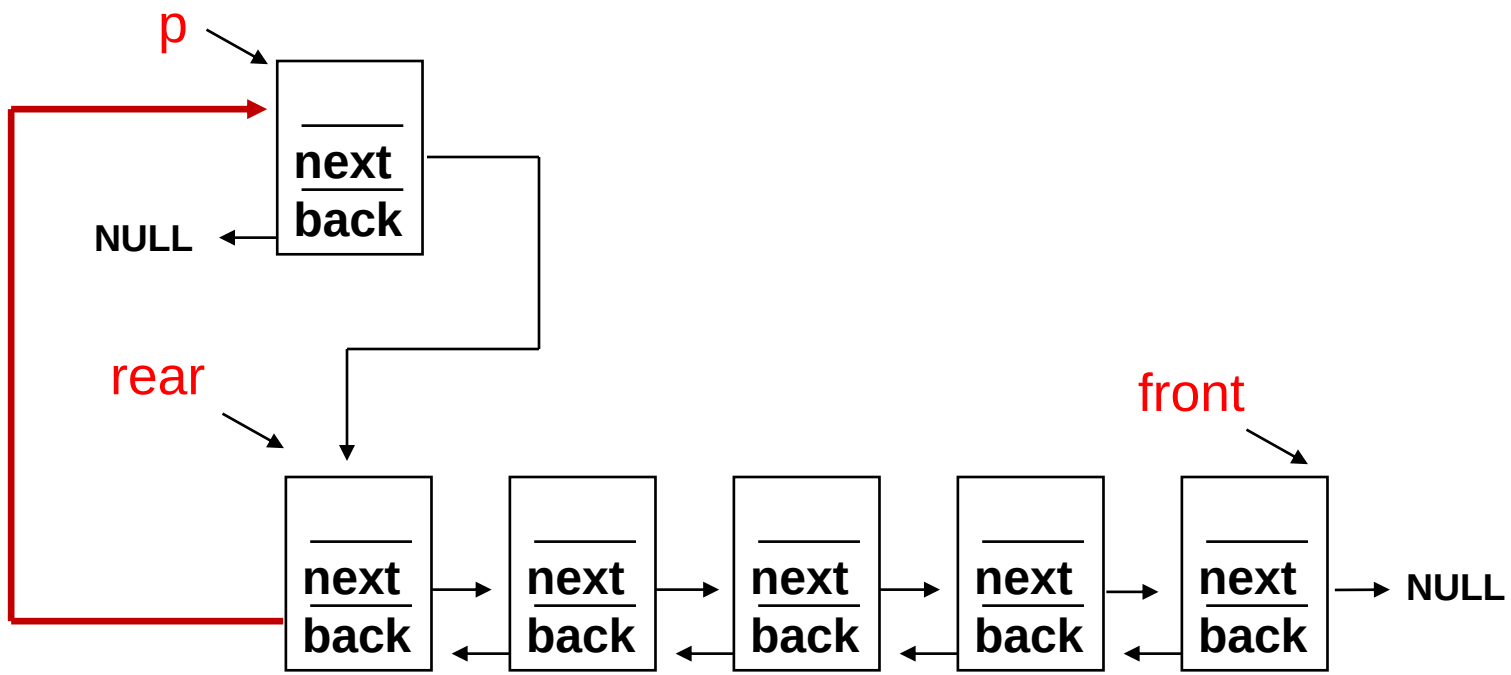




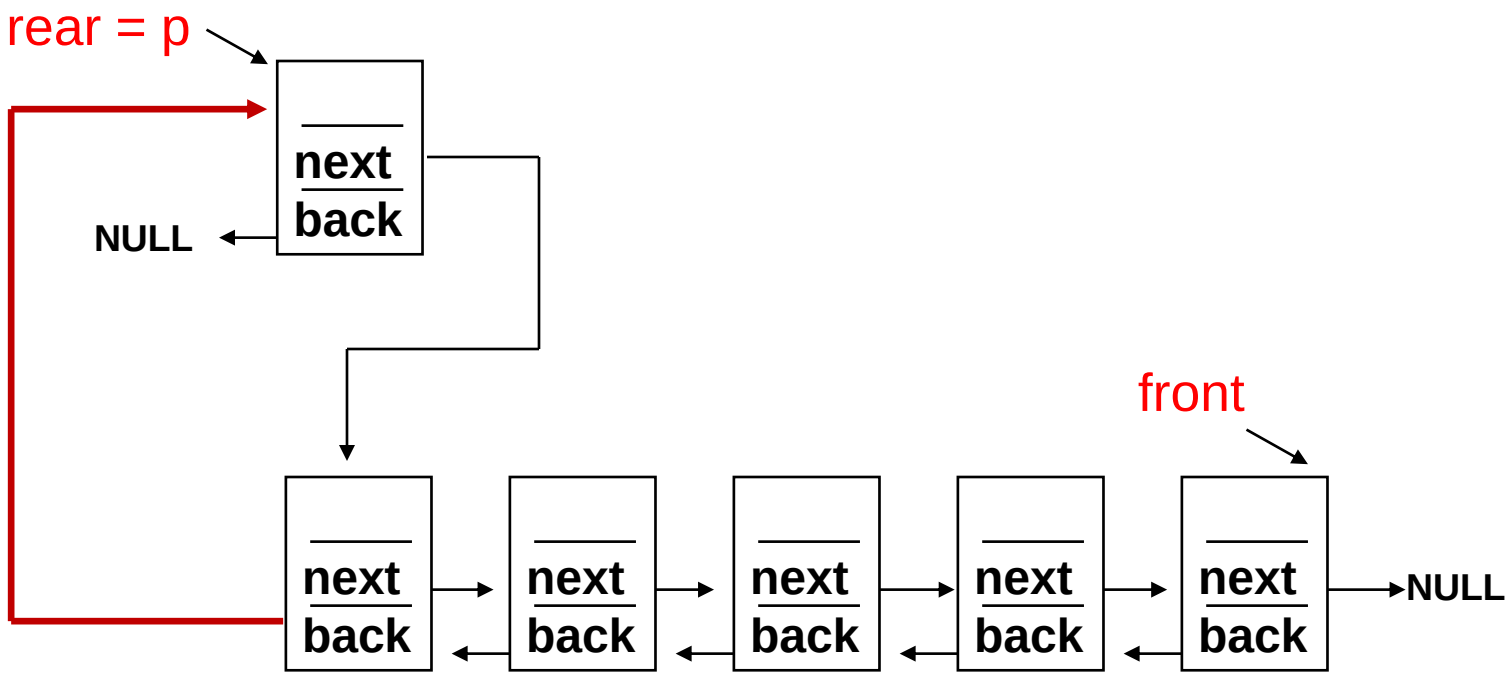
To enqueue…



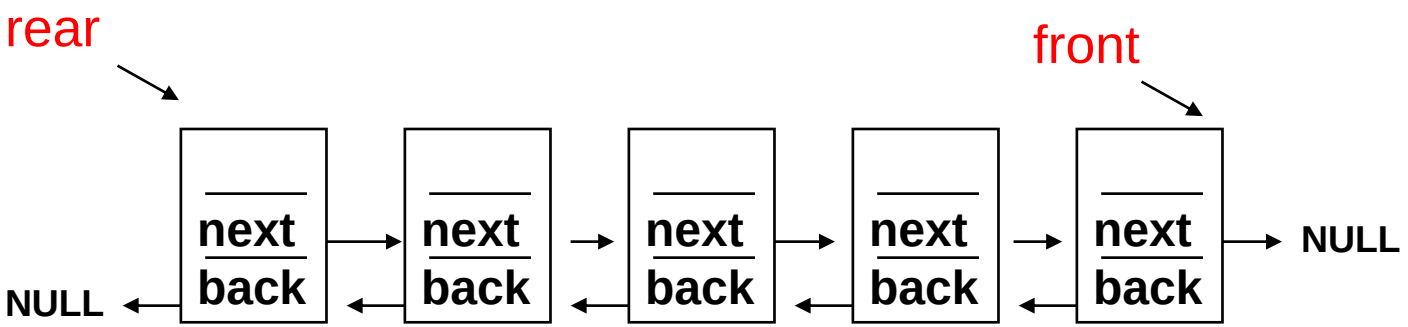
