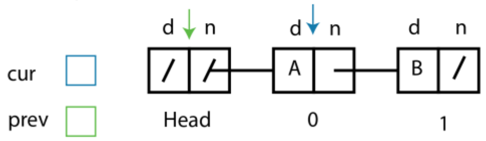
Homework logic for assignment one
Roger Benjume CSCD 300, SPRING 2022

public Object removeFirst(){}

Iteration 1.

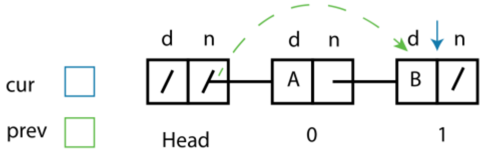
prev = this.head;   cur = this.head.next;



size = 2

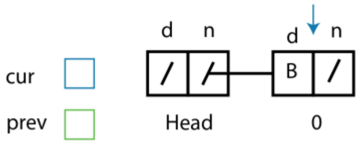
Iteration 2.

prev.next = cur.next;



size = 2

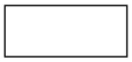
Result



return cur.data = A

this.size--;
size = 1

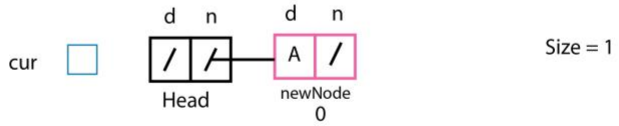
Edge Case: this.size == 0;
It's an empty list



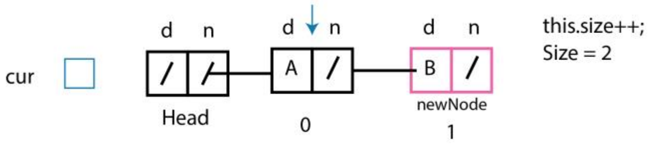
Size = 0

public boolean add(Object o){}

Edge Case: if head's next value is null then
add a new ListNode



Set a ListNode named cur to the head's next value while cur next is not null then set cur to it's own next value. We say cur is not null because that's how we indicate the end of the list.

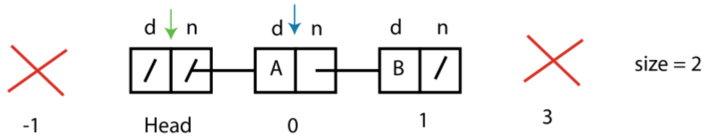


If at one point cur's own next value is null then set cur's next value to a new ListNode creation and finally increase the size of the LinkedList

public void addIndex(int index, Object o)

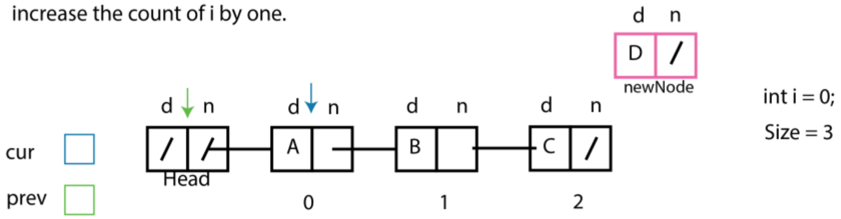
Edge Case: if the index is less than zero
or if the index is greater than size of the linked list
throw an out of bounds error.

Index < zero or index > size



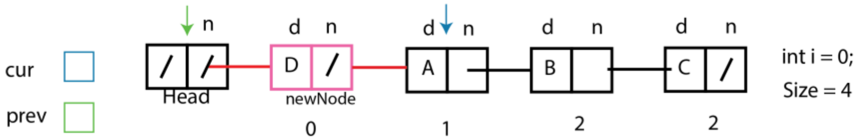
Otherwise, set a variable named previous to the head
and one called cur to head's next value. Initalize
a variable named i to zero. Also, create a variable that will hold a new list node.

while the variable i is not the given index, then prev gets cur, cur gets cur's next value, and
increase the count of i by one.



Note: while variable index is not the index
to iterate inside the loop prev will get cur, and
cur gets cur's next value. Keep incrementing i by one.
This example fails the while loop in the first iteration.

newNode(0, D)
while (0 !=0) false so
d



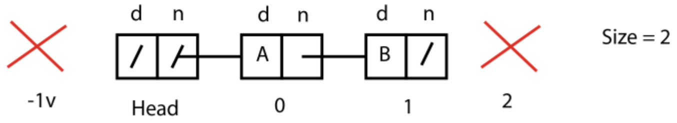
prev.next= newNode
newNode.next = cur
size increases by one

public Object get (int index) {}

Edge Case: if the index is less than zero
or if the index is greater than size of the linked list
throw an out of bounds error.

