

Terse Linked List


Think about the problem in a different way

Analysis

- ▶ The previous linked list insert code thinks in terms of nodes:
 - “Put this new node between these two nodes”
- ▶ That gives us a previous and a traverse pointer
- ▶ That gives us special cases
 - Empty list
 - Insert at the front of the list
 - Insert at the end of the list (maybe)
- ▶ The code is long but straightforward as long as you keep track of what case you are in

What if we think about it in a new way?

Let us think in terms of pointers to nodes instead of nodes

- ▶ The head pointer is no longer a special case, it has the same type as a next pointer
 - ▶ To do the insert, exactly 1 existing pointer in the list needs to change because something needs to point to the new node
 - ▶ That 1 existing pointer holds the value that the new node's next pointer needs
- 

Results

- ▶ Insert boils down to:
 - *Find the pointer that needs to change*
 - Give that value to the new node's next pointer
 - Set the pointer that needs to change to point to the new node
- ▶ There are *no* special cases at all
- ▶ The code is short (4 lines)
- ▶ The code is dense
- ▶ The code is harder to understand, write, debug, and maintain
- ▶ The code optimizes very well
- ▶ The code is “elegant”

How do we make this work?

- ▶ We exploit the fact that we can take the address of a pointer
 - This gives us a pointer to a pointer
 - Doing so demands careful attention to type:
 - Node
 - Pointer to node
 - Pointer to pointer to node
 - Doing so demands careful dereferencing (see above)
- ▶ We exploit pass by value to give us a perfectly initialized local variable
 - A pointer that points to the head pointer is no different than a pointer that points to a next pointer
- ▶ We exploit short-circuit evaluation to protect dereferencing

Calling insert

```
struct Thing { int dval; struct Thing *next;} ;  
struct Thing *head = NULL, *newnode = NULL;
```

```
/* assume we allocate the new node and populate the  
** data value in that node. Now we insert it:  
*/
```

```
insert( newnode, &head);
```

```
/* C is pass by value so we have to pass the address  
** of the head pointer since we intend to change it */
```

```
struct Thing { int dval; struct Thing *next;} ;
```

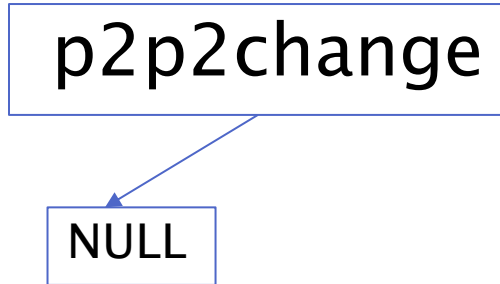
```
insert( struct Thing *newnode, struct Thing **p2p2change)
{
    printf("inserting %d (lives at %p)\n", newnode->dval, newnode );
    /* skip nodes with bigger data than data to insert */
    while( *p2p2change != NULL && (**p2p2change).dval > newnode->dval)
    {
        printf("Going past %d\n", (**p2p2change).dval);
        p2p2change = &(**p2p2change).next;
    }
    /* do the insert: */
    /* make new node next point to what the changing pointer currently points to */
    newnode->next = *p2p2change;
    /* make the pointer to change point to the new node */
    *p2p2change = newnode;
    printf("Setting changed to %p, next to %p\n", *p2p2change, newnode->next);
}
```

Let us deconstruct that code...

while(*p2p2change != NULL

- ▶ We have a pointer to a pointer, so we dereference 1 time to get a pointer to a node
- ▶ If that pointer we are pointing at is NULL, there is either nothing in the list (head is NULL) or we hit the end of the list (some next pointer is NULL)
- ▶ In either case we stop here and this pointer will point to the new node and the new node next will get set to NULL and our new node is at the end of the list.

Insert – NULL pointer



- If the pointer we are looking at is NULL, we have found the right spot for insert
- This could be the end of a list with many nodes
- The list could be empty

Let us deconstruct that code...

