Linked Lists

CSE 2421

Recommended Reading: *Pointers On C*, Chapter 12, Sections 12.1 through 12.2.2

Linked List Nodes

- Previously, we saw how to declare a struct with a pointer to a struct of the same type. This kind of struct variable can be used as ***a node*** in a linked list. The node typically has two members:

  - One member, called *data*, is a C struct which is used to store data of interest.

  - The other member, called *next*, is a pointer to a struct Node. For example:

```
struct Data {
  char first_name[15];
  char last_name[15];
  int idNumber;
};
typedef struct Node{
  struct Data student;
  struct Node *next;
}Node;
```

- The linked list will be constructed by creating a number of these nodes ***dynamically***, and linking them to each other with the *next* pointer.

Why Use Linked Lists?

- The structure of a linked list is more complicated than an array, and more work is required to construct and maintain a linked list.
- Nonetheless, depending on the application, linked structures often have advantages.
- Some say Linear linked structures (linked lists) are not so commonly used in real applications (I disagree), but they are simpler to construct and manipulate than binary (search) trees, so we will only work with a linear, singly-linked list here.
- Binary search trees have significant advantages compared with (sorted) arrays in many cases, such as faster insertion and deletion times, but comparable search performance.

How do we construct a linked list?

- Using the nodes that we have introduced before, what else do we need to construct a linked list?
- Only one thing: an additional pointer, *list_head*, to point to the first node in the list:

```
/*Set list head pointer to NULL initially to show list is empty */
Node *list_head = NULL;
```

- To construct the list, we will make the *list_head* pointer point to the first node, the *next* member of the first node point to the second node, ... the *next* member of the n – 1st node point to the nth (last) node, and the *next* member of the nth (last) node point to *no node* (make it a NULL pointer).
- Making the next member of the last node point to NULL will allow us to detect where the end of the list is (and avoid segmentation faults!). This is analogous to ending a character string with a NULL character.

Constructing the Linked List – Creating a Node

/* Here are the necessary declarations for a *Node* of the type declared above, and for a *head* pointer */

```
      struct Data {
             char first_name[15];
             char last_name[15];
             int id_number;
      };
      typedef struct Node {
                    struct Data student;
                    struct Node *next;
      }Node;
      /*Initialize list_head to NULL since list is empty at first. */
      Node *list_head = NULL;
```

Note: no structures of type Node have yet been declared or allocated, only a pointer to the head.

Constructing the Linked List – Creating a Node cont.

```
/*With the declarations above, to create a new Node, we can use the following */

Node *newNodePtr;
newNodePtr = malloc (sizeof (Node));
if (newNodePtr == NULL) {
      /*The storage was NOT allocated, so print an error message and exit */
}
else {
      /* The storage was allocated, and we can use the Node */
      /* Call a function to get the data for the node and store it in the node */
}
```

Note: When we read the data from input which is to be stored in the node, we can read it ***directly*** into the node (how?), rather than storing it in another variable, and then assigning that variable to the node.

Constructing the Linked List – Creating a Node (cont.)

- White space can be an issue, since scanf can read both numeric data (ints and floats) and non-numeric data (strings).
- Keep in mind that scanf ignores leading white space characters (space, tab, etc.) for *numeric* data, but **not** for non-numeric data in the way we will read it (we will be reading strings which have white space characters).
- Therefore, if you are reading numeric data, you do not have to be concerned about consuming preceding newlines, because scanf will take care of it for you (it "consumes" any whitespace characters before the 1st digit).
- If you are reading non-numeric data, however, you ***will*** have to consume any preceding newlines (use getchar()), because scanf will not consume them.