To enqueue, first set p->next to rear.



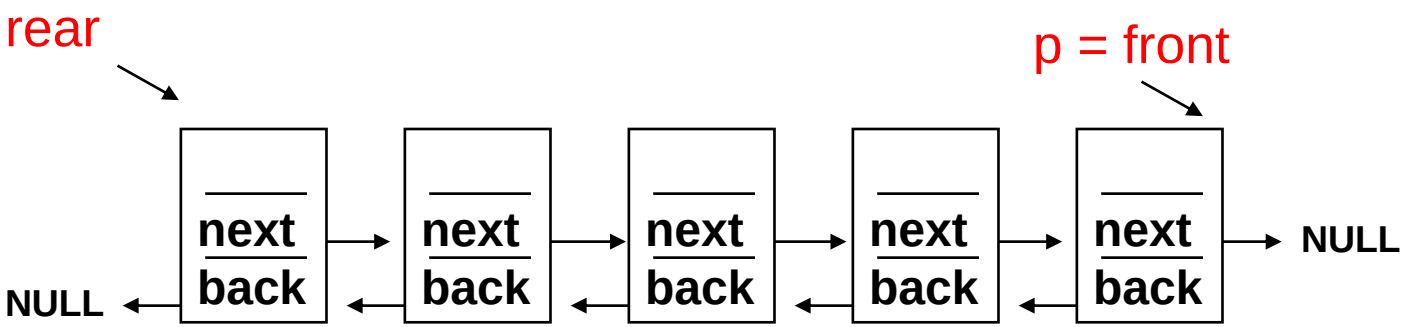
Then set rear->back to p.