&& (p2p2change).dval > newnode->dval**

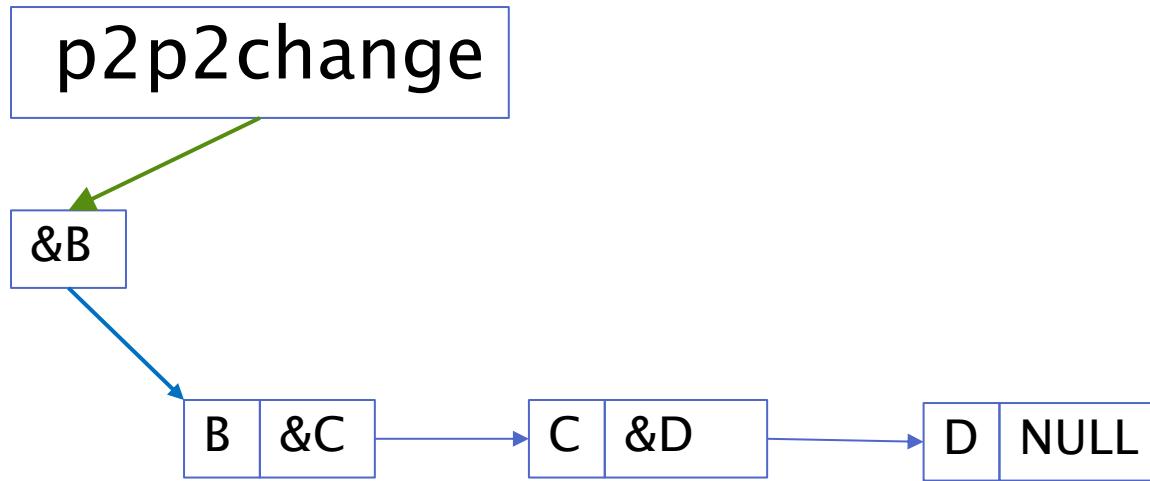
- ▶ Short circuit evaluation prevents this from being evaluated if the pointer our pointer points at is NULL (see previous slides)
- ▶ Since it is not null, we can dereference the pointer to a pointer 2 times to get to a node.
- ▶ We can thus compare the data value of the node pointed at by the pointer p2p2change points at to the data value in the new node.
- ▶ If the new data value is larger, the condition fails, stopping us at the right spot in the list (list is ordered biggest first)

Let us deconstruct that code...

p2p2change = &(p2p2change).next);**

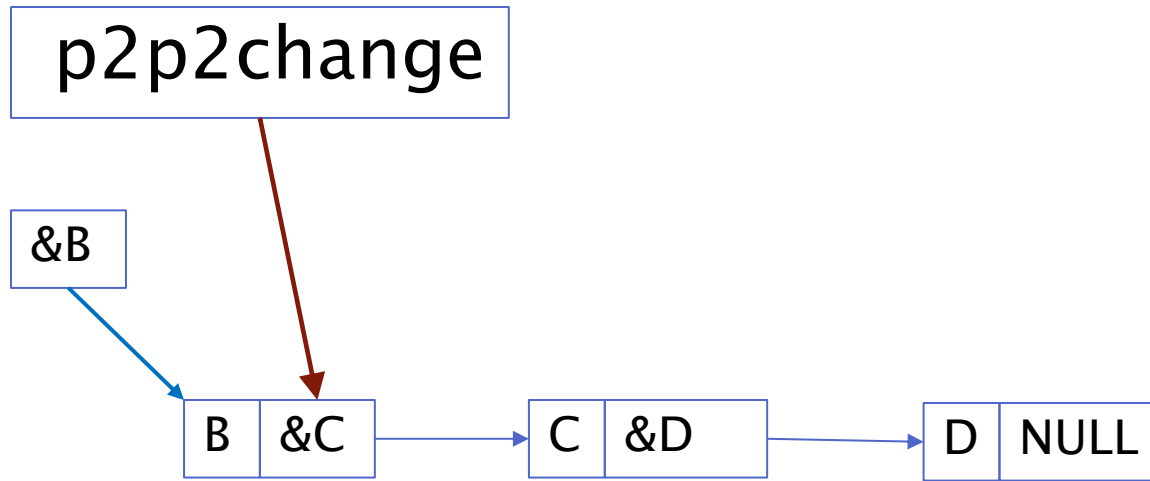
- ▶ We know that we can double dereference p2p2change (see previous slide). That takes us to a node (innermost parentheses)
- ▶ We can get to the next pointer of that node with **.next**
 - There are alternative ways to code this
 - The syntax here emphasizes type
 - There are *many* ways to code this that look right, compile, and *fail*
- ▶ We don't want the *value* of the next pointer since we aren't dealing in nodes. We want the *address of* the next pointer which we get with the outer &