How to Insert nodes into a List

- Typically, we want a list in sorted order so you must identify what value your linked list node contains that you want to sort on

Notes on list_head pointer

- Before we look at the details of insertion into the list, we need to consider the list_head pointer more carefully.
- Various functions will have to know the address in the list_head pointer in order to do things with the list (insert nodes, delete nodes, process data in the nodes, print data in the nodes, etc.).
- If we make the list_head pointer a file scope ("global") variable, we do not have to pass it as a parameter to these functions.
- This, however, is currently an unacceptable way to write software and should only be done in very limited cases (this is not one of them)! Debugging and maintenance are MUCH more difficult with global variables. (Oh, don't I know it!!)
- You are not allowed to use file scope variables in software you write for this course.
- Therefore, in general, the functions in your linked list program will have to be passed the list_head pointer, which is declared in main, as a parameter, if they need to access data in the list or modify the list in any way.
- The challenge for each of you is to determine whether the list_head pointer should be passed by value or by reference to each function that requires it. ☺

Notes on list_head pointer (cont.)

- We need to think about whether the function may need to modify (change) the list_head pointer or if it only needs to read it.
- If it needs to potentially modify it, we cannot pass the head pointer by value. Why?
- Which functions might need to modify the head pointer? Keep this question in mind as we look at the following slides.

How to Insert Nodes into a Sorted List

- If we want to insert a node into the list, what do we need to do?
- We will assume that the nodes in the list will be ordered in ***ascending order*** based on some member in the data member of the node. What member do we sort on for our lab 3?
- First, we have to create the new node (as shown above), and initialize the members of the data member, and the next member also, if you wish */
- Then, we have to call the insert function, and pass it the current list (how?), and the new node (how?).
- In the insert function, declare a pointer to traverse (go through) the list:

```
    Node *traversePtr;
```

- Set the traversePtr to point to the same address the list_head pointer points to, so that you start looking at the beginning of the list.

One-Purpose Statement for this Version of Insert

“What node do I insert the new node in front of?”

There are other ways of looking at the problem that will give us a very different implementation. We will cover that tomorrow.

How to Insert Nodes into a Sorted List

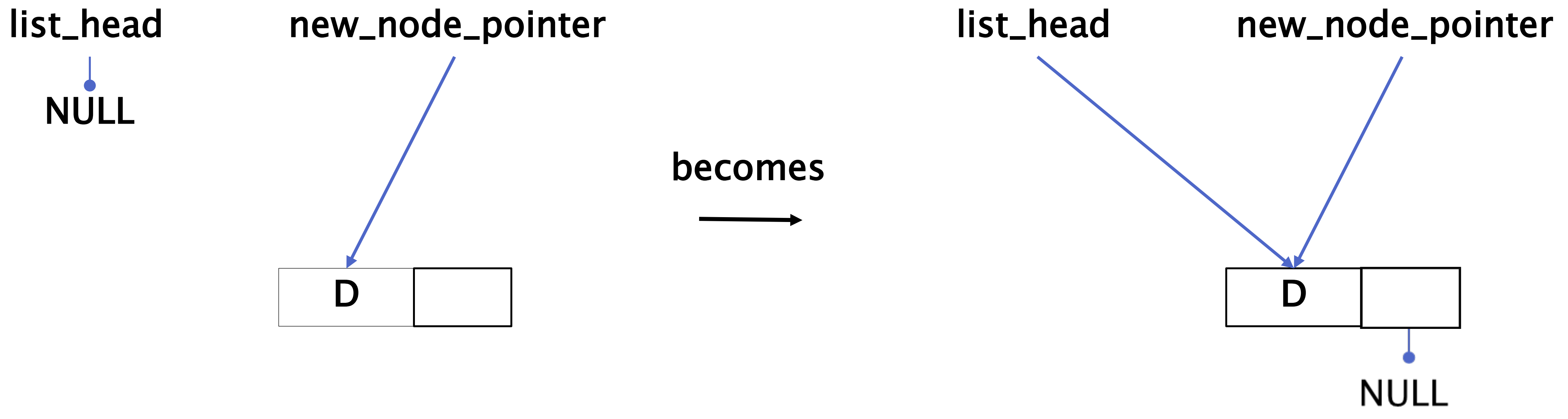
- When we traverse the list to determine where to insert, there are several possible cases or situations:

1. The list is empty (*list_head* is a NULL pointer).

   - To insert, just make *list_head* point to the inserted node. That is, list_head = address of node we wish to insert.

