Doubly Linked List

Prepared by: Reginald Neil Recario
rncrecario@gmail.com
Institute of Computer Science
Feb 2012

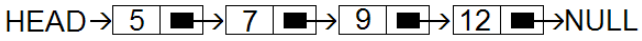
Doubly Linked List

One common problem encountered in a singly linked list is that when we move the head pointer [without the aid of any extra pointer variables] to the next node, the original node [preceding node] becomes inaccessible. This is particularly evident to novice programmers who have not yet mastered the concept of linked list. Below is a depiction of the said problem:

Note: We will be using the data type nd [which is declared below].

```
typedef struct node{
      int x;
      struct node * next;
}nd;
```

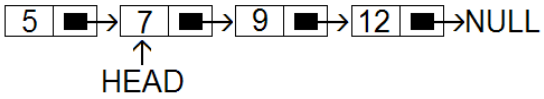
Let us assume that we already have a [singly] linked list built [assume HEAD is declared as a pointer to data type nd] and it looks like this:



Now suppose we have this statement:

```
head = head->next;
```

Obviously it will result to

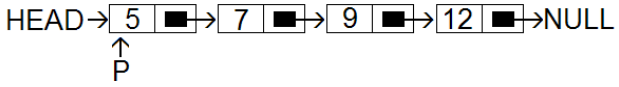


At this point, all nodes are accessible except node containing 5.

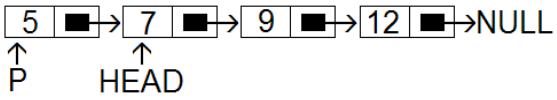
In our singly linked list discussion, we addressed this problem by either using an extra variable to point to the next node or let the head pointer point to the next node but an extra variable holds the memory of the first node. Just like this example:

[Assuming extra variable p is declared we have:]

```
p = head;
```




```
head = head->next;
```



Note that the node containing 5 is still accessible [because of pointer p].

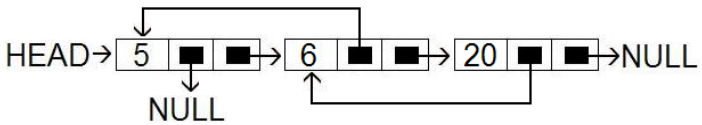
Now suppose I insist that we only use a single pointer variable [which is head]. How can we "move" the head pointer to the next node but still, can access the "previous" nodes?

An answer to this problem is the **doubly linked list**.

Definition of Doubly Linked List

Definition (*wikipedia*):
A **doubly linked list** [or **two-way linked list**] is a more sophisticated kind of linked list. Each node has two links: one points to the previous node, or points to a null value or empty list if it is the first node; and one points to the next, or points to a null value or empty list if it is the final node.

Below is an example of a doubly linked list.



Since the nodes have two pointers (one pointing to the next node, the other to the previous node), we need to update our data type nd:

```
typedef struct node{
      int x;
      struct node * prev; //pointer to
//the previous node
      struct node * next; //pointer to
//the next node
}nd;
```

Operations on Doubly Linked List

- Insertion of a node
- Deletion of a node
- *View of node data*
- *Search*
- *Updating of node data*

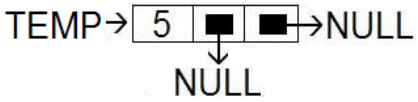
We will discuss a pseudo code of each operation [except those italicized].

Before we proceed to the pseudo code, let us have these assumptions: [1] a data type nd has been created [as declared above], [2] aside from head, a temporary variable temp and p are used primarily to point to a newly created node and for traversing respectively, and [3] initially, both head temp and p are pointing to NULL [so as temp->prev and temp->next]. Head may not be NULL [depending on the example].

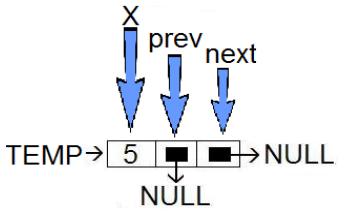
With these things in mind, let us proceed to the pseudo code.

Insertion of a node

Let us assume that temp points to a newly created node and the x component has been initialized to 5.



