ALGONQUIN COLLEGE

CST8234 – C Programming

Week 9a –Linked List

Week of October 29th, 2018

Self-Referencing Structs

- The previous lecture discussed struct vs program namespaces, and how the format of struct naming allows it to be referenced with a struct

```
typedef struct House {
    char *strStreet;
    struct House *pNeighbour;
} House;
House housesOnStreet[NUM_HOUSES];
```

= *adds 'House' to struct namespace*

= *uses struct namespace*

= *adds 'House' to program namespace*

Initializing

- We can now initialize our list of houses, so that each house points to its next door neighbour

```
for ( i = 0; i < NUM_HOUSES-1; i++ ) {
    housesOnStreet[i].pNeighbour = &housesOnStreet[i+1];
}
housesOnStreet[NUM_HOUSES-1].pNeighbour = null;
```

Last house is NULL

- The last house on the street has no neighbour, so we assign its neighbour to be NULL

How it Looks in Memory

- Here's a possible layout for our housesOnStreet

		Address	Value
house[0]	.strStreet	0x01001000	"2 Elm Street"
	.pNeighbour	0x01001004	0x01001008
house[1]	.strStreet	0x01001008	"4 Elm Street"
	.pNeighbour	0x0100100c	0x01001010
house[2]	.strStreet	0x01001010	"6 Elm Street"
	.pNeighbour	0x01001014	0x01001018
house[3]	.strStreet	0x01001018	"8 Elm Street"
	.pNeighbour	0x0100101c	NULL

Sample Application

- Problem: Print out the street addresses of all the houses
- Solution 1: Indexing into the array

```
int i = 0;
for ( i = 0; i < NUM_HOUSES-1; i++ ){
   printf( "address = %s\n", housesOnStreet[i].strStreet);
}
```

Reference to the struct:

```
typedef struct House {
char *strStreet;
struct House *pNeighbour;
} House;

House housesOnStreet[NUM_HOUSES];
```

Sample Application

- Solution 2: Incrementing a pointer.

```
House pHouse = &housesOnStreet[0];
while ( <some condition> ) {
   printf( "address = %s\n", (pHouse++)->strStreet);
}
```

Reference to the struct:

```
typedef struct House {
char *strStreet;
struct House *pNeighbour;
} House;

House housesOnStreet[NUM_HOUSES];
```

Sample Application

- Solution 3: We can make **use** of our new “pNeighbour” member

```
House pFirstHouse = &housesOnStreet[0];
while ( pHouse != null ){
   printf( “address = %s\n”, pHouse->strStreet);
    pHouse = pHouse->pNeighbour;
}
```

Reference to the struct:

```
typedef struct House {
char *strStreet;
struct House *pNeighbour;
} House;

House housesOnStreet[NUM_HOUSES];
```

How it Looks in Memory if not continuous

- Here's a possible layout for our housesOnStreet

pFirstHouse →

head Pointer ↑

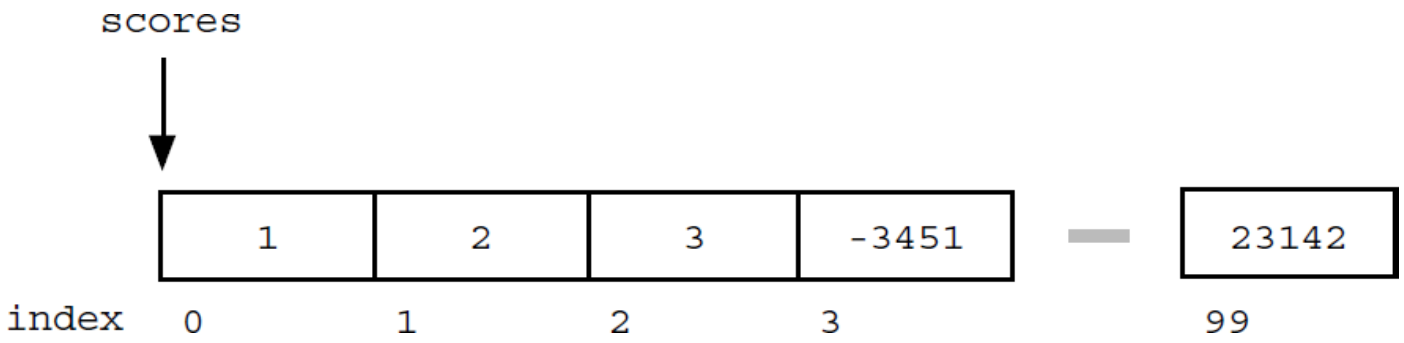
	Address	Value
.strStreet	0x01001000	"4 Elm Street"
.pNeighbour	0x01001004	0x01001008
.strStreet	0x01001008	"6 Elm Street"
.pNeighbour	0x0100100c	0x0ca81000
	:	
.strStreet	0x03301000	"2 Elm Street"
.pNeighbour	0x03301004	0x01001000
	:	
.strStreet	0x0ca81000	"8 Elm Street"
.pNeighbour	0x0ca81004	NULL