Index < zero or index >= size

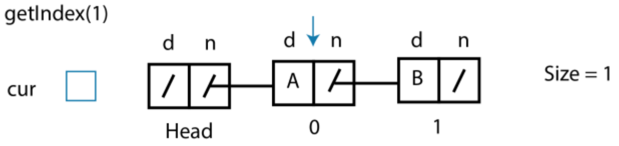
getIndex(2)



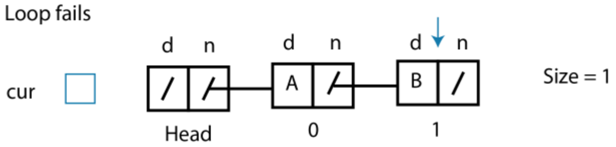
Otherwise, set a variable named cur to head's next value. Initalize a variable named i to zero.

while the variable i is not the given index, then cur gets cur's next value, and increase the count of i by one. If cur is null then get cur's data value.

getIndex(1)



Loop fails



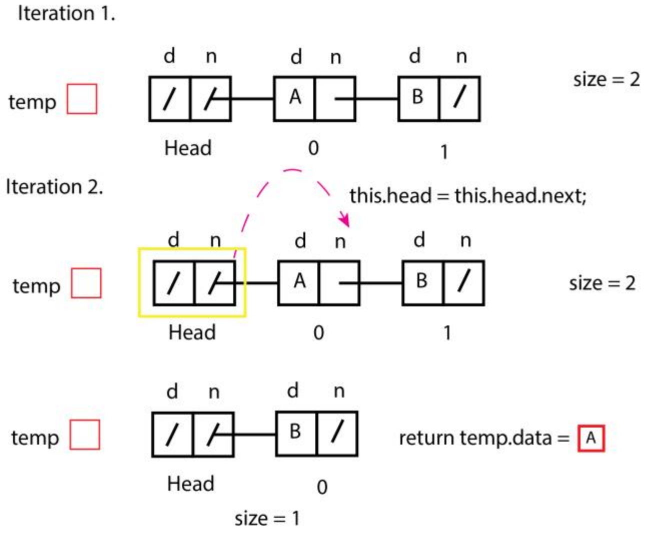
return cur.data = $\boxed{A}$

public Object Remove(int index){}

Edge Case: if the index is less than zero
or if the index is greater than size of the linked list
throw an out of bounds error.

Index < zero or index >= size

Edge Case: if the index is zero then create
a temp object variable named temp. Set that variable
to the head's next value. Grab the head and set it to head's next value.
Decrease the size by one and return the data.



this.head = this.head.next;

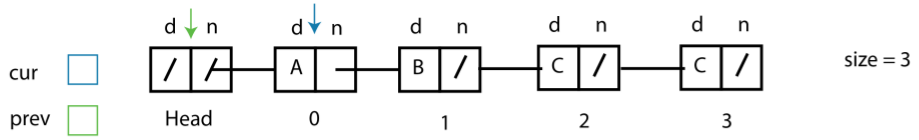
return temp.data = A

Continued: public Object Remove(int index){}

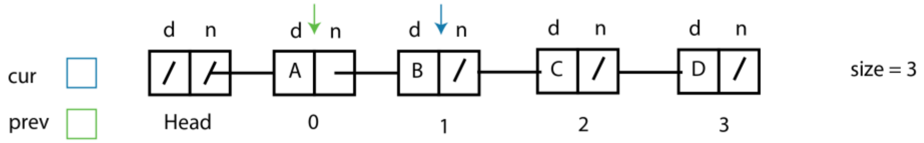
Normal case: set a list node variabled called prev
to head. Also, set a list node variable called cur to
to head’s next value.

Since we are removing it is important to notice that
two variables work together in order for a skip to happen.