Insert – skipping over a node



- `p2p2change = &((**p2p2change).next);`
- Move `p2p2change` to point to the next pointer of the node pointed to by the pointer we are looking at
- *Note: `p2p2change` does not point at the node, it points at the next pointer in the node*

Insert – skipping over a node



- `p2p2change = &((**p2p2change).next);`
- Move `p2p2change` to point to the next pointer of the node pointed to by the pointer we are looking at
- *Note: `p2p2change` does not point at the node, it points at the next pointer in the node*

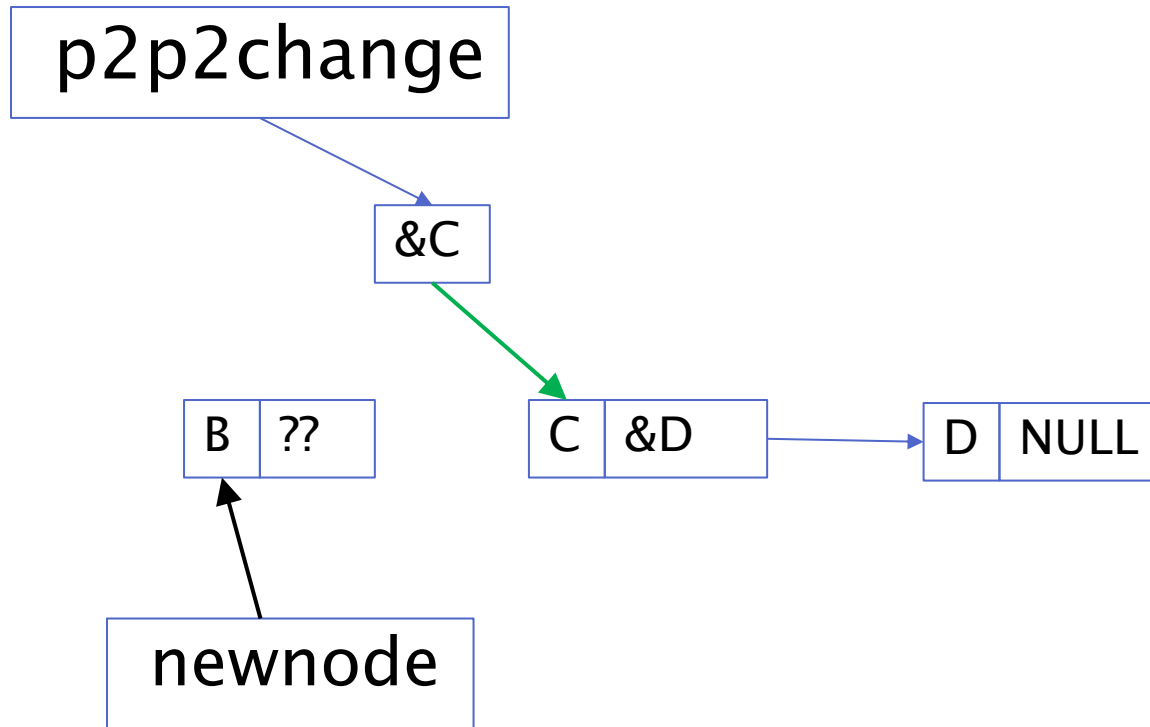
Let us deconstruct that code...

newnode->next = *p2p2change;