Why Not Just Use Arrays?

- Using array indexing (e.g., housesOnStreet[i]) or pointer incrementing (e.g., pHouse++) are both very simple!
- Except…
  - Have to know the start of array, in order to use an index
    - Sometimes you just have a pointer to a house, without knowing where it sits within the array
  - Have to know when to end the iteration when using a pointer
    - Remember, C arrays have no built-in knowledge of their own length!
  - All the house objects have to be in a contiguous block of memory (i.e., an array)
    - What happens if we are dynamically creating our ‘House’ structs as we need them, so they are scattered all over the heap?

Array Revisited

Most common data structure
Declaring an array `int scores[100]` , sets memory for 100 elements.



Once the array is set:

- The size of the array is fixed – 100.
- Statically memory is allocated.
- Inserting new elements is potentially expensive-elements need to be shifted.

Deletion

- If we ever need to destroy our neighbour, we can simply skip over it

```
        House pDoomedHouse = pHouse->pNeighbour;
pHouse->pNeighbour = pDoomedHouse->pNeighbour;
pDoomedHouse->pNeighbour = NULL;      // no longer in the list
```

- We have now shrunk the size of the list by one… with very little effort
- In contrast, to splice an item out of the middle of an array

```
        House pDoomedHouse = pHouses[iDoomed];
        /* shift all records up one spot */
while ( iDoomed < n-1 )
    pHouses[iDoomed] = pHouses[++iDoomed];
pHouses = realloc( pHouses, (--n) * sizeof( House * ));
```

- Again, we have to do
  - a lot of copying to shift items up AND
  - we have to realloc (which does even more copying) AND
  - we have to make sure we update the number of items

Splitting a list

- Frequently, you may want to divide a list into two halves…
- Easy with a list…

```
House *pEndFirstList= getHouseByIndex( getHouseCount()/2 );
House *pSecondList = EndFirstList->pNeighbour;
pEndFirstList->pNeighbour = NULL;
```

- With arrays, this requires
  - Allocating space for the second list
  - Copying over all the items
  - Reallocing the first list
  - Updating the new size of the first list
  - Recording the size of the second list

Turning into a Circular Buffer

- You can simply set the neighbour of the last house to be the first house

```
pLastHouse->pNeighbour = pFirstHouse;
```

- Now you can merrily iterate forever, over the fixed number of houses, without having a specific beginning or end

```
pHouse = pHouse->pNeighbour
```

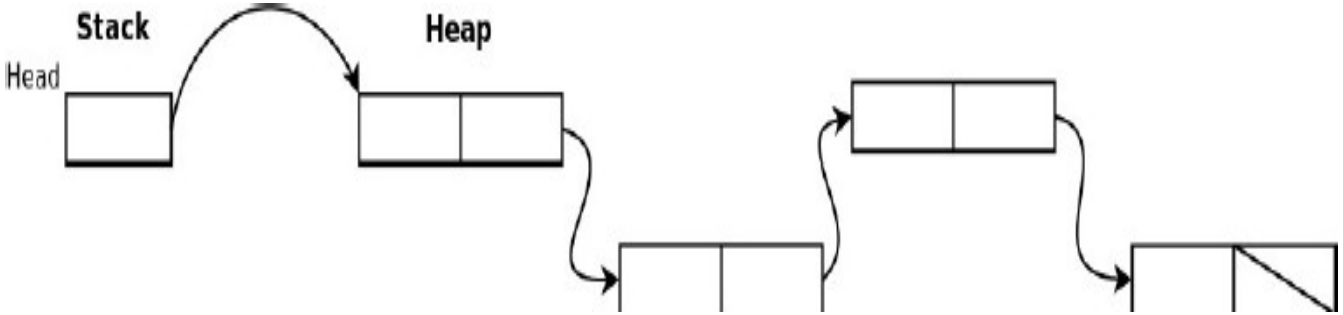
- With an array, you could accomplish the same thing with

```
iHouse = (++iHouse)%n;
```

