COP 3223: C Programming
Spring 2009

Dynamic Storage Structures In C – Part 2

Instructor : Dr. Mark Llewellyn
markl@cs.ucf.edu
HEC 236, 407-823-2790
http://www.cs.ucf.edu/courses/cop3223/spr2009/section1

School of Electrical Engineering and Computer Science
University of Central Florida

Dynamically Allocated Arrays

- The last example in the previous set of notes created a dynamically allocated array using the `malloc` memory allocation function. We'll start off this set of notes using essentially the same example, but this time will use the `calloc` function to allocate the memory for the array. Other than this, the two programs are identical.
- Recall that `calloc` clears the memory locations by initializing every bit in the allocated block to 0.

As an aside on `calloc`, although it is most commonly used with arrays, it can be used to allocate memory for any object. By invoking `calloc` with the constant value 1 as its first argument, you can allocate space for a single data item (object/structure/type) of any type. The example below illustrates this technique:

```
struct twoDPoint {int x, y} *ptr;

ptr = calloc(1, sizeof(struct twoDPoint));
```


dynamic arrays using calloc.c

```c
//Dynamic Structures In C - Part 2 - dynamically allocated arrays using calloc
//April 21, 2009     Written by: Mark Llewellyn

#include <stdio.h>
#include <stdlib.h>

int main()
{
    int *ptr; //pointer to an integer - used to point to dynamic array
    int n; //number of integers in the array - user input from keyboard
    int i; //loop control variable

    printf("How many integer values do you want to store? \n");
    scanf("%d", &n); //read in n
    if ((ptr = calloc(n, sizeof(int)) ) == NULL ) {
       printf("Sorry, couldn't allocate that much space.\n");
    }//end if stmt
    else {
       for (i = 0; i < n; i++){
          printf("Enter number %d: ", i);
          scanf("%d", &ptr[i]); //notice pointer being used as a normal array name
       }//end for stmt
       printf("\n\nThe array contains:\n");
       for (i = 0; i < n; i++){
           printf("ptr[%d] = %d\n", i, ptr[i]);
       }//end for stmt

       printf("\n\n");
       system("PAUSE");
       return 0;
```

Notice the difference in the call to `calloc` compared to `malloc`.

Using The `realloc` Function

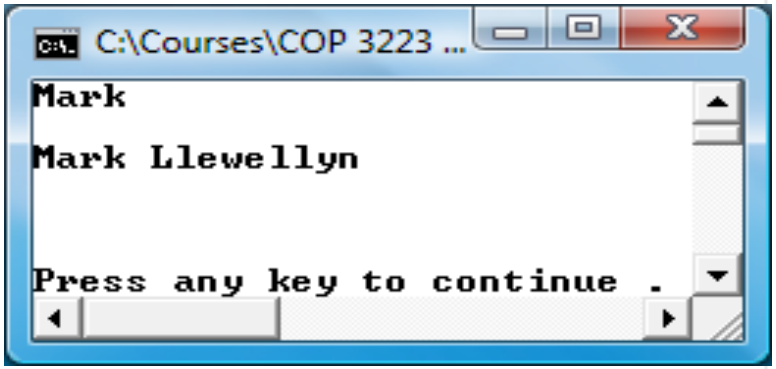
- The `realloc` function is used to resize a previously dynamically allocated block of memory. It can be either increased or decreased in size depending on the need.
- The `realloc` function must be invoked with a first parameter that is a pointer to a block that was previously returned by an invocation of either `malloc`, `calloc`, or `realloc`. If any other pointer is passed to `realloc`, the results will be unpredictable.
- The following example illustrates a common usage of the `realloc` function. In this example, an array of characters (a string) is allocated using `malloc` and then subsequently resized using `realloc` to allow an increase in the length of the string.

an example using realloc.c

```c
#include <stdio.h>
#include <stdlib.h>
#define N 5

int main()
{
    char *ptr1, *ptr2; //pointers to strings
    int i; //loop control variable

    if ((ptr1 = malloc(N)) == NULL) {
      printf("Sorry, couldn't allocated memory for ptr1\n");
    }//end if stmt
    else {
       strcpy(ptr1, "Mark");
       puts(ptr1);
       printf("\n");
       //Now I remembered that I wanted to add my last name to the string - duh!
       if ((ptr1 = realloc(ptr1, N+10)) == NULL) {
          printf("Sorry, couldn't allocated memory for ptr1\n");
       }//end if stmt
       else {
         strcat(ptr1, " Llewellyn");
         puts(ptr1);
         printf("\n");
       }//end else stmt

    printf("\n\n");
    system("PAUSE");
    return 0;
    }//end else stmt
```

Initial allocation only allocates 5 bytes 0 not enough room to hold both my first and last names.

Reallocation using `realloc` increases the allocation by 10 bytes, thus allowing enough room for my last name in the string.



Rules To Know When Using `realloc`

- The C standard spells out several different rules that dictate how the `realloc` function behaves:

1. When it expands a memory block, `realloc` does not initialize the bytes that are added to the block, even if the original block was allocated using `calloc`.
2. If `realloc` cannot expand a memory block as requested, it returns a null pointer. The data in the original memory block is unchanged.
3. If `realloc` is called with a null pointer as its first argument, it behaves exactly as `malloc`.
4. If `realloc` is called with 0 as its second argument, it frees the memory block.

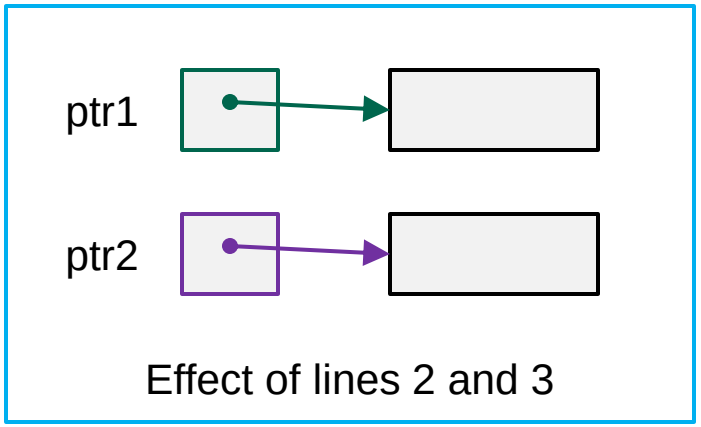
Freeing Dynamic Memory Allocations

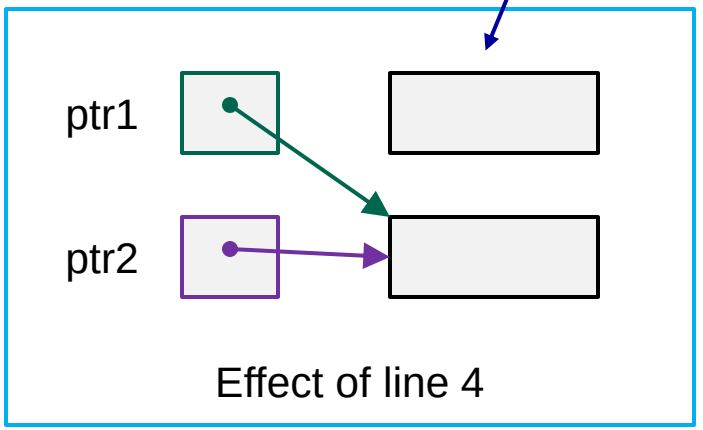
- When the `malloc`, `calloc`, and `realloc` functions obtain memory, it is allocated from the computer's heap memory.
- Calling these functions too often or requesting very large blocks of memory can exhaust the heap memory causing the functions to return null pointers.
- Programs can also loose track of allocated memory blocks through errors in programming logic. Such "lost memory" space is typically not recoverable until the program terminates, which means that such blocks are no longer able to be allocated to any executing program, even though they are not actually being used by any program.