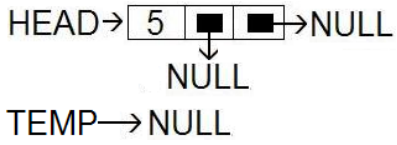
Take note that:



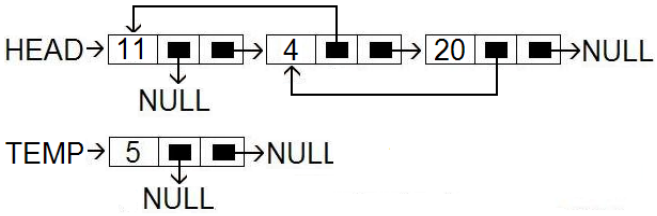
Cases in addition:

[1] If the list is initially empty (meaning head is initially NULL)

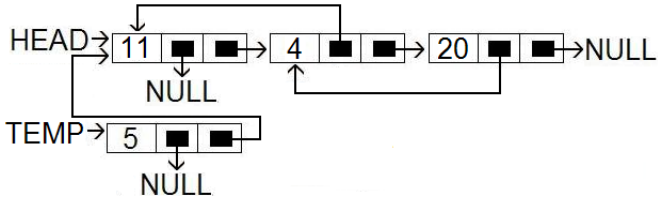
Solution: Allow head pointer to point to temp.
Set temp to NULL (optional).



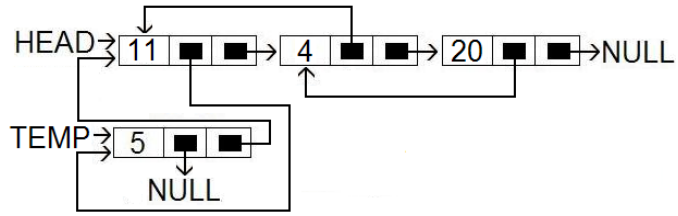
[2] If the list contains nodes but the newly created node is to be inserted before the first node of the list.



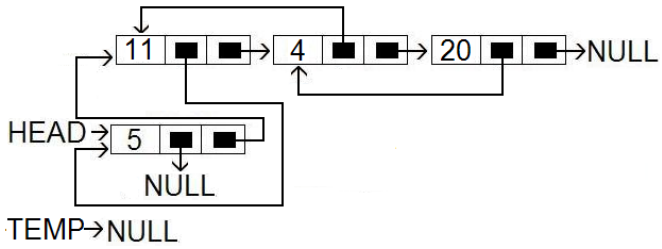
Solution: Allow `temp->next` to point to the memory location pointed to by head.



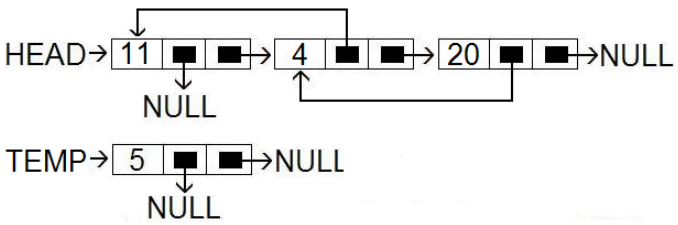
Allow `head->prev` to point to the memory location pointed to by temp.



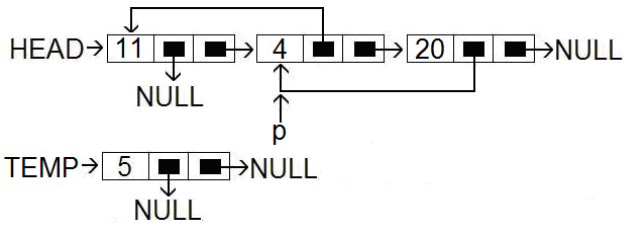
Allow `head` to point to the memory location pointed to by temp. Optionally set temp to `NULL`.



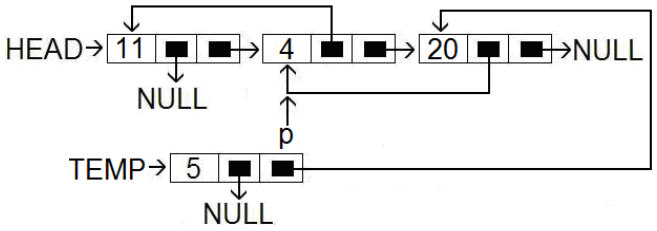
[3] Insertion in the middle of the list. Suppose we would like to insert the newly created node in between the node containing 4 and 20.