- But you still need to know the start of the array, and know the size

Linked Lists

Everything discussed so far is essentially describing a “linked list” and its benefits
We usually talk about “Nodes” when talking about Linked Lists.



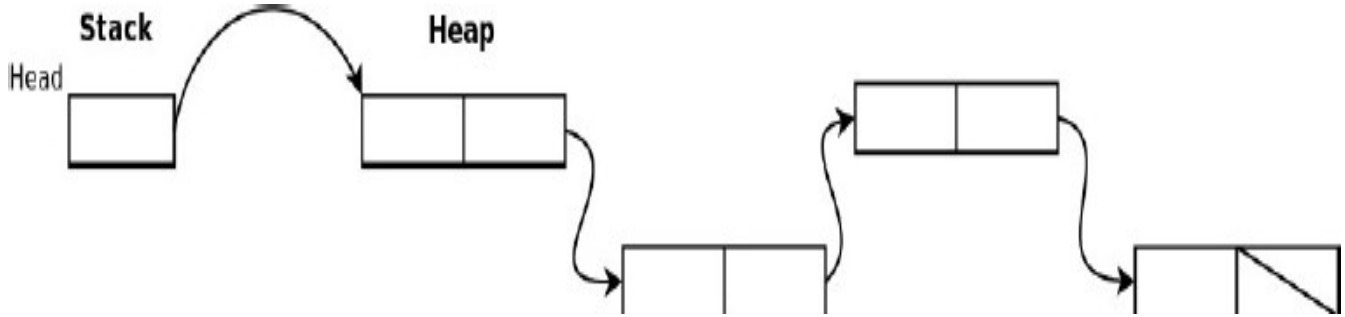
Linked Lists — cont'd

A linked list is a data structure based in a set of dynamically allocated nodes, arranged in such a way that each node contains:

- One value or data element with the main information.
- One pointer or a reference (memory address) to next element of the list (node)
- The pointer always points to the next member of the list.
- If the pointer is NULL, then it is the last node in the list.