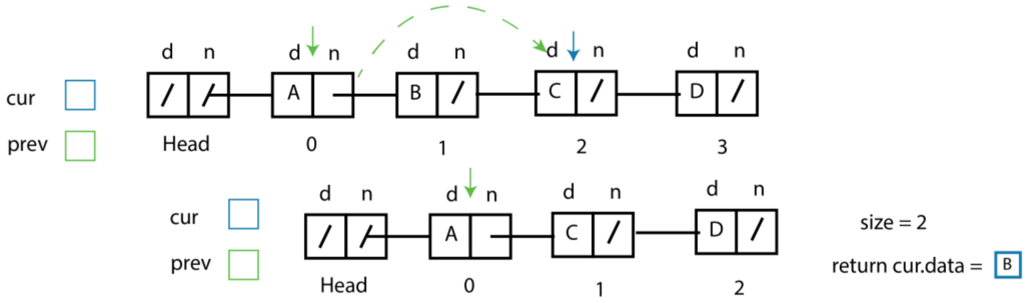
Iteration 1: remove(2)



A for loop that stops one step before the index will help us iterate
through a loop in this case. In the for loop prev will get cur
and cur will get cur next’s value. This is the process to step forward in the list.



This is where the loop stops since we have reached the index value minus
one. So prev’s next value gets cur’s next value. The list’s size decreases

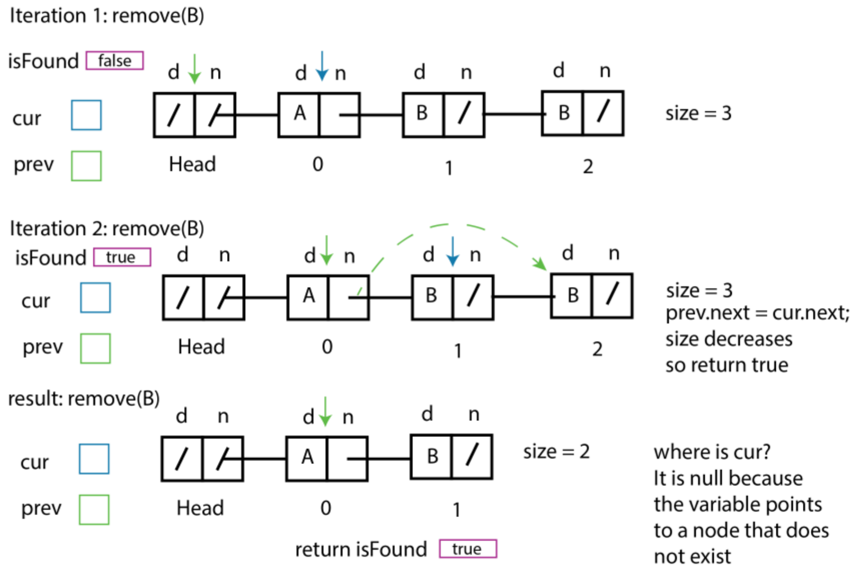


public boolean remove(Object o) {}

Normal Case: using a for loop we will iterate all of the list's size. Then check if cur's data or if the object equals cur's data. If that is not true then let prev get cur and cur get cur's next value to move foward in list.

If cur's data is not null or Object o equals cur's data then the value has been found.

Since it has been found, in it's own if statement. That's check if it true we can say that the variable prev's next equals cur's next to skip over the found variable. The size of list will decrease and we will return true;



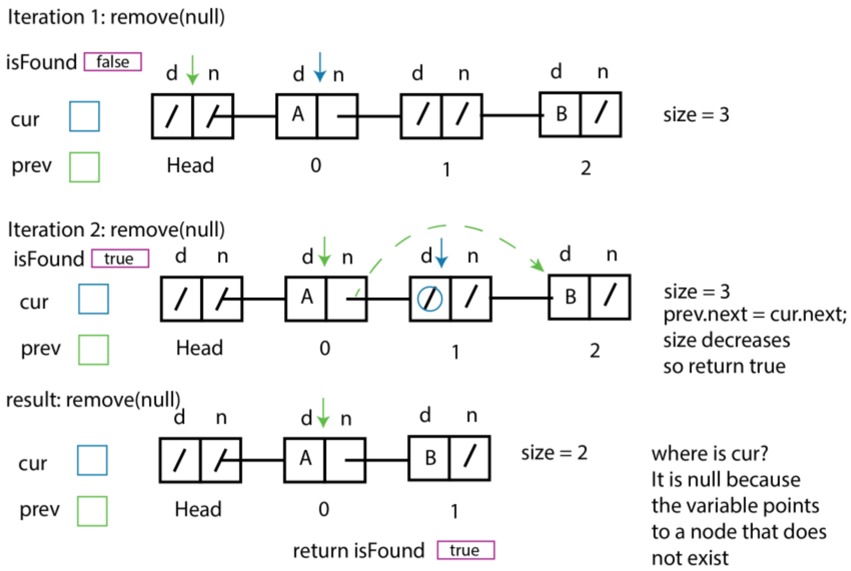
prev.next = cur.next;
size decreases
so return true

where is cur?
It is null because
the variable points
to a node that does
not exist

Continued: public boolean remove(Object o) {}

If the passed object is null then double check with another
if statement that cur's data is null. If that is true
we have considered the null parameter.