***p2p2change = newnode;**

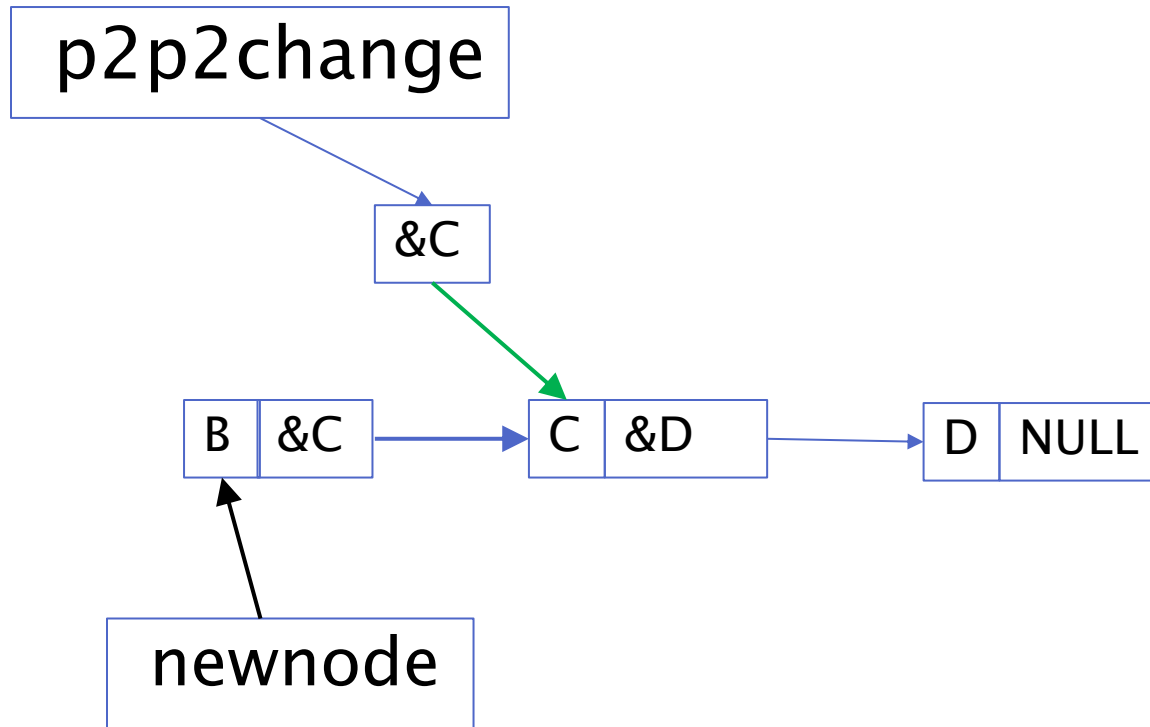
- ▶ Whatever value the pointer we have a pointer to has, we want the new node's next pointer to also point there.
 - It could be NULL
 - It could point to a node with a smaller data value
- ▶ We need to do the actual insert by changing the pointer that needs to change
 - We dereference once to get to that pointer
 - We set it to point to the new node

Insert – insert node B



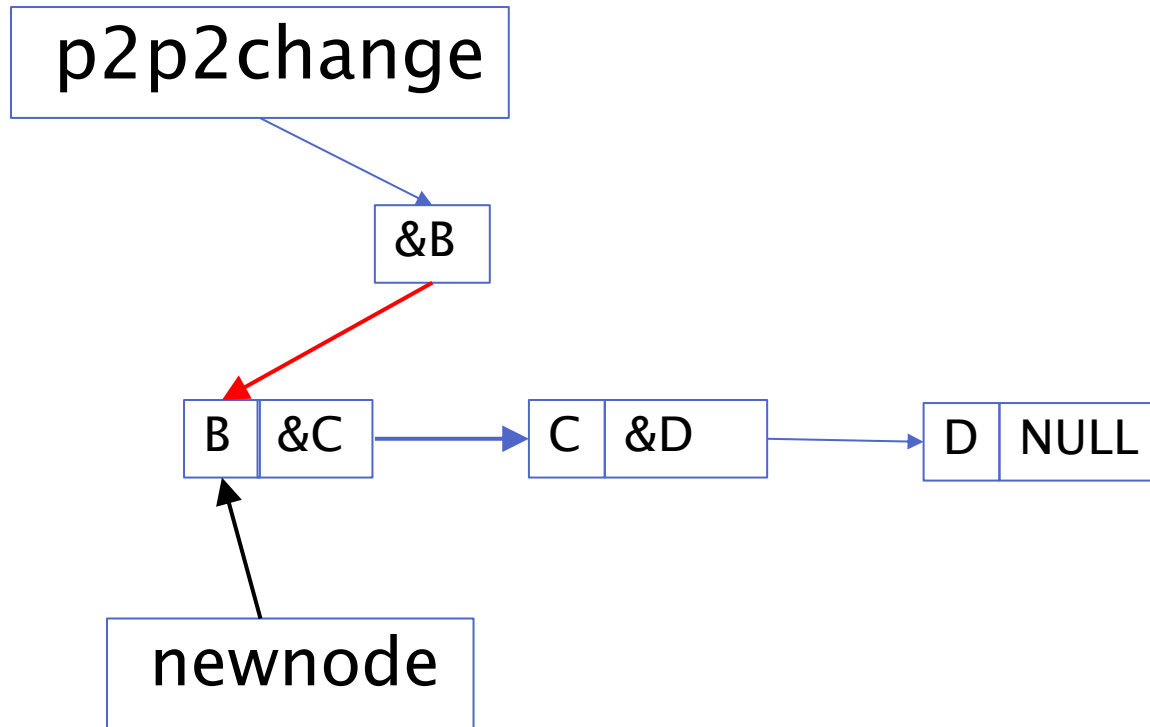
- To do: First set newnode next to point where the pointer we are looking at points

Insert – insert node B



- Shown here: First set newnode next to point where the pointer we are looking at points
- To do: Then change the pointer we are looking at so that it points to the new node

Insert – insert node B




- Already done: First set newnode next to point where the pointer we are looking at points
- Shown here: Then change the pointer we are looking at so that it points to the new node

Is it worth it?