Freeing Dynamic Memory Allocations

- To see how dynamically allocated memory can be "lost" through lack of attention by a programmer; consider the following scenario:

```
1. int *ptr1, *ptr2; //pointers to int
2. ptr1 = malloc(sizeof(int));
3. ptr2 = malloc(sizeof(int));
4. ptr1 = ptr2; //move ptr1
```





The `free` Function

- C provides the `free` function in `<stdlib.h>` to be used by the programmer to return memory no longer needed to the heap.
- The prototype for this function is: `void free (void *ptr);`
- Using the `free` function is quite easy, simply pass it a pointer to a memory block that is no longer needed and the system will return it to the heap memory.
- The proper way to have handled the previous example would be:

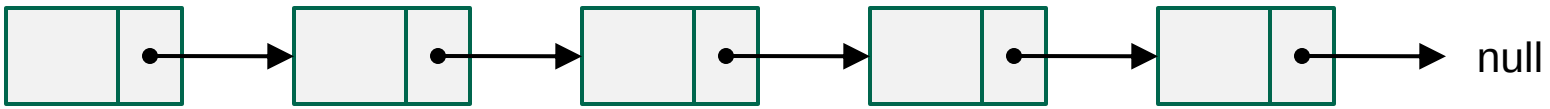
```
1. int *ptr1, *ptr2; //pointers to int
2. ptr1 = malloc(sizeof(int));
3. ptr2 = malloc(sizeof(int));
4. free(ptr1); //free int referenced by ptr1
5. ptr1 = ptr2; //reassign ptr1
```


Dynamically Allocated Self-Referential Structures

- Dynamic storage allocation is extremely useful for building many data structures common to an enormous range of applications.
- Some data structures that are typically constructed using dynamic memory are linked lists, stack, queues, trees, and graphs. If you go on in Computer Science you will become familiar with all of these data structures (and many more).
- We'll introduce you this concept using the linked list as an example, which can be applied to a very wide range of problems.
- Data structures of this type are said to be made up of nodes. A node is simply an object capable of holding some information. For our purposes, think of a node as a `struct`.

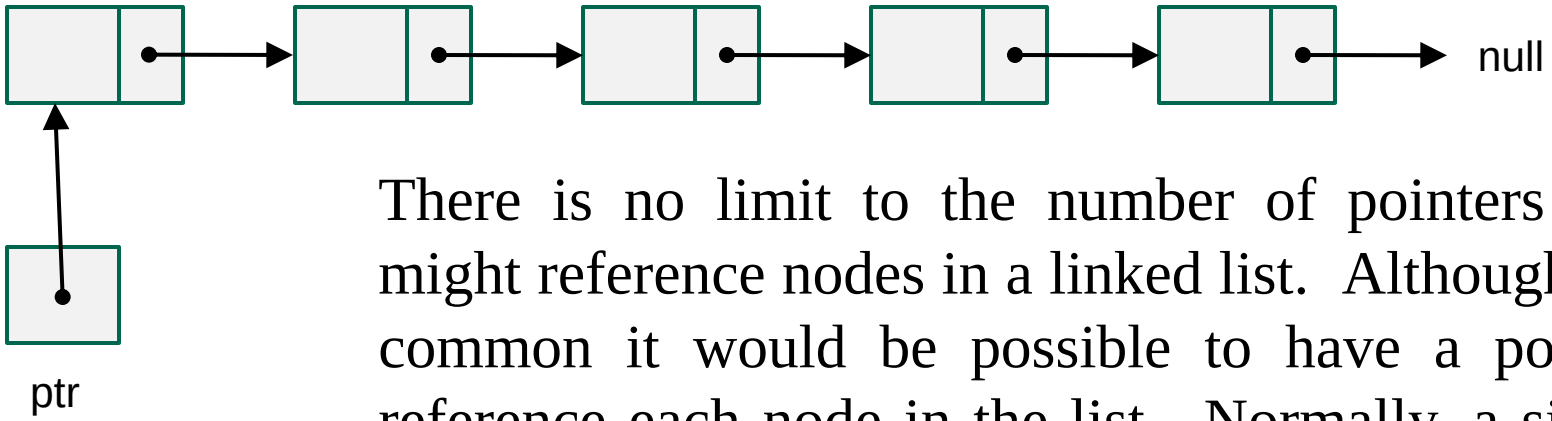
Dynamically Allocated Self-Referential Structures

- The nodes of data structures that are self-referential simply contain a member which is a pointer to another node of the same type.
- The pointer in one node is used to point to the next node in some logical ordering of the nodes.
- By chaining a number of nodes together using their pointer member, you create a linked list.
- Conceptually a linked list looks like this:



Dynamically Allocated Self-Referential Structures

- The one thing missing from the previous conceptual view of the linked list, is that we need some way to know where the logical beginning of the list is located. As you would expect, this is done with a pointer. So, the complete conceptual picture of a linked list actually looks like the one below:



There is no limit to the number of pointers that might reference nodes in a linked list. Although not common it would be possible to have a pointer reference each node in the list. Normally, a single node is used to traverse or "walk" a list using the self-referential links to the next logical node. The following example illustrated both extremes.

linked list example 1.c

```
1  //Dynamic Structures In C - Part 2 - a linked list example
2  //April 24, 2009      Written by: Mark Llewellyn
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #define MAX 10
7
8  struct aNode{
9     char letter;
10    struct aNode *next;
11 };
12 typedef struct aNode node;
13
14 void printList(node *ptr)
15 {
16       node *localPtr; //local pointer to move down list
17
18       localPtr = ptr; //assign pointer location
19       while (localPtr != NULL) {
20          printf("%c", localPtr->letter);
21          localPtr = localPtr->next;
22       }//end while stmt
23       printf("\n\n");
24       return;
25 }//end printList function
26
```

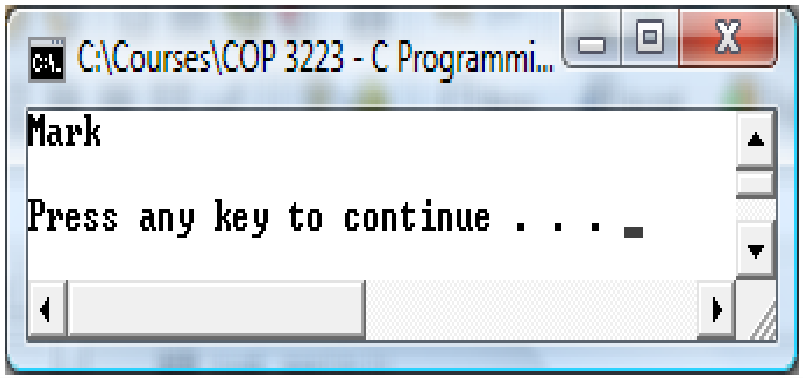
Notice the pointer used to move down the list is being reassigned to the value in the `next` field of the current node.