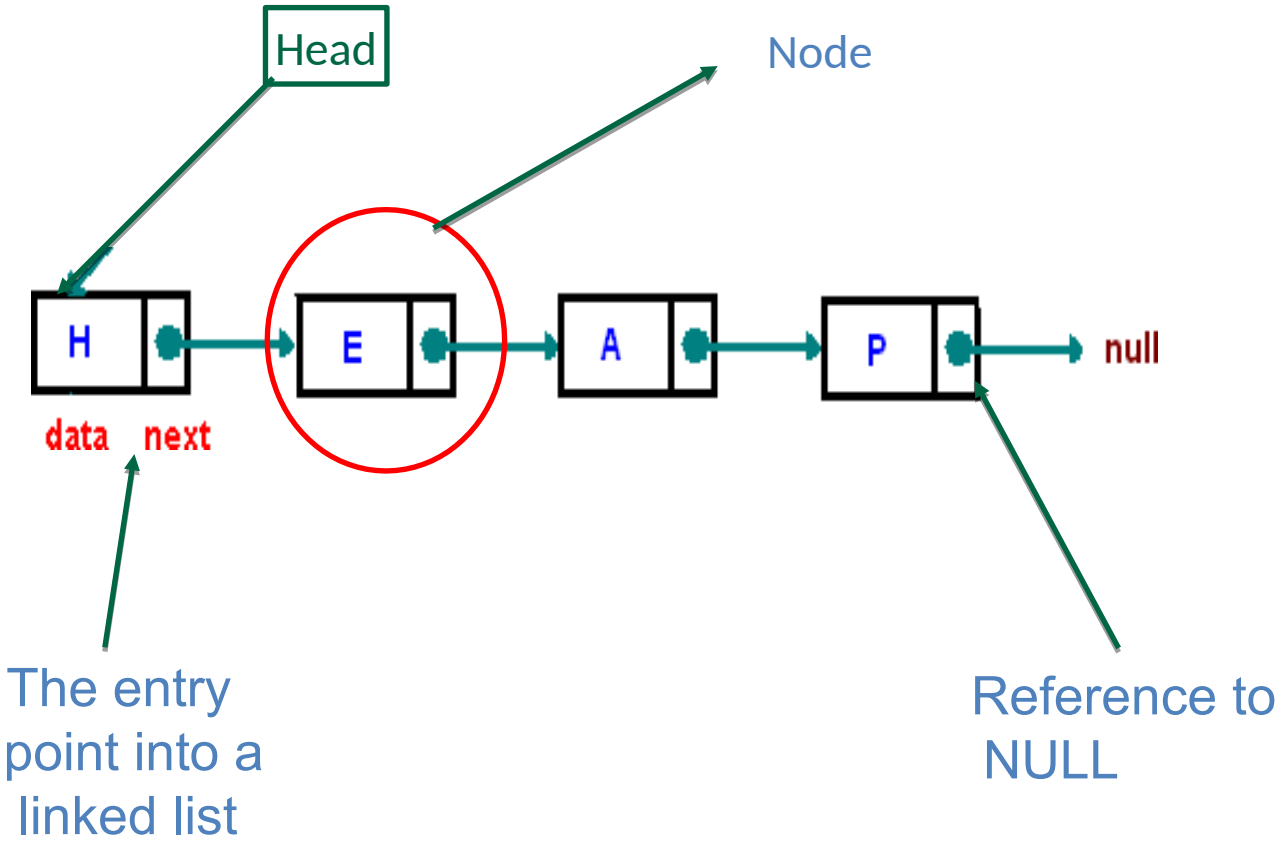
A linked list is held using:

- A local pointer variable which points to the first item in the list, usually called **`head`**.
- If that pointer is also NULL, then the list is considered to be empty.



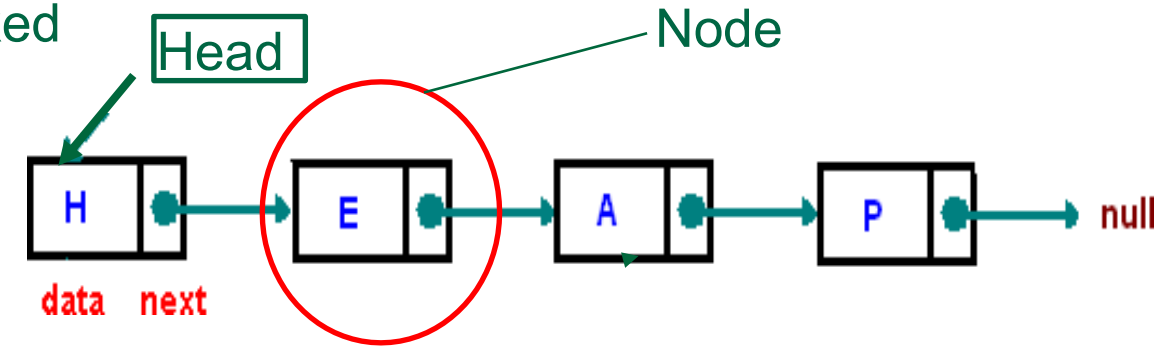
Linked List Terminology

Is a linear data structure (each element is a separated object).

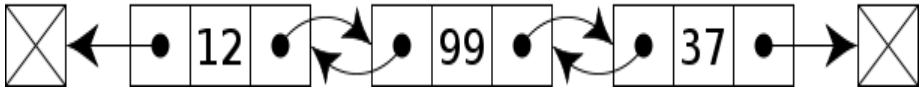


Linked List variation

Singly Linked List:



Doubly Linked List



Node
- (1) Data
- (2) Link forward
- (3) Link backward

Why Linked List?