- ▶ If you understand pointers to pointers and all of the other C language details, it's hard to beat a 4 line linked list insert routine
- ▶ If you don't understand any single bit of the code it becomes incomprehensible and thus worthless
- ▶ Industry code out there comes in 3 flavors
 - Too simple and thus long and tedious
 - Just right
 - Too clever and impossible to touch without wholesale replacement

What about delete?

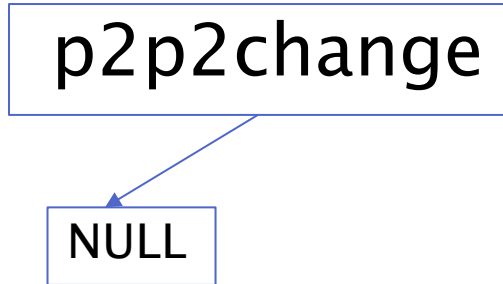
The same analysis works for delete:

- ▶ Somewhere in the list we might have pointers that will need to change because they point to nodes that need to be deleted
 - ▶ Those pointers need to be set to point wherever the node being deleted pointed to
 - ▶ We probably need to deal with the node itself before we burn all of the pointers we have that point to it
- 

Delete

- ▶ The code is 1 while loop – stop on NULL pointer
 - ▶ There is 1 if statement inside the while loop – do we delete this node?
 - ▶ There are no special cases
 - ▶ A single local variable is helpful so that we still have a pointer to any node we want to delete after we remove that node from the list. This way we can properly dispose of the node.
 - ▶ See diagrams to follow
- 