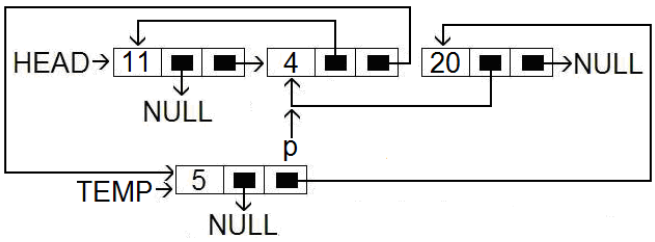
Solution: Search first where the new node is to be inserted. Allow pointer `p` to point the location.



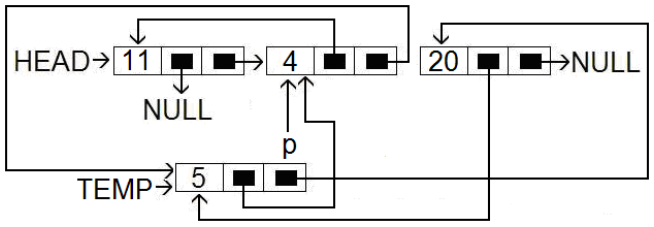
Allow `temp->next` to point to the location pointed to by `p->next`.



Allow `p->next` to point to the memory location pointed to by `temp`.

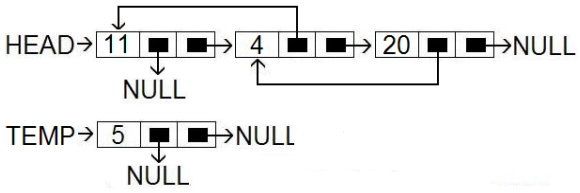


Are we done? No, not yet. Remember that we have to set the "prev" component of the affected nodes. Hence, we need to set `temp->prev` to point to the node preceding it [which is pointed by `p`]. Also, the "next" component of the node containing 20 should no longer point to 4 [since its preceding node is the node containing 5]. Thus, we set `temp->next->prev` to point to the memory location pointed to by temp.

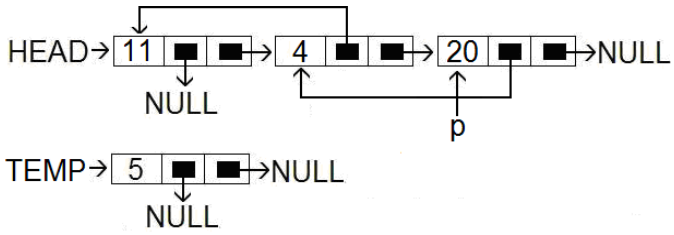


Optionally set `temp` and `p` to `NULL`.

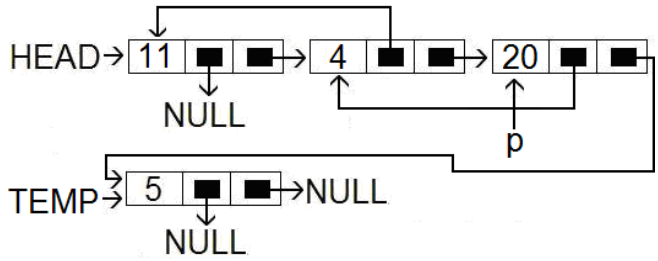
[4] Insertion at the end of the list.



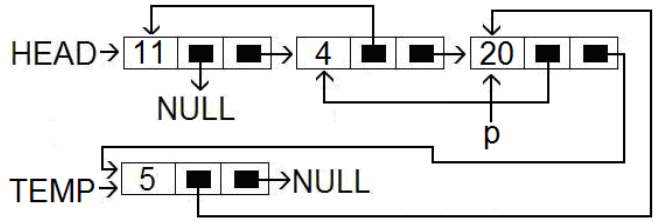
Solution: Find the end of the list [check if the "next" component is equal to `NULL`]. Allow another pointer `p` to point to that memory location.



Allow `p->next` to point to the memory location pointed to by `temp`.



Allow `temp->prev` to point to the memory location of the last node in the original list [in this case, it was pointed to by `p`].