- Linked lists and arrays are similar, both store collection of data.
- Well first, in Java and C++ there are dynamic arrays called “vectors”.
- Vectors are good for efficient random read access, traversing is efficient.
  - Example: obtaining the last node or finding a node.

- Benefits of Linked lists:
  - Linked Lists are more efficient memory (CPU usage, memory).
  - Linked Lists are good for inserting and deleting elements in front or back
  - Simple to implement.

Linked Lists: Node

Let's define a linked list node using struct:

```
typedef struct node {
    int val;
    struct node * next;
} node_t;
```

Notice that

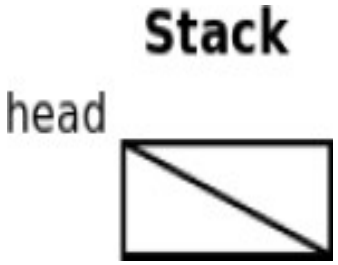
- We are defining the struct using self referencing struct (in a recursive manner)
- Let's name our node node_t.

Linked List: Empty List

A list with no elements

It is represented with the value of NULL as the head pointer.

Working with lists, first check if the list is empty before adding any elements.



Creating Nodes: Head

Now we can use the node. Let's create a local variable which points to the first item of the list (called head).

```
node_t * head = NULL;
head = (node_t *) malloc(sizeof(node_t));
if (head == NULL) {
   return 1;
}

head->val = 1;
head->next = NULL;
```

- We've just created the first node in the list.
- We must set the next item to be empty, if we want to finish populating the list.
- We should always check if malloc returned a NULL value or not.

Creating Nodes: 2nd Node

To add a variable to the end of the list, we can just continue advancing to the next pointer:

```
node_t * head = NULL;
head = malloc(sizeof(node_t));
head->val = 1;
head->next = malloc(sizeof(node_t));
head->next->val = 2;
head->next->next = NULL;
```

- After 2nd Node, we may add as many nodes as we want.
- We have to make sure that, in the list, `lastNode->next` will be NULL.

Iterating over a Linked list

Let's build a function that prints out all the items of a list. To do this:

- We need to use a current pointer that will keep track of the node we are currently printing at.
- After printing the value of the node, we set the current pointer to the next node, to advance in the list.
- We iterate until we've reached the end of the list (the next node is NULL).

```
void print_list(node_t * head) {
    node_t * current = head;
    while (current != NULL) {
        printf("%d\n", current->val);
        current = current->next;
    }
}
```

Counting Nodes of a Linked list

To count the number of nodes in a list, iteration over the complete list:

- Iteration process finish until NULL pointer is reached.
- Head (the head of the list) cannot be displaced.

- Declare a pointer (usually refer as current) and point it to the head.
- Declare and initialize a counter variable to zero.

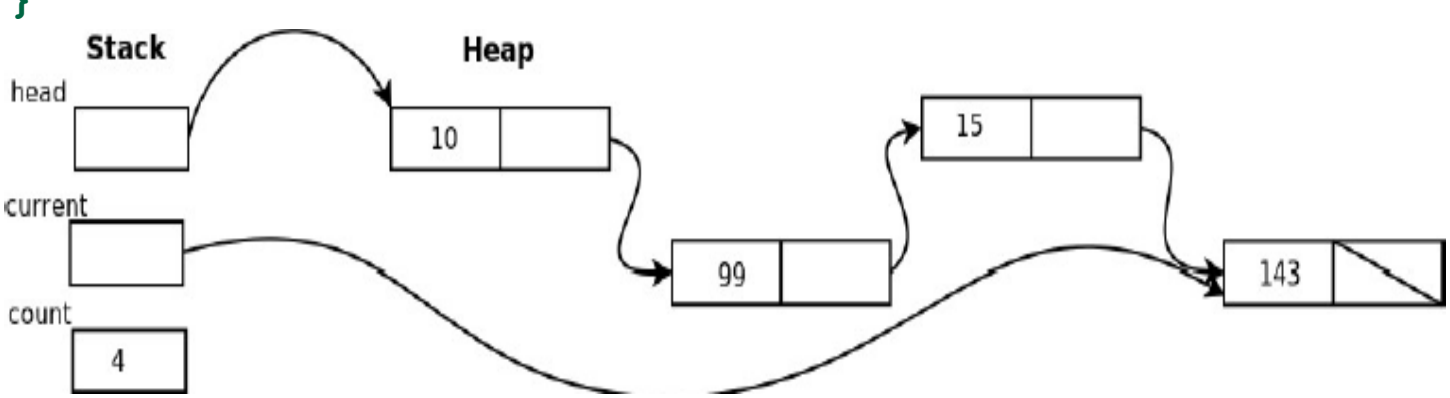
```
node_t * current = head;
int count =0;
```