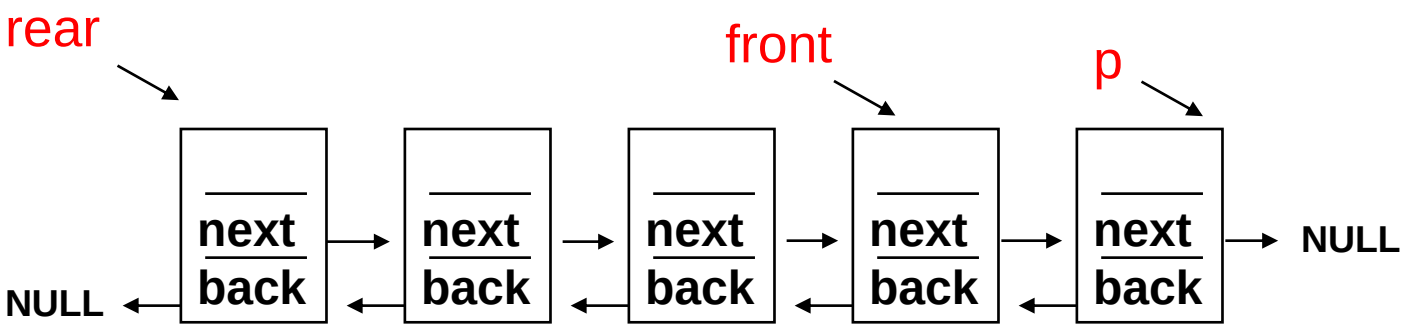
Then set rear to p.


rear
front
next
back
next
back
next
back
next
back
next
back
NULL
NULL

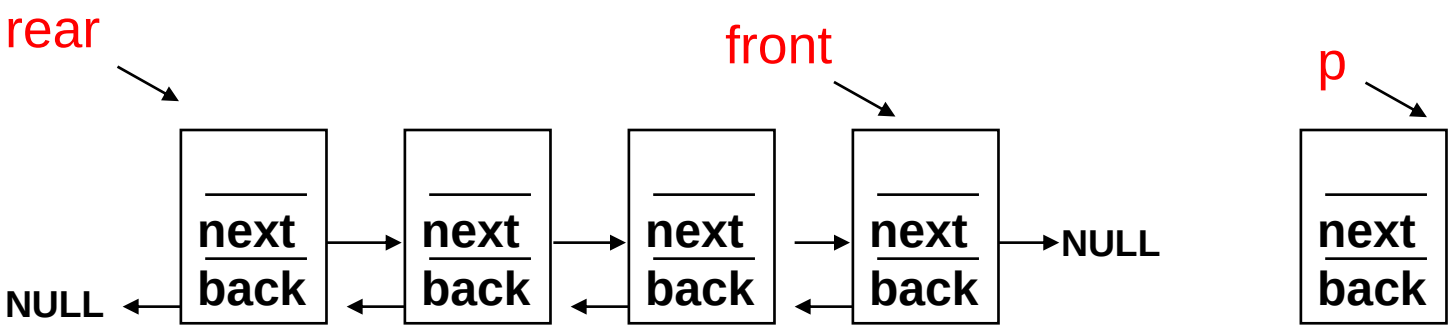
To dequeue…



To dequeue, assign a temporary variable p to front.

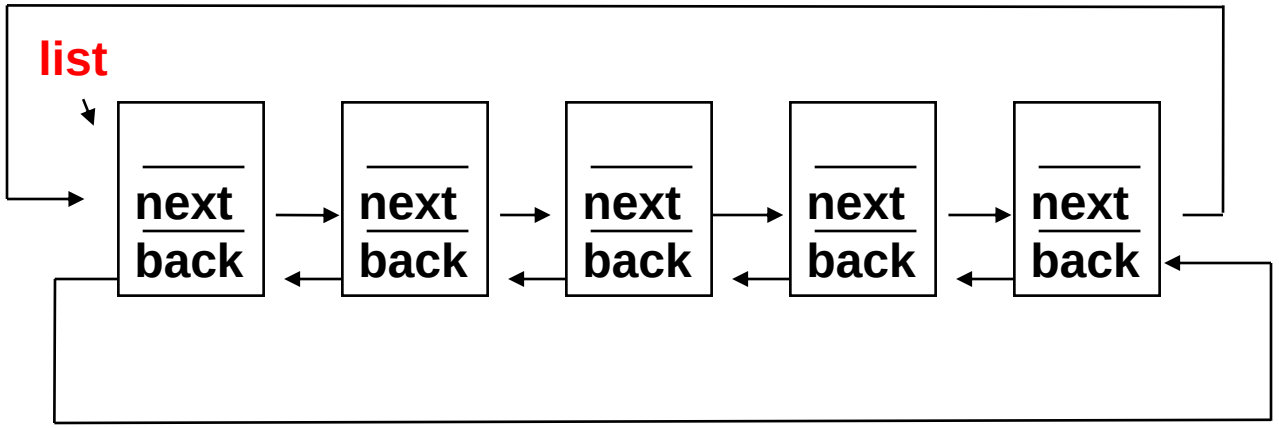


Then update the front pointer by setting front to front->back.



Then set front->next to NULL.

A ***circular*** **doubly-linked list** eliminates the special cases at the beginning and end of non-empty list:



Unfortunately, doubly-linked lists can lead to somewhat obscure code, especially when deletions are involved. However, a single dummy node can greatly simplify things.