   - Make the *next* pointer of the inserted node point to NULL (it may have been initialized to NULL, but if not, this needs to be done here).

Insert to empty list



How to Insert Nodes into a Sorted List (cont.)

2. The list already has one or more nodes.

   – Check to see if the new node needs to be inserted as the first node in the list. How?

   - If so, insert it (It is not hard to figure out how to do this).

   - If not, as we traverse the list to search for where to insert the node, we have to maintain a pointer to the node *before* the node where we are (see below):

```
Node *priorNode;
```

   - So, traverse the list, node by node, to find out the correct place to insert the node; advance traversePtr and priorNode to go through the list.

   – When traversing, we need to check that the traverse pointer is not NULL before trying to access members of the node to which it points. If it is NULL, we have reached the end of the list, and should insert the new node there (as described below).

   - When the correct place is found, change these pointers (in this order!):

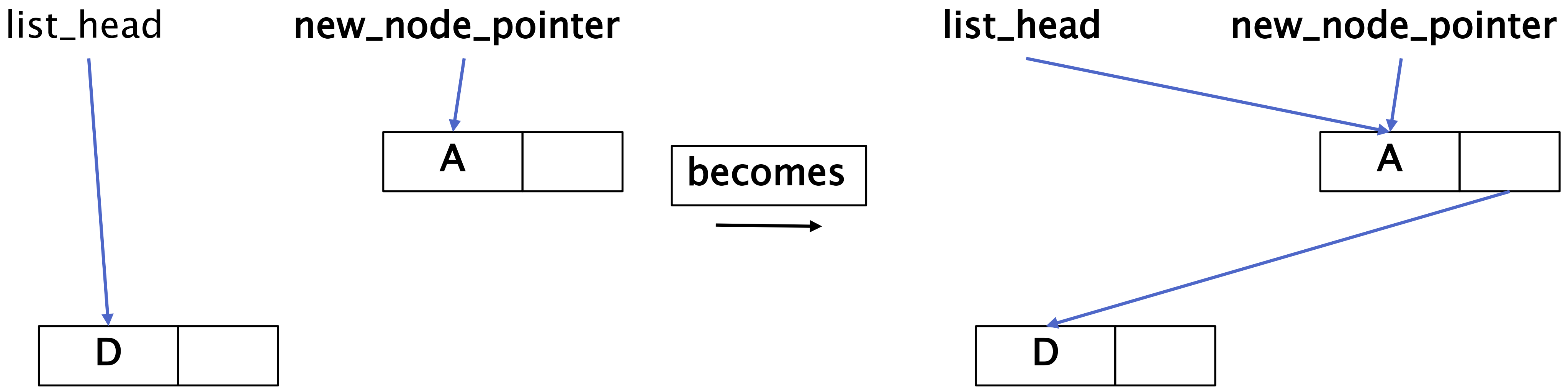
   a. The *next* pointer of the node to be inserted:

```
newNodePtr->next = traversePtr;
```