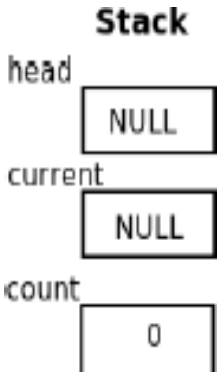
Counting Nodes of a Linked list

While current is not NULL:

- increment your counter
- move to the next node

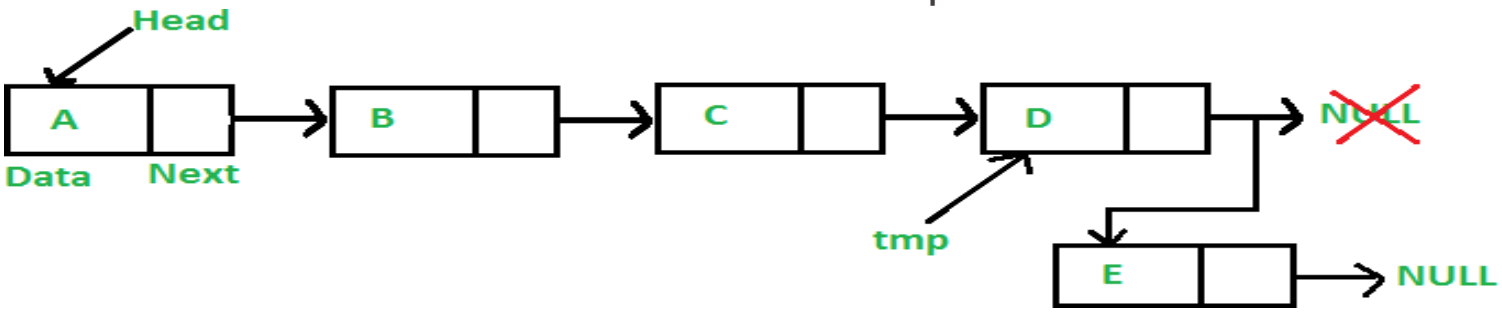
```
node_t * current = head;  int count =0;
while (current != NULL){
count++;
current = current->next;
}
```





Adding a Node at the end of the Linked list

- We use a current pointer and we set it to start (i.e., head)
- Next, in each step, we advance the pointer to the next item in the list, until we reach the last item.
- We add the new node reference to the last node next pointer