linked list example 1.c

```
27 int main()
28 {
29     node *ptr1, *ptr2, *ptr3, *ptr4;   //pointers to node objects
30
31     ptr1 = malloc(sizeof(node));
32     ptr2 = malloc(sizeof(node));
33     ptr3 = malloc(sizeof(node));
34     ptr4 = malloc(sizeof(node));
35
36     ptr1->letter = 'M';
37     ptr2->letter = 'a';
38     ptr3->letter = 'r';
39     ptr4->letter = 'k';
40
41
42     ptr1->next = ptr2;
43     ptr2->next = ptr3;
44     ptr3->next = ptr4;
45     ptr4->next = NULL;
46
47     printList(ptr1);
48
49     system("PAUSE");
50     return 0;
51 }//end main function
52
```

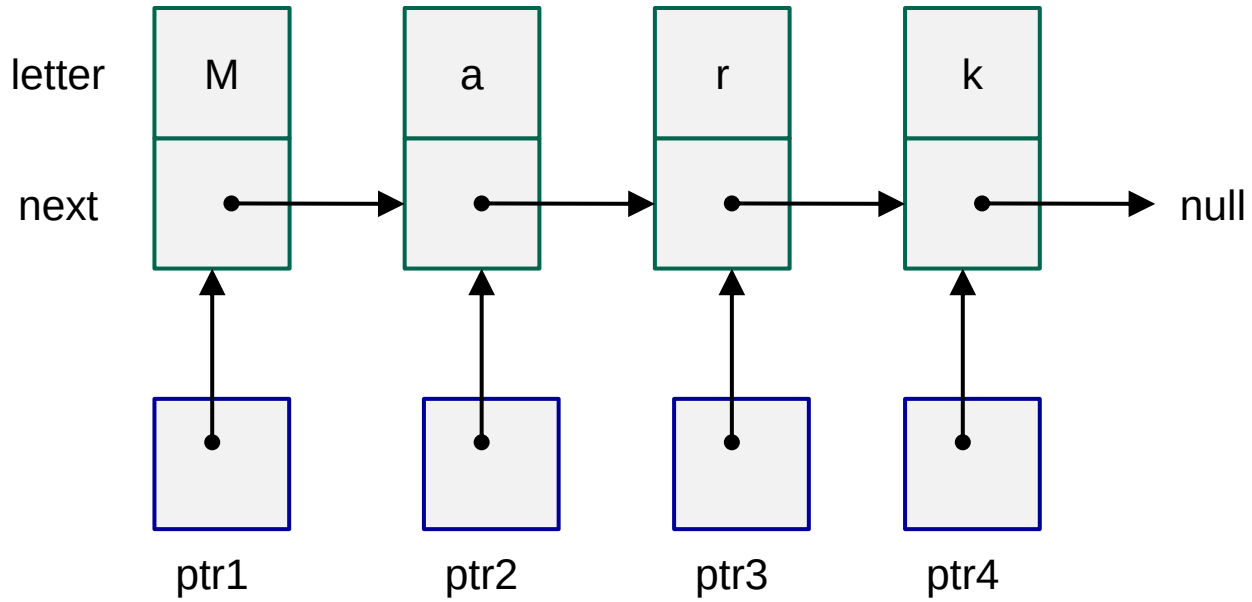
Notice that since all the variables are pointers to structures that the structure pointer operator is necessary to reference the various member fields in the structure.

Each nodes next field is assigned to point to (reference) the next node in the correct logical order in which the list is to be maintained.



A Closer Look At The Example

- In the previous example, the list nodes are constructed dynamically using `malloc`, one node at a time. The resulting structure looks like the one shown below:

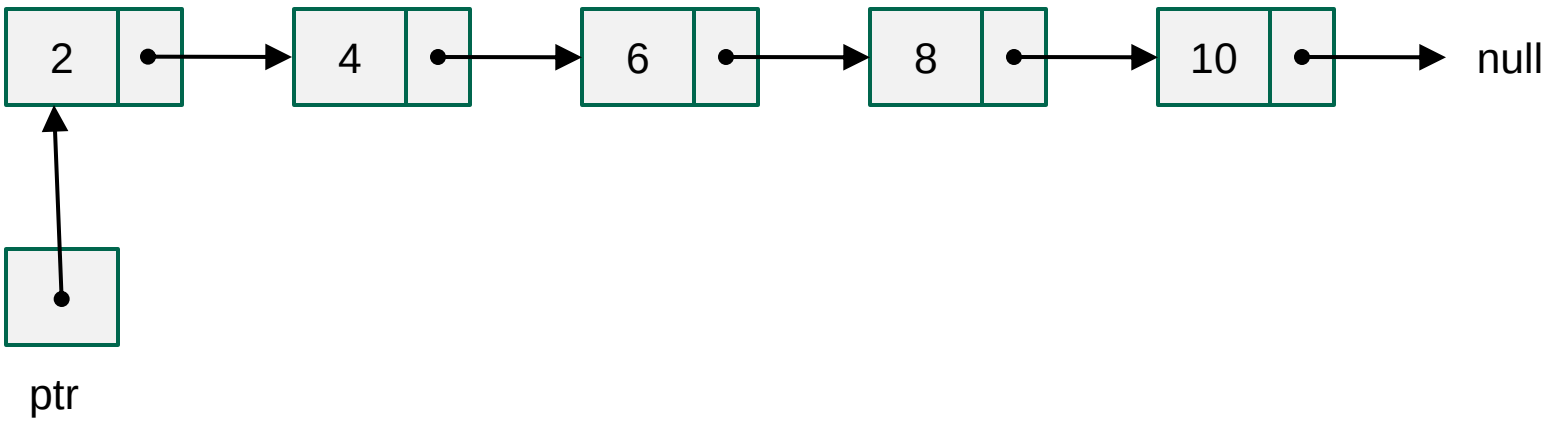


Manipulating Linked List Nodes

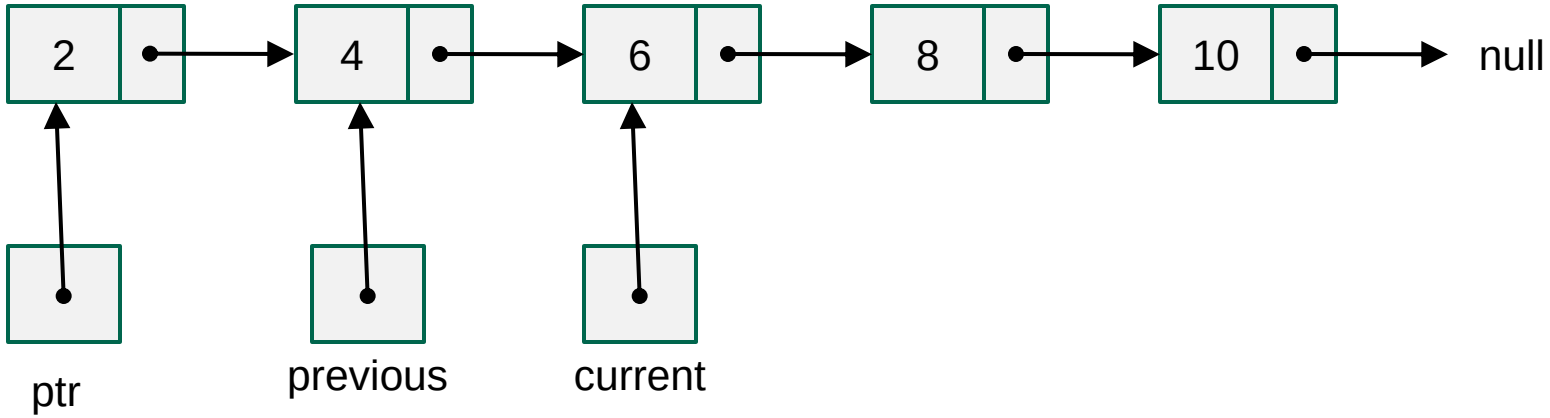
- The nodes of a linked list can be maintained in any logical order that the programmer desires for the application at hand. Alphabetical and numerical orderings are obviously quite common, but any logical ordering can be maintained (including random).
- Common operations on linked lists include, traversing the list from one end to the other or partial traversing. This is often referred to as "walking the list". Inserting new elements, deleting existing elements, and modify the contents of existing elements are also common operations.
- Let's first consider deleting a node from a list. Let's suppose in the initial list shown on the next page that we want to delete the node containing 6.

Manipulating Linked List Nodes - Deletion

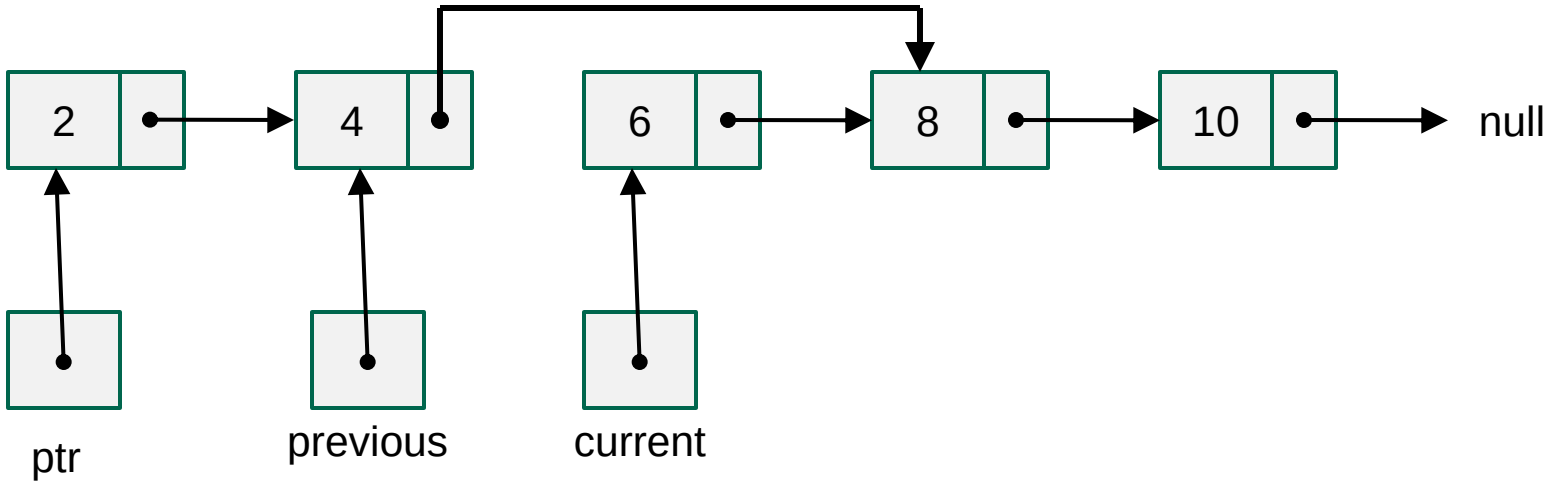
Initial list



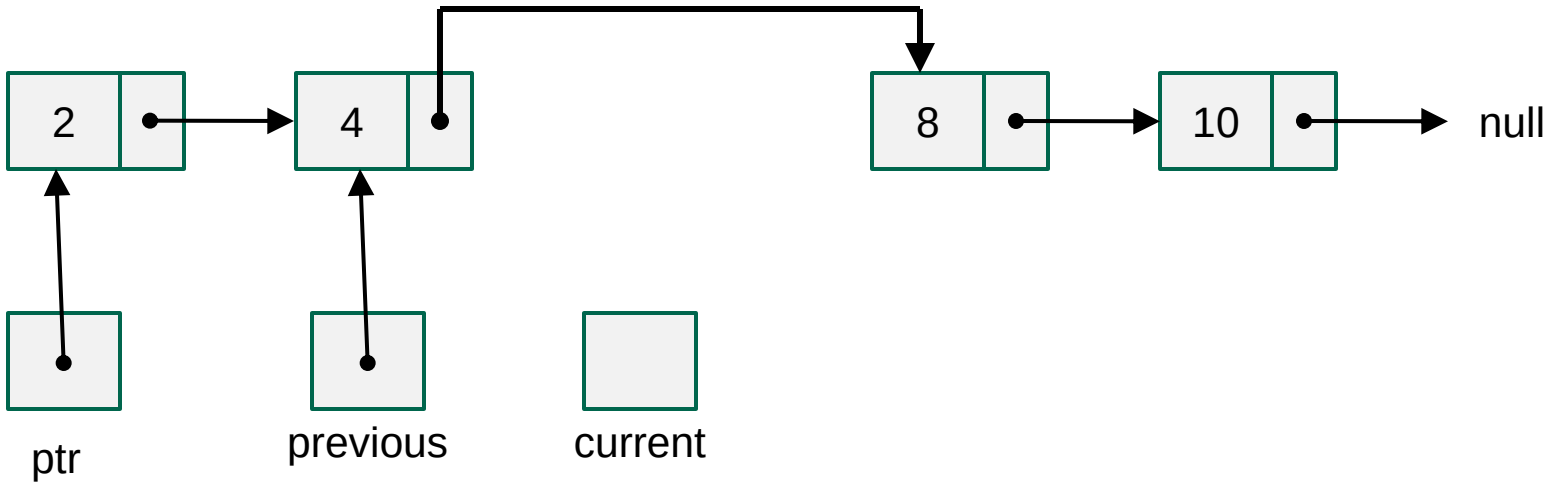
Step 1 to delete 6: walk to the node containing 6 with a trailing pointer following us