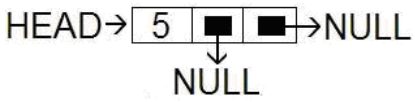


Deletion of a Node

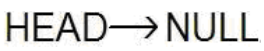
There are four cases of deletion:

1. Deleting a (doubly) linked list with a single node.
2. Deleting the first node of the linked list.
3. Deleting the node at the middle of the list.
4. Deleting the last node.

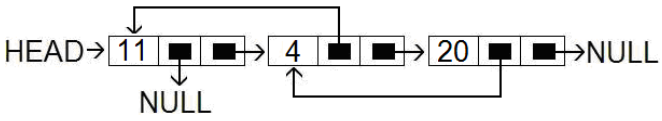
[1] Deleting a (doubly) linked list with a single node.



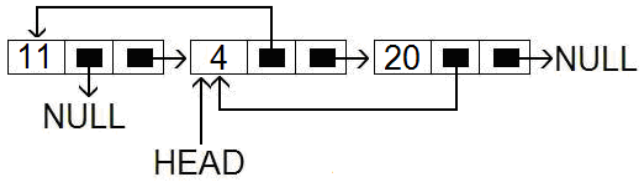
Solution: Simply free the memory location pointed to by `head` and set it to `NULL`.



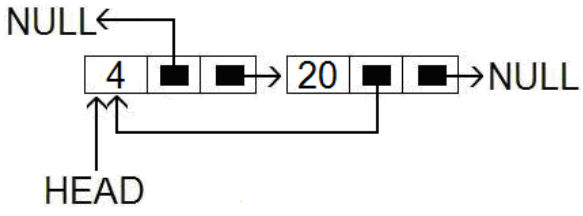
[2] Deleting the first node of the linked list..



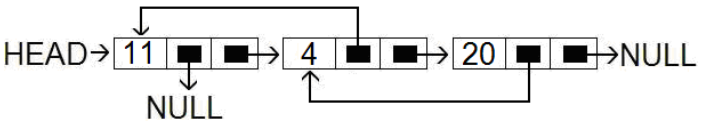
Solution: Allow `head` to point to the next node.



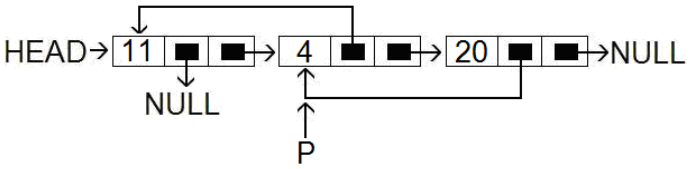
Free the containing 11 using head [`head->prev`]. Set `head->prev` to `NULL`.



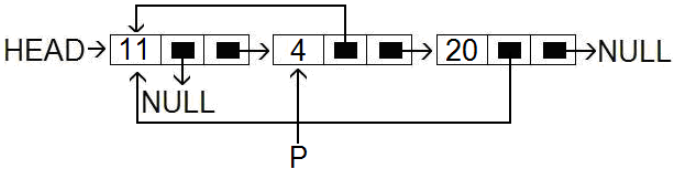
[3] Deleting the node at the middle of the list. Suppose we want to delete the node containing 4.



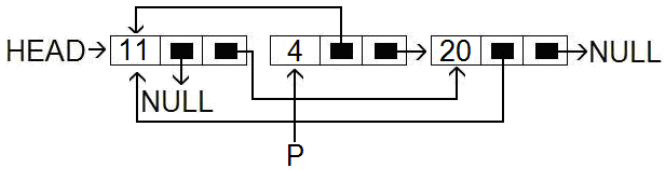
Solution: Allow a pointer `p` to traverse until it points to the node of interest.



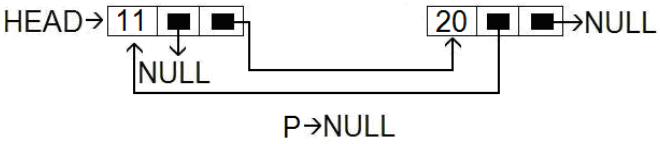
Allow the "next" component of the node containing 20 to point to the node containing 11.