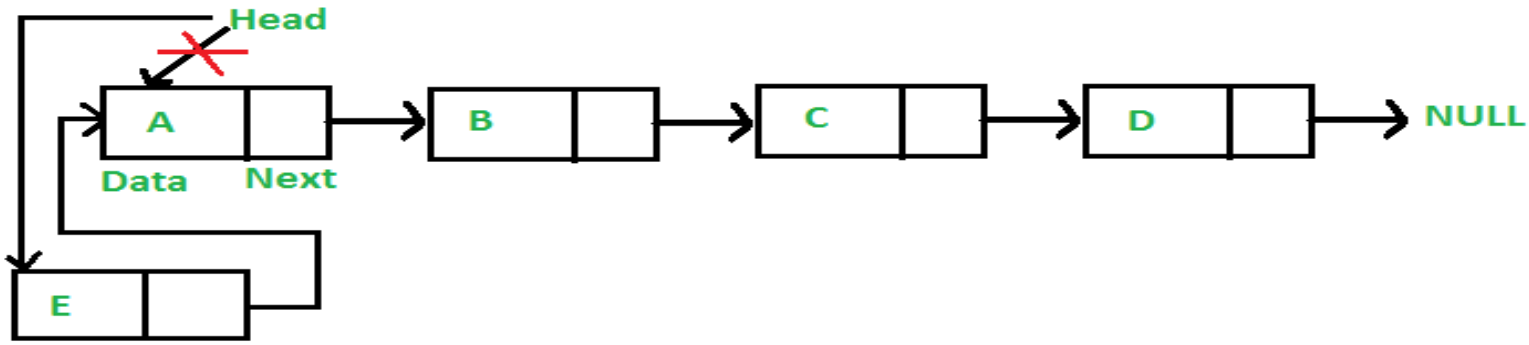
Adding a Node at the end of the Linked list - cont'd

```
void pushEnd(node_t * head, int val) {
   node_t * current = head;
   while (current->next != NULL){
      current = current->next;
    }

   /* now we can add a new node next to node D */
   current->next = malloc(sizeof(node_t));
   current->next->val = val;
   current->next->next = NULL;
}
```

Adding a Node at the start of the Linked list

- Create a new Node and set its value
- Set that the new Node ->next points to the current head of the list
- Set the new Node as the head of the list



Adding a Node at the start of the Linked list — cont'd

```
void pushStart(node_t ** head, int val) {
    node_t * new_node;
    new_node = malloc(sizeof(node_t));
    new_node->val = val;
    new_node->next = *head;
    *head = new_node;
}
```

Deleting the first Node of the Linked list

- Take the next Node that the head points to and save it
- Free the current head Node
- Set the saved Node to be our new head Node

```
int popStart(node_t ** head) {
    int retval = -1;
    node_t * next_node = NULL;
    if (*head == NULL) {
     return -1;
    }
    next_node = (*head)->next;
    retval = (*head)->val;
    free(*head);
    *head = next_node;
    return retval;
}
```

Deleting the last Node of the Linked list

- Deleting the last Node from a list is very similar to adding it to the end of the list, but with one big exception
- Since we have to change one Node before the last Node……..,
- We actually have to look two Nodes ahead and see if the next Node is the last Node in the list

```
int removeLast(node_t * head){
    int retval = 0;
    /* if there is only one item in the list, remove it */
    if (head->next == NULL){
        retval = head->val;
        free(head);
        return retval;
    }
```

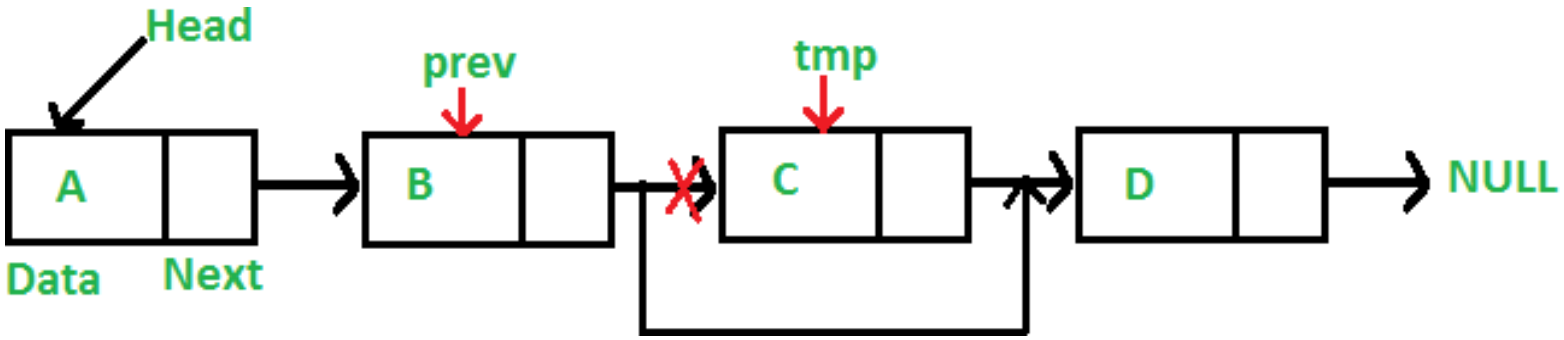
Deleting the last Node of the Linked list ….

```
/* get to the second to last node in the list */
   node_t * current = head;
   while (current->next->next != NULL) {
      current = current->next;
   }

 /*points to the second to last item of the list,  so let's
remove current->next */
    retval = current->next->val;
    free(current->next);
    current->next = NULL;
    return retval;
}
```