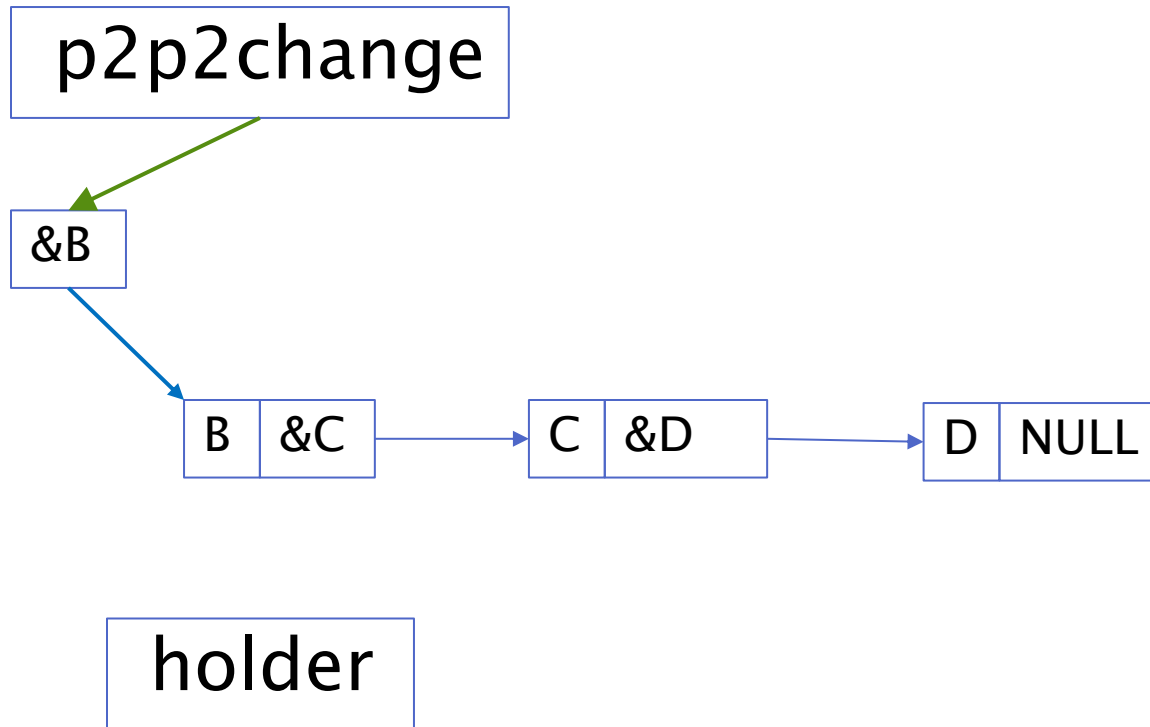
Delete – end case



- If the pointer we are looking at is NULL, we are done

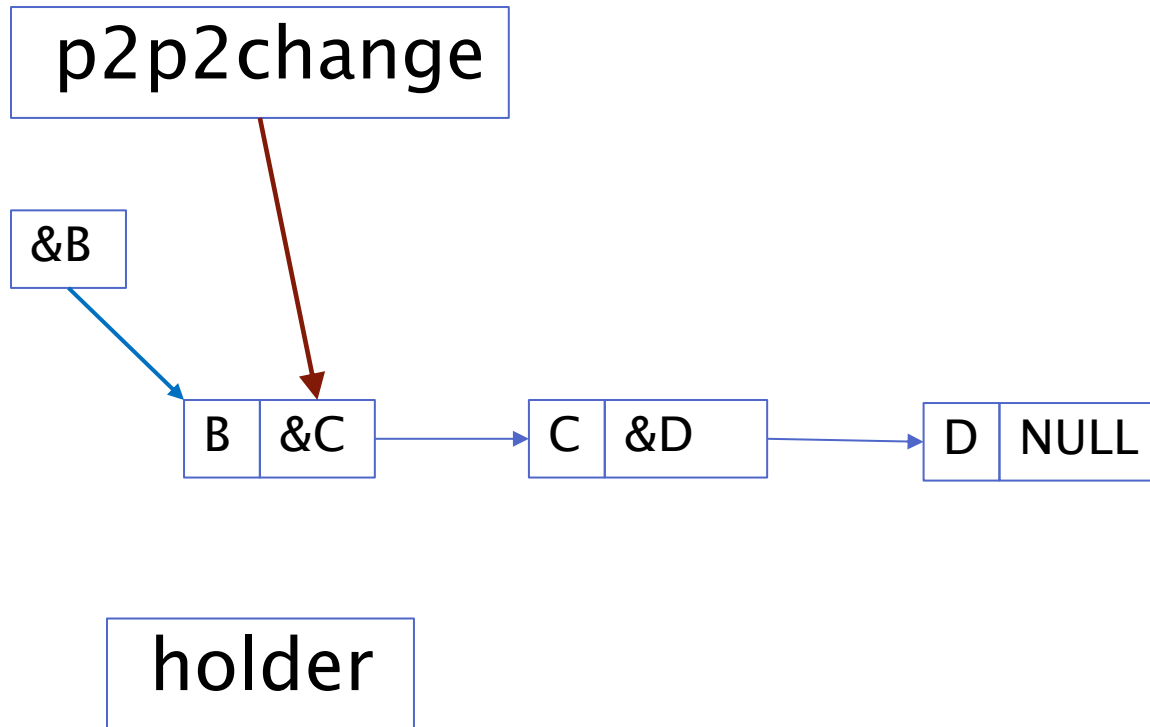
holder

Delete – skip over node B



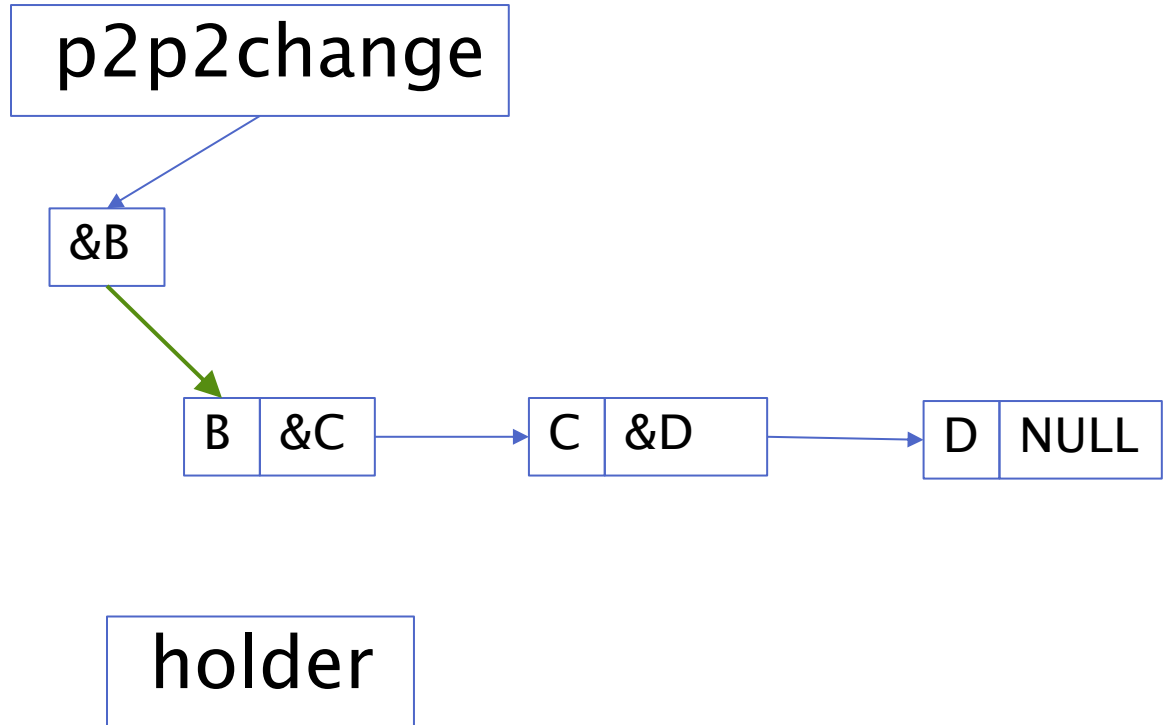
- The pointer we are looking at holds the address of B
- It could be the head pointer or it could be the next pointer of a previous node
- Since node B stays, the pointer we are looking at won't change