Step 2: set previous.next to current.next



Step 3: free current node

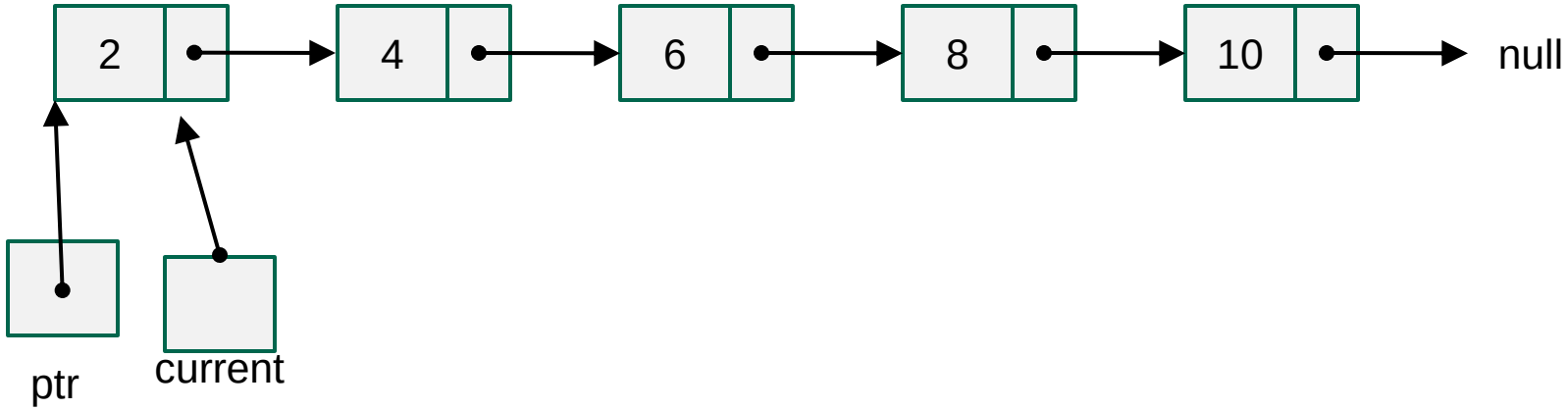


Manipulating Linked List Nodes - Deletion

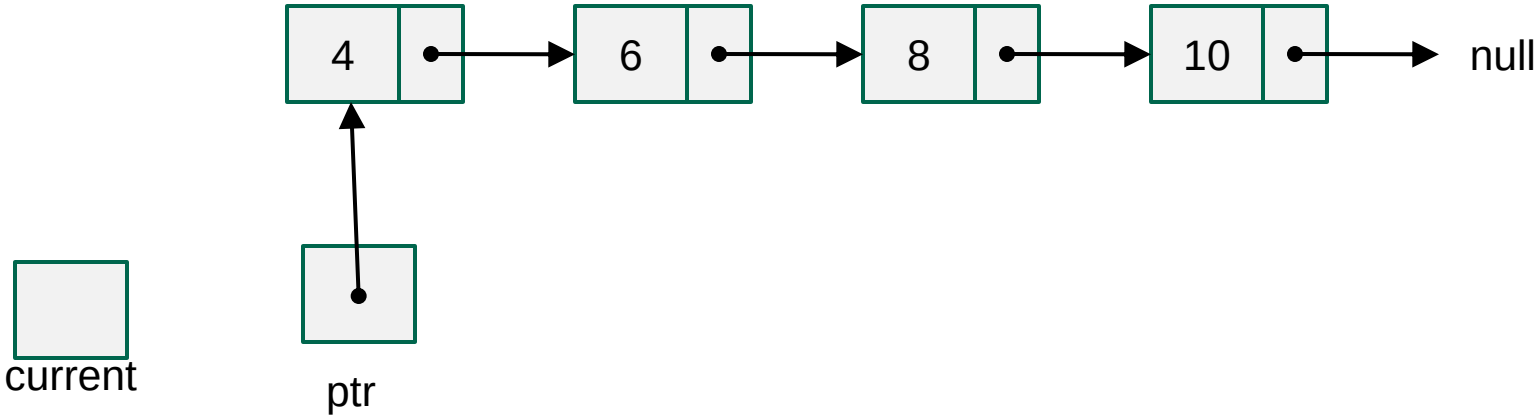
- Notice that our deletion of the node containing 6 occurred in the middle of the list. What would happen if we needed to delete the first or last node in the list? Would our technique need modification?
- The answer is yes if we need to delete the first node in the list. The answer is no if we need to delete the last node in the list.
- So, deleting the first node in the list is a special case. It's illustrated on the next page, where we need to delete the node containing 2.

Manipulating Linked List Nodes - Deletion

Step 1: walk puts ptr and current at same node



Step 2: set ptr to ptr->next and free current.