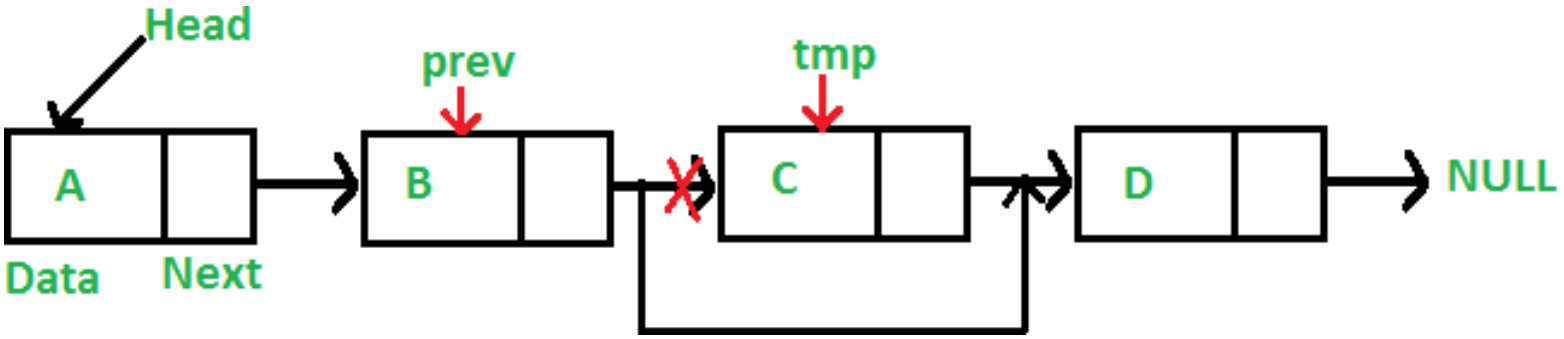
Deleting a specific Node from the Linked list ….

- To remove a specific item from the list,
  - either by its index from the beginning of the list or
  - by its value,
- we will need to go over all the items, continuously looking ahead to find out if we've reached the node before the item we wish to remove.
- This is because we need to change the location to where the previous node points to as well.



Deleting a specific Node from the Linked list ….

- Iterate to the node before the node we wish to delete
- Save the node we wish to delete in a temporary pointer
- Set the previous node's next pointer to point to the node after the node we wish to delete
- Delete the node using the temporary pointer



Deleting a specific Node from the Linked list ….

```
int remove_by_index(node_t ** head, int n){
    int i = 0;
    int retval = -1;
    node_t * current = *head;
    node_t * temp_node = NULL;
    /* index n = 0 means first node */
    if (n == 0) {
        return pop(head);
    }
    for (i = 0; i < n-1; i++) {
        if (current->next == NULL) {
            return -1;
        }
        current = current->next;
    }
    temp_node = current->next;
    retval = temp_node->val;
    current->next = temp_node->next;
    free(temp_node);
    return retval;
}
```

Doubly-Linked Lists

- So far, our linked-list has only been good for traversing one-way down our list
- But there's nothing stopping you from having TWO self-referential pointers in your struct
  - ```
    typedef struct Node {
      /* some members */
      struct Node *pPrev;
    struct Node *pNext;
    } Node;
    ```
    = *points to my left-hand neighbour*
    = *points to my right-hand neighbour*
    - The "head" of our list has a NULL 'pPrev' pointer (no one before me)
    - The "tail" of our list has a NULL 'pNext' pointer (no one after me)

- Now we can go from beginning to end, or from end to beginning
  - `pNode = pNode->pNext;` = *traversing towards end of list*
  - `pNode = pNode->pPrev;` = *traversing towards head of list*
    ```