- To do: Move p2p2change to point to the next pointer of the node pointed to by the pointer we are looking at

Delete – skip over node B



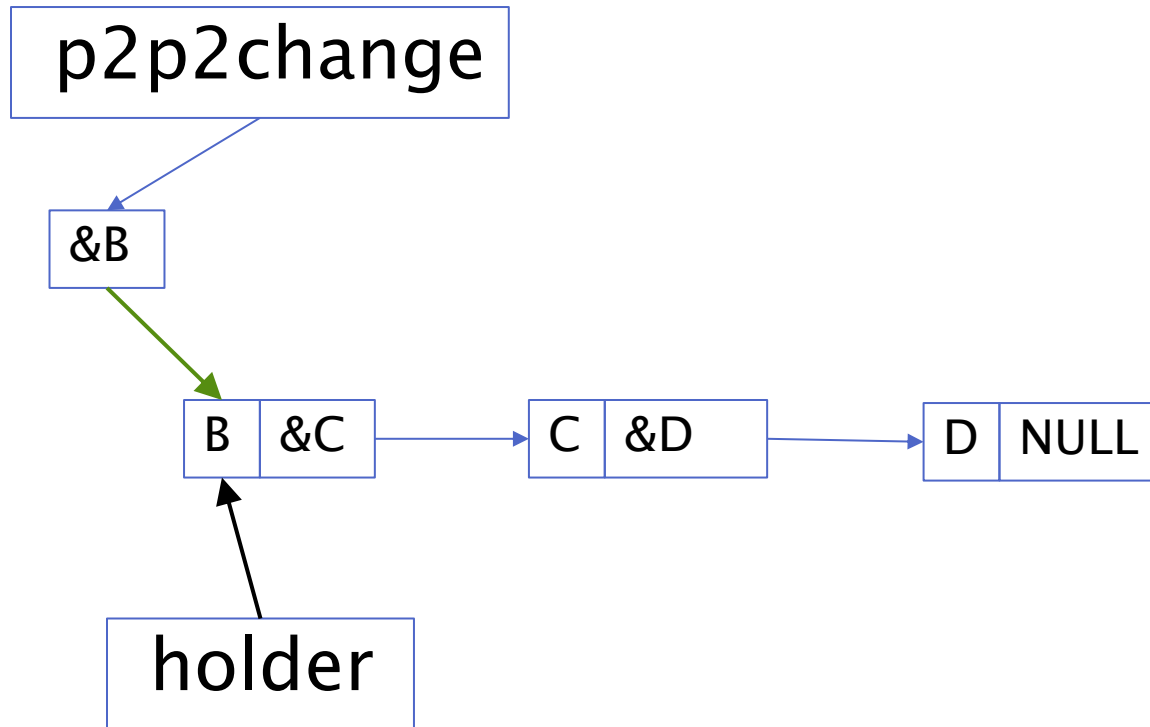
- The pointer we are looking at holds the address of B
- It could be the head pointer or it could be the next pointer of a previous node
- Since node B stays, the pointer we are looking at won't change
- Shown here: Moved p2p2change to point to the next pointer of the node pointed to by the pointer we are looking at