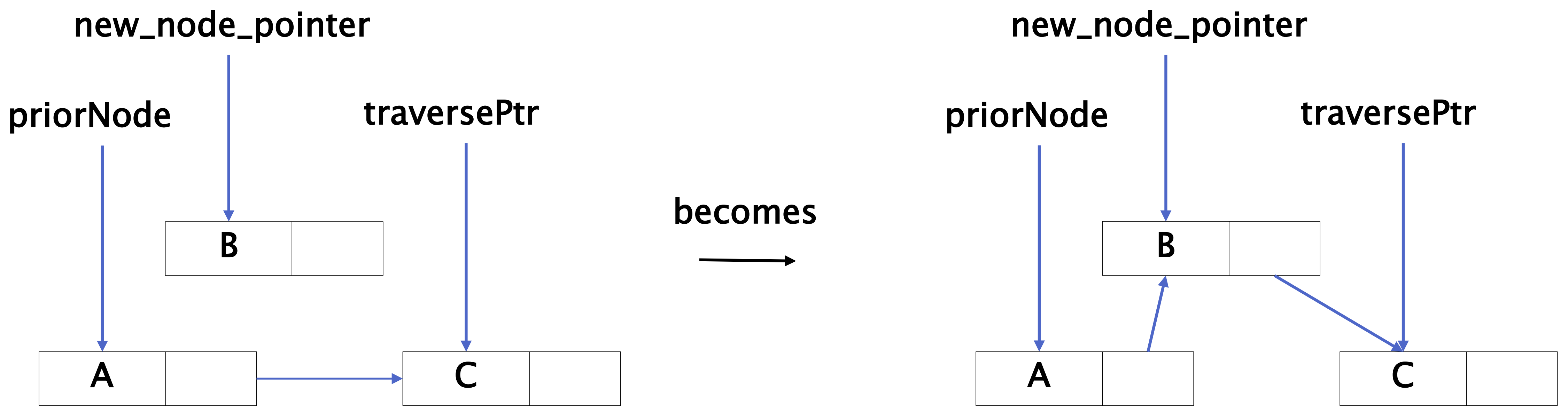
   b. The *next* pointer of the node after which the new node is being inserted (this is priorNode->next):

```
priorNode->next = newNodePtr;
```

Insert to front



General case insert



How to Delete Nodes from the List

- If we want to delete a node from the list, what do we need to do?
- Declare a pointer to traverse the list:

```
Node *traversePtr;
```

- Set the traversePtr to point to the same address the list_head pointer points to.

One Purpose Statement for this Version of Delete

“Do I need to delete this node?”

There are other ways of looking at the problem that yield a very different implementation.

How to Delete Nodes from the List cont.

- When we traverse the list to find the node to be deleted, there are several possible situations:

  1. The list is empty: ERROR - There are no nodes, so there is nothing to be deleted! Print an error message informing the user that they have tried to delete a non-existent node.

How to Delete Nodes from the List cont.

2. The list already has one or more nodes.

- Check to see if the node to be deleted is the first node. If so, in order to delete it, do these things, in this order (!!!):

  Make the list_head pointer point to the same thing as traversePtr->next

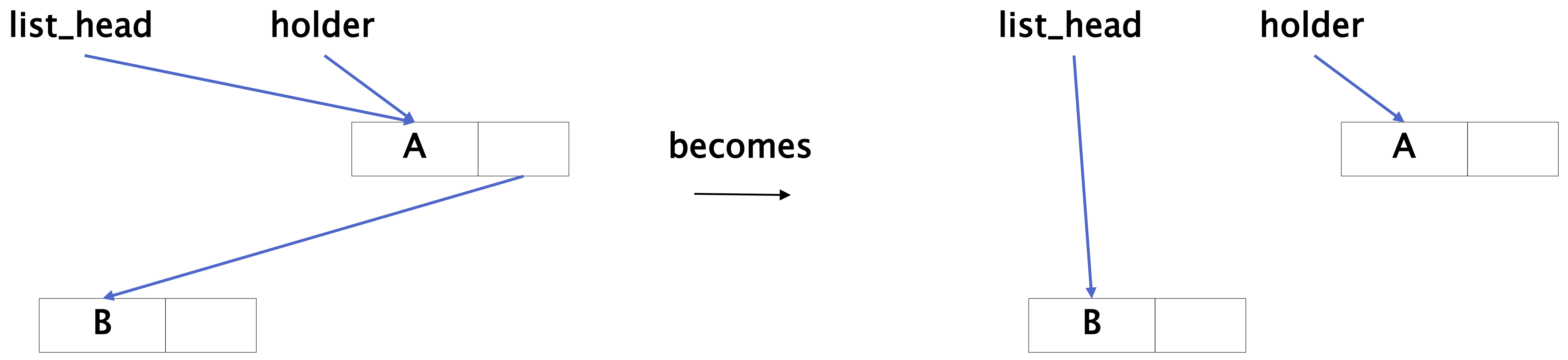
```
free(traversePtr);
```

- If not, as we traverse the list to search for the node to be deleted, we have to maintain a pointer to the node *before* the node where we are (see below):

```
struct Node *priorNode;
```

- So, traverse the list, node by node, to find the node which has the value to be deleted; advance traversePtr and priorNode to go through the list.