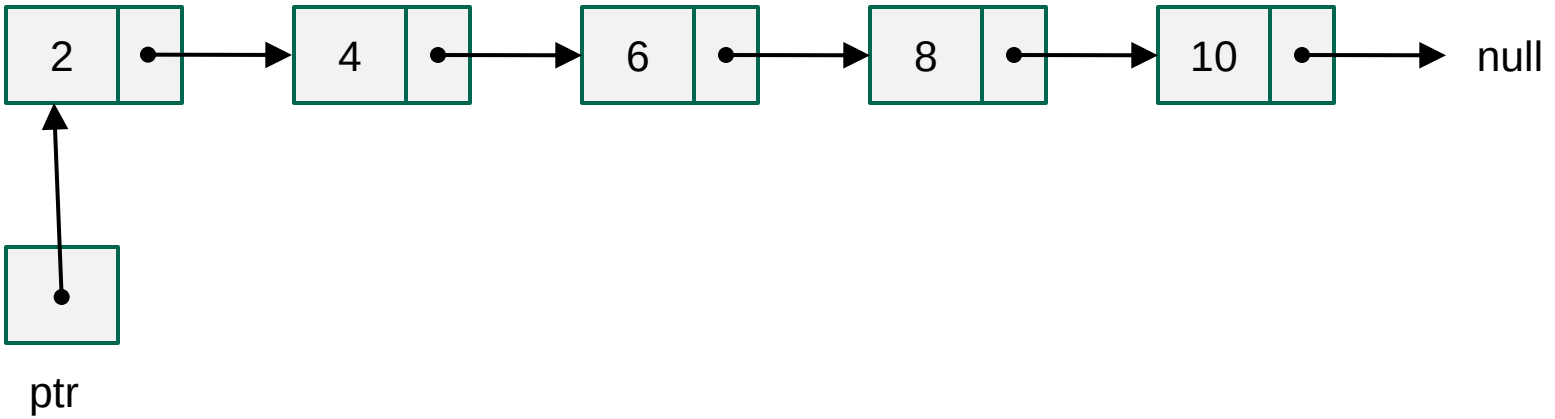


Manipulating Linked List Nodes – Insertion

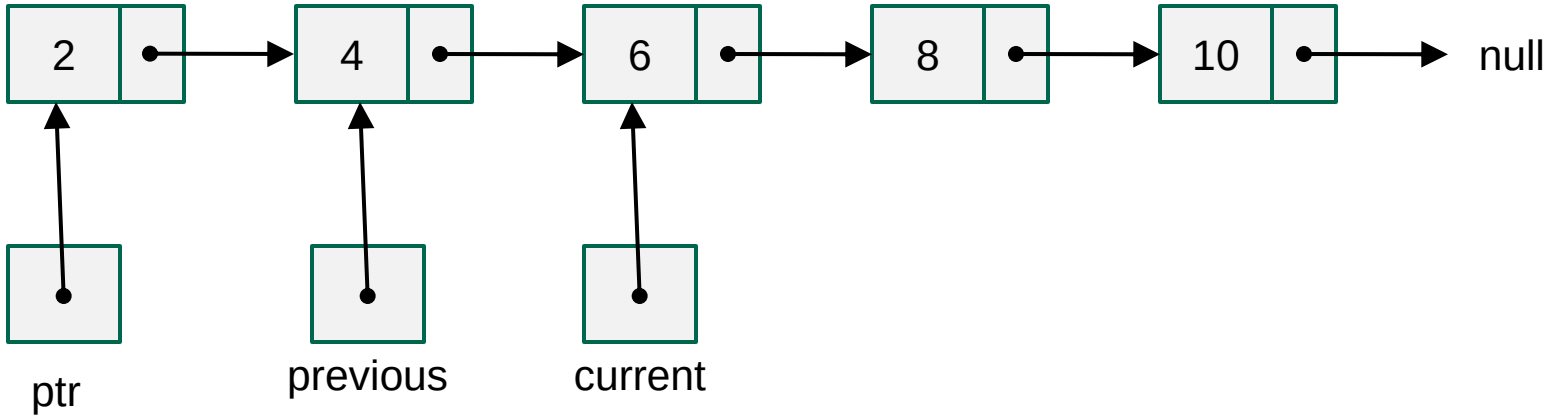
- Now let's consider inserting a new node into a list.
- As before, we'll need to walk the list to the correct insertion point and we'll maintain a trailing pointer.
- Notice that the insertion point will be between the current pointer and the trailing pointer.
- The example on the following pages illustrates a typical mid-list insertion. In this case, we want to insert a new node with the value 5 in its proper location in the list.

Manipulating Linked List Nodes - Insertion

Initial list

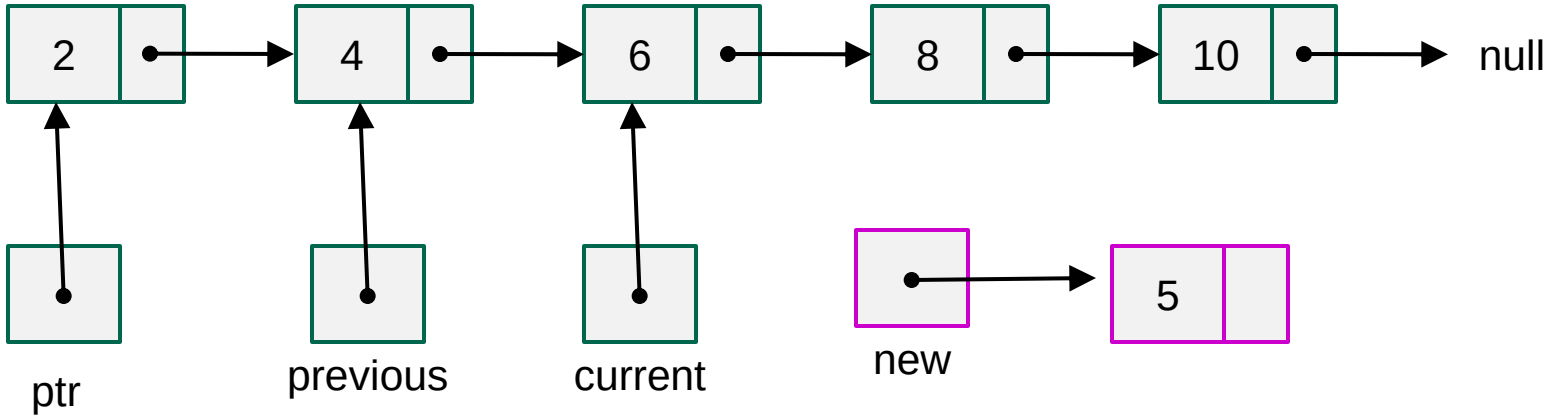


Step 1 to insert 5: walk to the node containing 6 with a trailing pointer following us

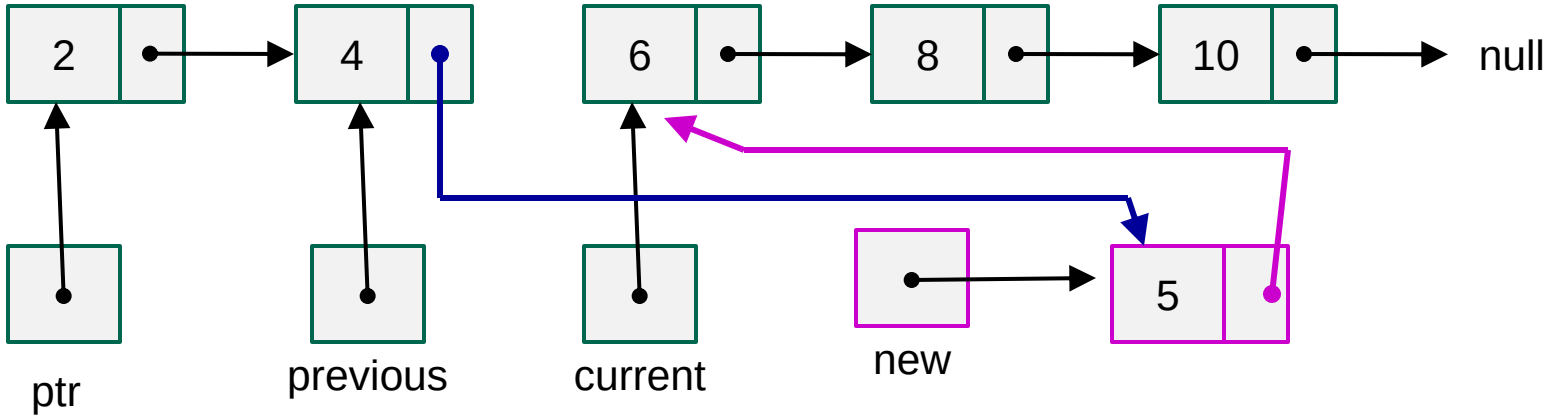


Manipulating Linked List Nodes - Insertion

Step 2: create new node.



Step 3: set previous-next to new and new->next to current.