Delete – delete node B



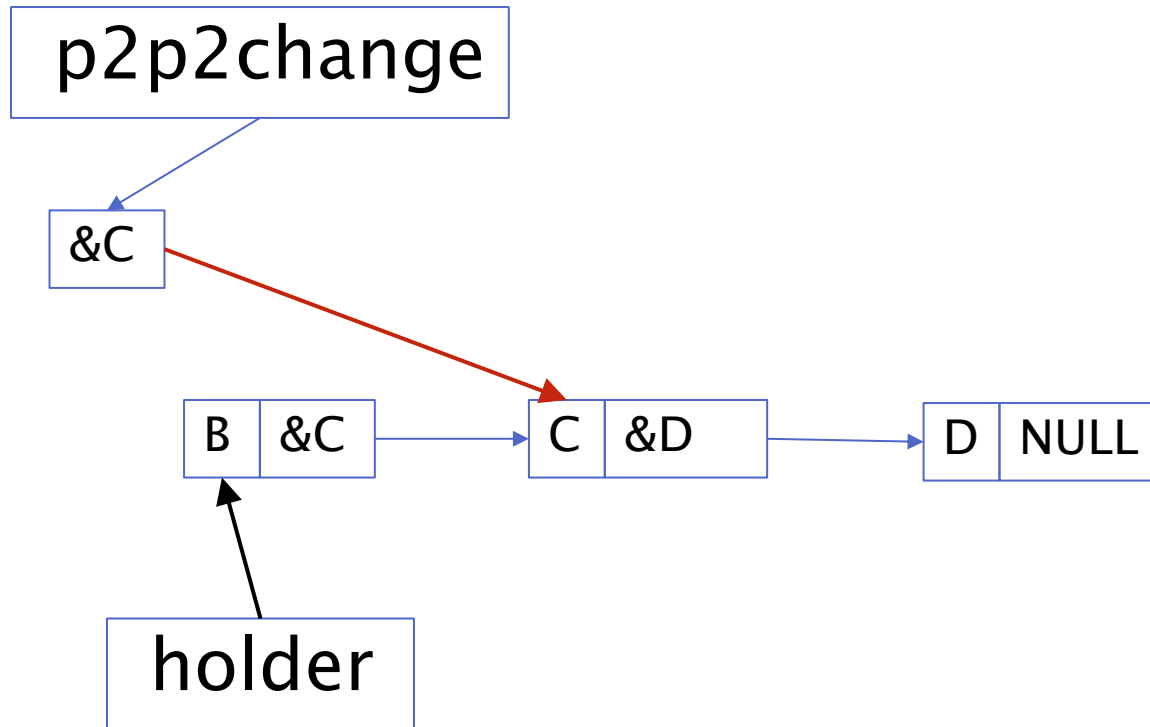
- To do: First set the holder to point to the node so we can properly dispose of it

Delete – delete node B



- Shown here: Set the holder to point to the node so we can properly dispose of it
- To do: Change the pointer we are looking at so that it points to the next node
- Do not change p2p2chage – it is possible that node C will be deleted on the next iteration of the loop. If that is the case we are right now looking at a pointer that will need to change again.

Delete – delete node B



- Already done: Set the holder to point to the node so we can properly dispose of it
- Shown here: Change the pointer we are looking at so that it points to the next node
- Do not change p2p2chage – it is possible that node C will be deleted on the next iteration of the loop. If that is the case we are right now looking at a pointer that will need to change again.
- Now we can dispose of the node that holder points to

Terse Delete – code

- ▶ The code for this delete is left for the student

One Purpose Statement

“Somewhere in this list, a pointer needs to be changed.”

The difference between finding the right node and finding the right pointer is key.

The analysis is more important than the code – the analysis *gives* us the code.

This code and the analysis behind it is an open invitation to raise your game.