The boolean variable evaluates to true so we will
skip over to the next node. Prev next gets cur next
size decreases, and we return true because null has been found.



size = 3
prev.next = cur.next;
size decreases
so return true

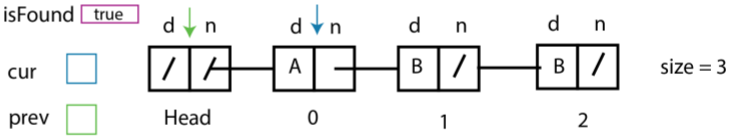
where is cur?
It is null because
the variable points
to a node that does
not exist

public boolean contains(Object o) {}

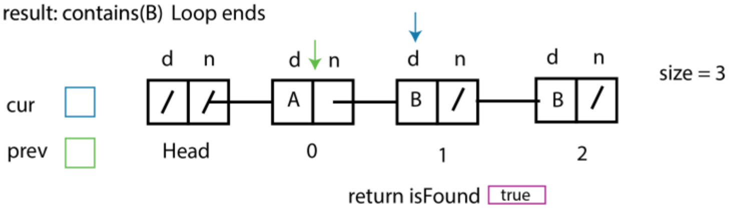
Normal Case: this time I decide to go with a while loop. while cur is not null then evaluate if cur's data is not null and Object o equals cur's data. If these two statements are true then return the boolean variable value of true.

Edge Case: while cur is not null then check if the passed object was null. At this point double check if cur's data is null. If it is true then return the boolean variable evaluated as true.

Iteration 1: contains(B)



result: contains(B) Loop ends



result: contains(null)

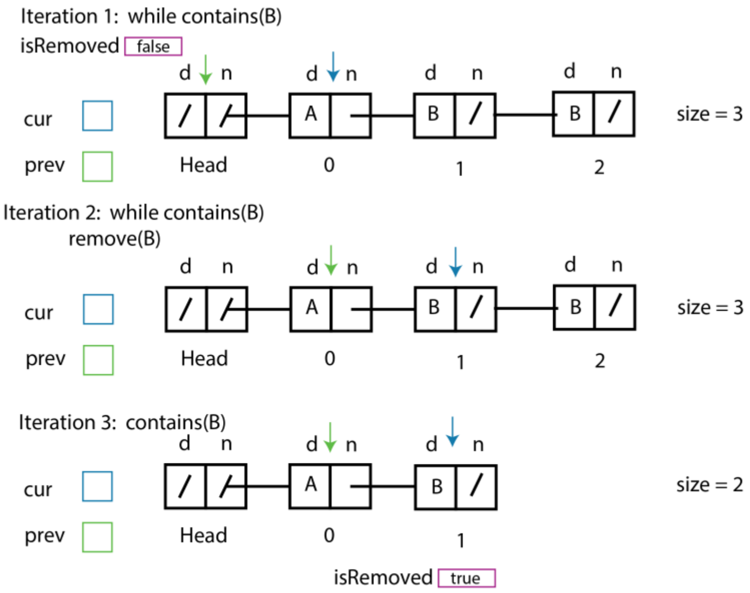
If none of these conditions evalaute to true
then the value is not in the list so therefore contains if false.