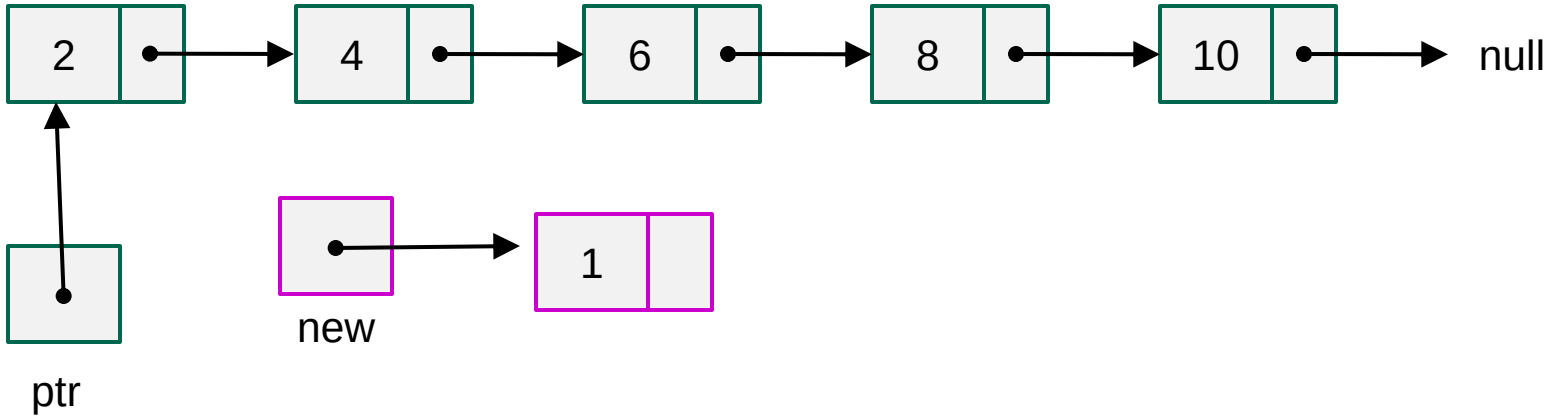


Manipulating Linked List Nodes - Deletion

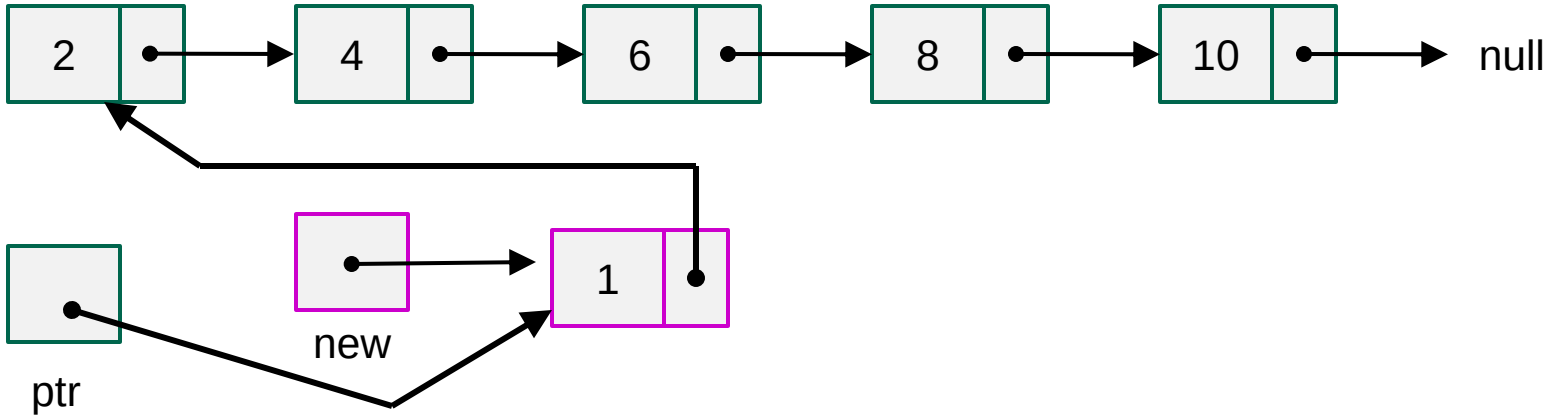
- What would happen if we needed to insert a new first or last node in the list? Would our technique need modification?
- The answer is yes in both cases this time.
- So, inserting a new first node in the list and a new last node are both special cases. They are illustrated on the next couple of pages, where we want to insert 1 and 11 into the list.

Manipulating Linked List Nodes - Insertion

Step 1: insert 1 create new node (at beginning of the list).

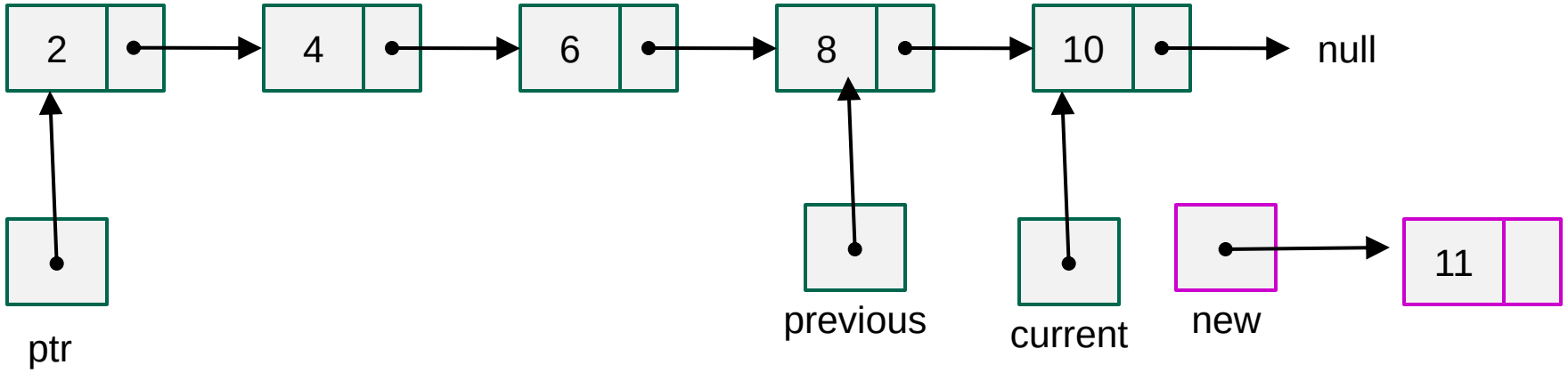


Step 2: set new->next to ptr and ptr to new.

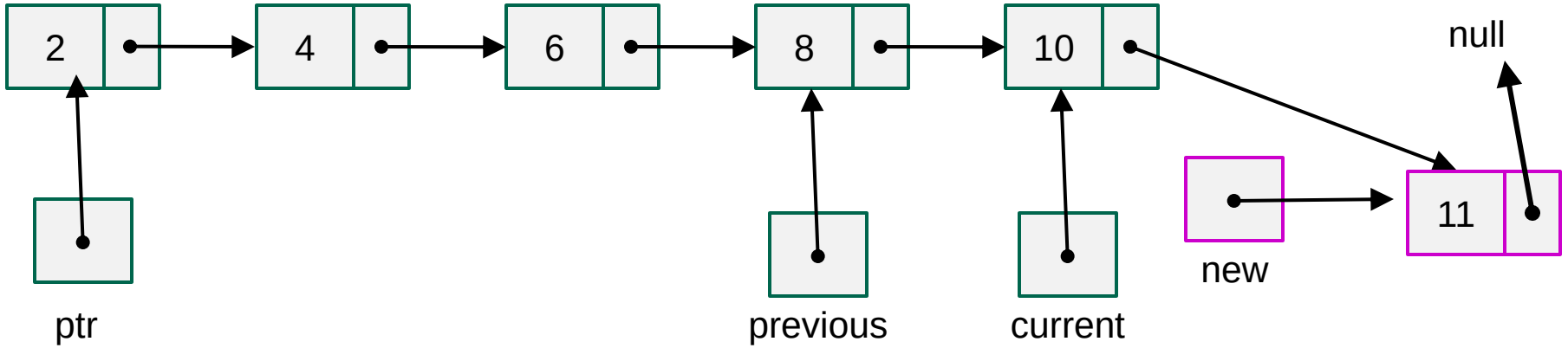


Manipulating Linked List Nodes - Insertion

Step 1: insert 11 create new node (at end of the list).