Delete from the front



How to Delete Nodes from the List cont.

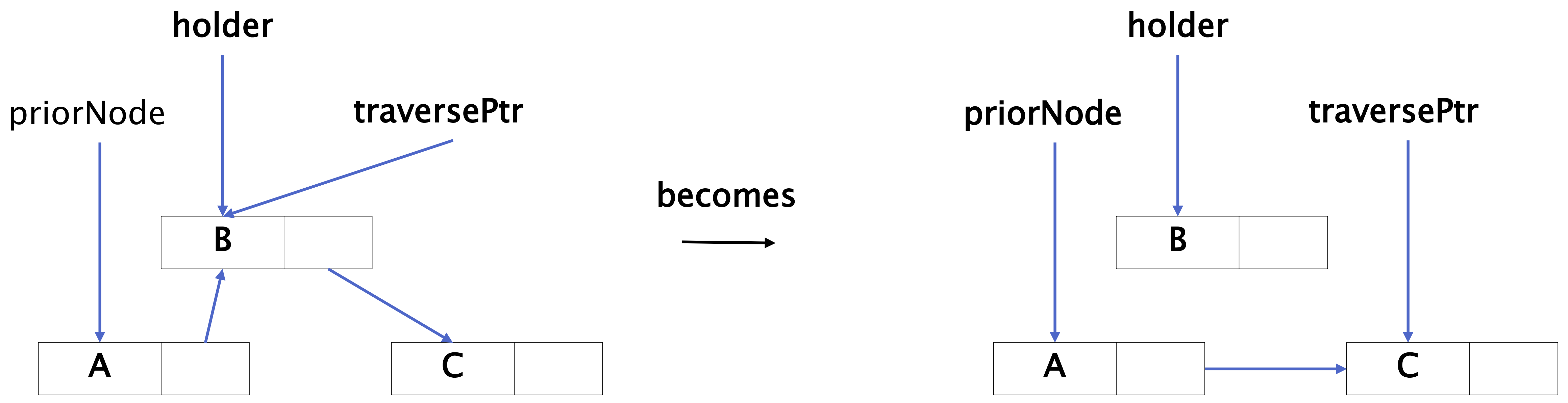
2. (continued)

  - When traversing, we need to check that the traverse pointer is not NULL before trying to access members of the node to which it points.
    - If it is NULL, the node is not in the list. Print an error message to the user.
  - If the node is found, do these actions (in this order!):
    a. Change the *next* pointer of prior node (the node before the node which is being deleted):

```
priorNode->next = traversePtr->next;
```

    b. Call free() with the pointer to the node being deleted (what pointer is this?).

General case delete



If deleting more than one node, you must also update the traverse pointer

Other Operations on Linked Lists

- Besides insertion and deletion, there are some other kinds of operations we can identify on linked lists:

  List traversal for some purpose:

  - Find an element in the list, and print out the related data.

  - Print all the items in the list.

  - Do computations on the data in the list, such as totaling all the items, counting the number of items in the list, computing averages, finding minimum and maximum values, etc.
- In general, traversing the list for these purposes is less complicated than it is for insertion and deletion, because we do not have to keep track of the prior node; we can declare a traversePtr and make it point to the beginning of the list and then go through the entire list.
- We do, however, still have to ensure that we do not try to access nodes beyond the end of the list (you should now be able to figure out how to do this)!

Freeing the Memory for the Nodes in a Linked List

- Before your program terminates, you should call free() to return the space for all the nodes, which was allocated as the program was running.

- Best practice is to m/calloc() a node just as you need the space and call free() for each node as soon as there is no longer a need for it. Think of pulling a paper cup from the stack, using it, then throwing it away when done.