public boolean removeAllCopies(Object o) {}

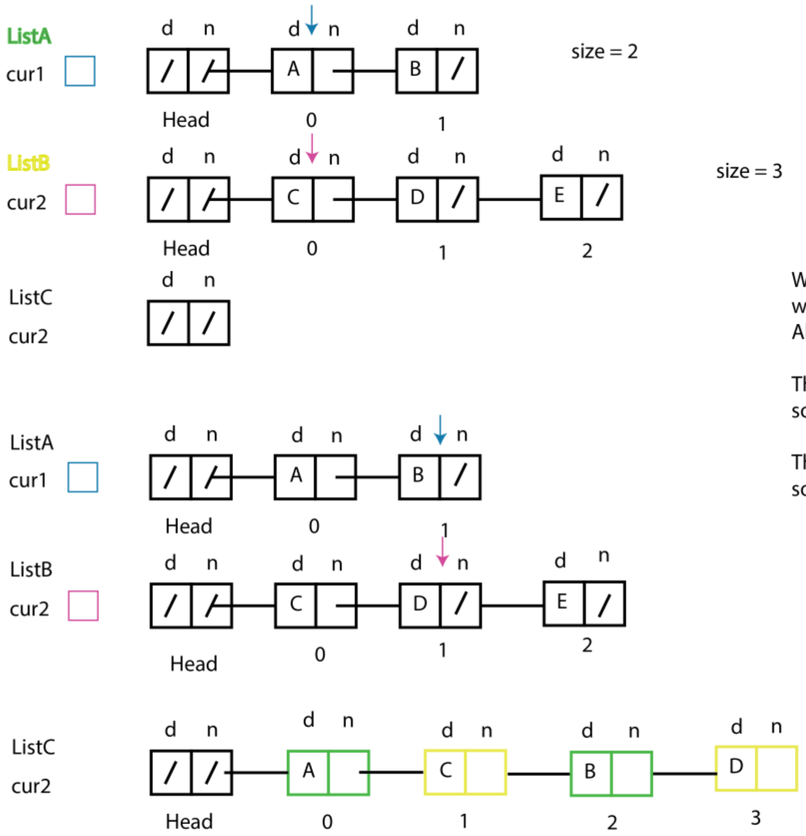
This method will use both the contains(o) and remove(o) method.
We are evaluating if there are copies so a while loop is used.
while the list contains a certain value.

Then set the boolean variable to the remove() method
At this point the boolean variable is either true or
false depending on the remove() method result.

Considering multiple consecutive copies that is how
the remove() variable takes a role. The remove()
method will remove the element in the lowest index even
though there are multiple ones. If consecutive it will remove
the lowest index element.



public static MyLinkedList interleave(ListA, ListB){}

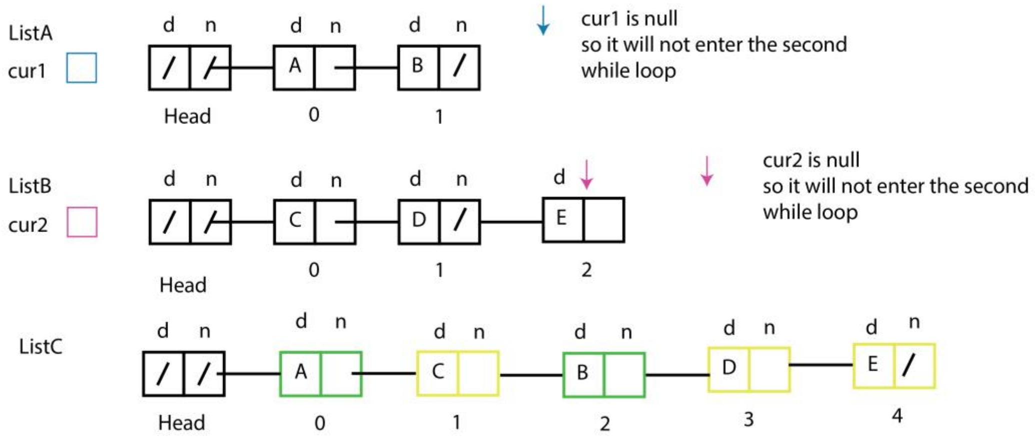


While cur1 and cur2 are not null then we will add cur1's data to list c. Also, we will add cur2's data to list c.

The walker cur1 must walk forward so cur1 gets cur1 next.

The walker cur2 must walk forward so cur2 get cur2 next.

Continued: public static MyLinkedList interleave(ListA, ListB){}



cur1 is null
so it will not enter the second
while loop

cur2 is null
so it will not enter the second
while loop

cur1 is now null so the intial while loop fails.

At this point we enter a new while loop and evaluate while cur2 is not null

add cur2's data into list c, and let cur2 get cur2's next value.

Moreover, we enter another while loop and evaulate while cur1 is not null. Add cur1's data into list c, and let cur2's data get cur1 next value.

These two additional while statements are used because either last can have an undetermined amount of remainders. So, we must check for the remainders so we can add them to list c.