Step 2: set new->next to current->next (null) and current->next to new.



Manipulating Linked List Nodes

- The following example program combines all of these cases into a single program that will build and maintain a linked list in alphabetic ordering of the characters entered by the user.
- It is a menu driven program that gives the user the choice of inserting or deleting an element. The list is initially empty and after each operation the list is drawn out so you can see the effect of the operation.
- Again the program is too large to put into the notes (170 lines), only selected parts are shown, but the entire program is available on the web site so download it an play around with it.

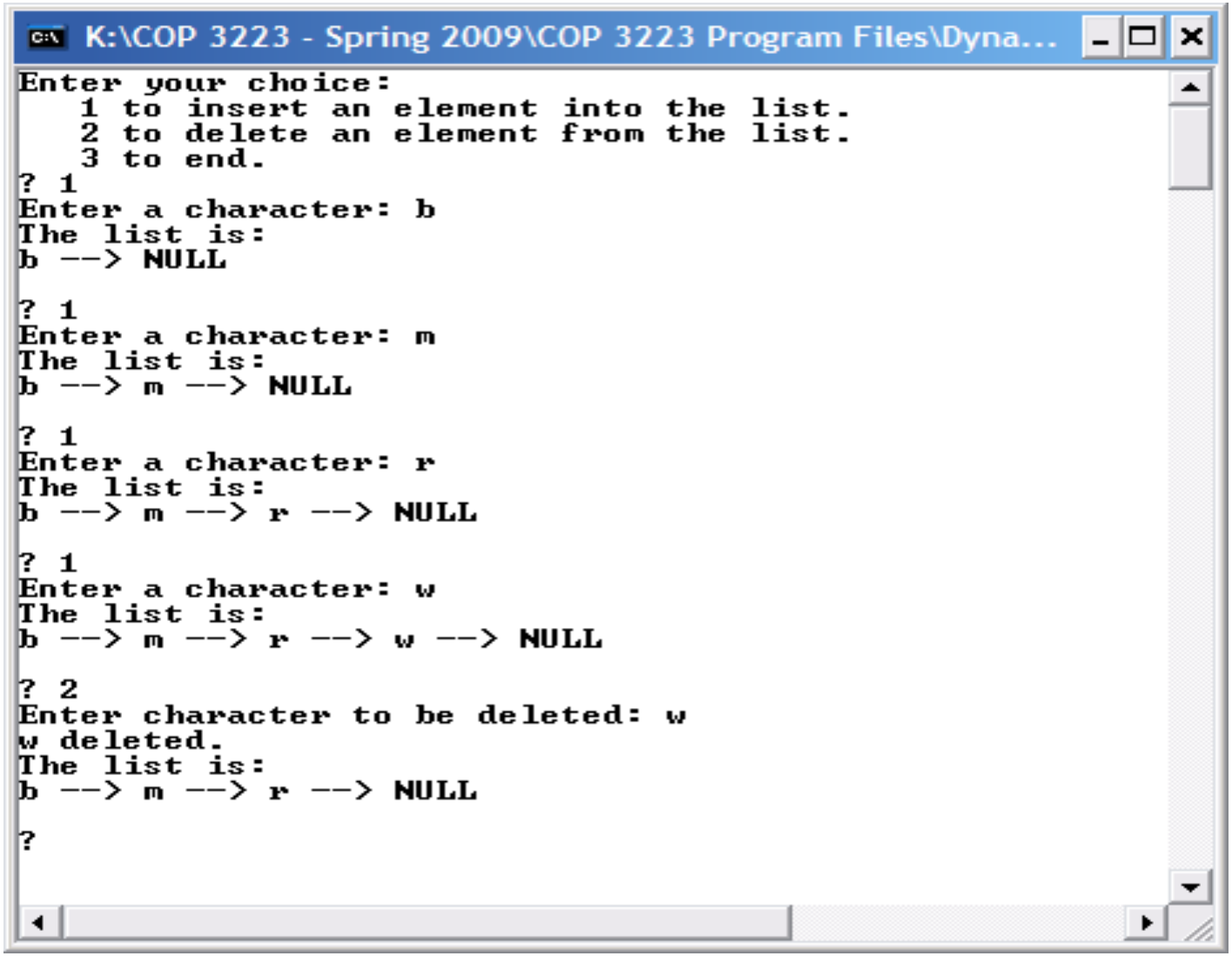
a big linked list example.c

```c
//Dynamic Structures In C - Part 2 - A large linked list program
//This program creates and maintains a list of characters in alphabetic order
//The user has two basic options: 1 add a new character to the list, and 2
//delete an existing charcater from the list
//April 23, 2009     Written by: Mark Llewellyn

#include <stdio.h>
#include <stdlib.h>
#define TRUE 1
#define FALSE 0

// self-referential structure
struct listNode {
   char data; // each listNode contains a character
   struct listNode *nextPtr; // pointer to next node
}; // end structure listNode/

typedef struct listNode ListNode; // synonym for struct listNode
typedef ListNode *ListNodePtr; // synonym for ListNode*

// draw menu
void drawMenu()
{
   printf( "Enter your choice:\n"
      "   1 to insert an element into the list.\n"
      "   2 to delete an element from the list.\n"
      "   3 to end.\n" );
} // end drawMenu function
```


a big linked list example.c

```
31 void insert(ListNodePtr *sPtr, char value)
32 {
33    ListNodePtr newPtr;       // pointer to new node
34    ListNodePtr previousPtr; // pointer to previous node in list
35    ListNodePtr currentPtr;  // pointer to current node in list
36
37    newPtr = malloc(sizeof(ListNode)); // create node
38
39    if (newPtr != NULL) { // is space available
40       newPtr->data = value; // place value in node
41       newPtr->nextPtr = NULL; // node does not link to another node
42       previousPtr = NULL;
43       currentPtr = *sPtr;
44       // loop to find the correct location in the list to insert new node
45       while ( currentPtr != NULL && value > currentPtr->data ) {
46          previousPtr = currentPtr;  //walk to ...
47          currentPtr = currentPtr->nextPtr;  //... next node
48       }// end while stmt
49       // insert new node at beginning of list
50       if ( previousPtr == NULL ) {
51          newPtr->nextPtr = *sPtr;
52          *sPtr = newPtr;
53       } // end if stmt
54       else { // insert new node between previousPtr and currentPtr
55          previousPtr->nextPtr = newPtr;
56          newPtr->nextPtr = currentPtr;
57       }// end else stmt
58    }//end if stmt
```

ListNodePtr currentPtr


K:\COP 3223 - Spring 2009\COP 3223 Program Files\Dyna...
Enter your choice:
1 to insert an element into the list.
2 to delete an element from the list.
3 to end.
? 1
Enter a character: b
The list is:
b --> NULL
? 1
Enter a character: m
The list is:
b --> m --> NULL
? 1
Enter a character: r
The list is:
b --> m --> r --> NULL
? 1
Enter a character: w
The list is:
b --> m --> r --> w --> NULL
? 2
Enter character to be deleted: w
w deleted.
The list is:
b --> m --> r --> NULL
?

THE END