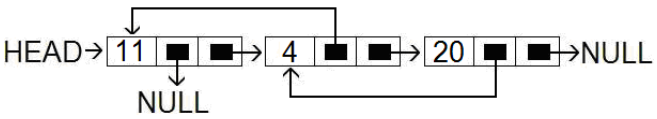
Using pointer `p`, allow the node containing 11 to point to the node containing 20.



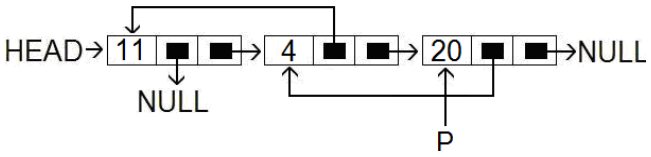
Free the memory pointed to by `p`. Set `p` to `NULL`.



[4] Deleting the last node.

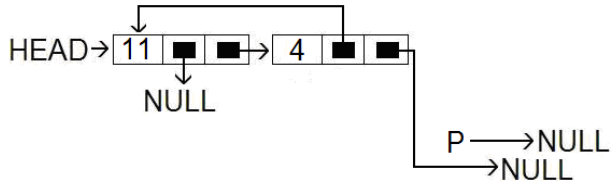


Solution: Using an extra pointer `p`, traverse the list until we reach the last node [we know that we reached the last node when `p->next` is equal to `NULL`].



Using `p`, set the "next" component of the node containing 4 to `NULL`.

Free the memory location pointed to by `p`. Set `p` to `NULL`.



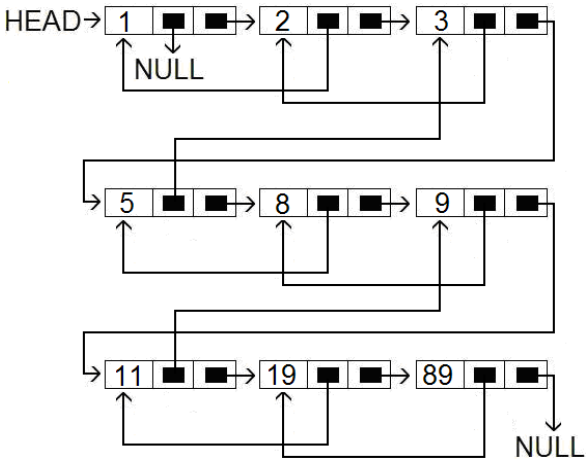
Common Problems Encountered on Doubly Linked List.

1. Forgetting to update the "prev" component of a node.
2. Mastery of what the next and prev are referring to [ex: what node does `head->next->prev` refer to?
3. Syntax [Programming]
4. Creating conditions [especially when there is a need to traverse the nodes].
5. Incorrect comparisons. Example would include `while(head->next != 9)` [`head->next` is pointer to the structure and cannot be compared to an integer], `head !=3` [same reason as the first], etc.
6. Forgetting to set the "next" component of the last node to `NULL`.

Hence, one should take note of the following problems. Some of these problems are also the problems of most students in other forms of linked lists.

MASTERY OF DOUBLY LINKED LIST

Refer to the linked list . The data type [for doubly linked list] was used. Only three pointers are available: head, p, and temp.



Q1: What value does head->next->prev->next->prev->x refer to? X=_____.

Q2. What value does head->next->next->next->prev->x refer to? X=_____.

Q3. What value does head->next->next->next->next->prev->prev->x refer to? X= _____.

Q4. To node containing what does this statement mean: head->next->next->next->prev->next->prev? To node containing _____

Q5. Starting from head, how can we access node containing 9?

_____.

Q6. Assume pointer p points to the node containing 8. How can we access the node containing 1 and change the x component to 0?

Q7. Assume that p points to the last node. From there, how can you access the node containing 8 and update its x component to 6?

Q8. Delete the node containing 19.

Q9. Delete the node containing 1.

Q10. Insert a new node with the x component equal 50. Assume that it is pointed to by pointer temp [temp->next and temp->prev are both equal to NULL]. Insert the new node in between node containing 19 and 89.

Now, I think you have enjoyed so much of tracing the linked list, it's time to go up and face the real challenge. The exercise for this week's meeting is relatively easy. [You should have mastered the singly linked list concepts, otherwise you would have a hard time following].