Doubly Linked List

1 DOUBLY LINKED LIST

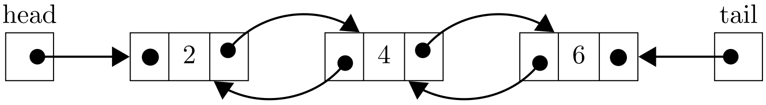
Doubly Linked List are linked lists that contain links to next and previous nodes. Unlike the singly linked lists with one way traversal, we can traverse the doubly linked lists using the previous pointer or the next pointer.

Base Structure Definition for a Doubly-Linked List:

```
typedef struct node_tag {
   int data;
   struct node_tag *prev; //Two pointers for the two traversal
   struct node_tag *next;
} NODE;
```

The contents of a doubly linked list can be easily traversed since there are two pointers. Data traversion can be done in two directions (forward and backward). Doubly linked lists also make insertion and deletion easier.

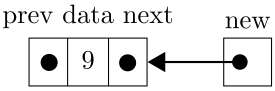
However, also keep in mind that since an additional member was added in the node, additional memory space will also be allocated for the node and there will be more pointer operations.



2 OPERATIONS ON A DOUBLY LINKED LIST

2.1 INSERT

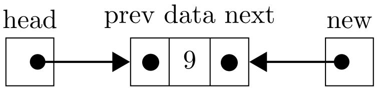
1. Create a new node using `malloc()` and initialize the members of the structure.



```
NODE * new = (NODE *) malloc (sizeof(NODE));
scanf("%d", &new->data);

//ensures that the node is not connected to anything yet
new->next = NULL;
new->prev = NULL;
```

2. Several Cases to Consider when Inserting a Node:

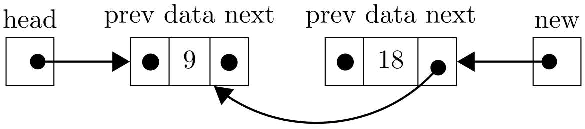
   (a) If the node will be **inserted in an empty list**, then use insert at head.



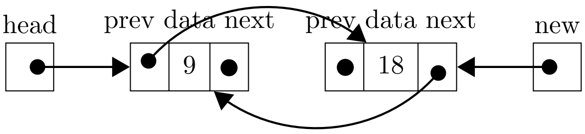
```
if (head == NULL) { //checks if list is empty
   head = new; //new's pointers are already NULL
}
```

   (b) If the node will be **inserted as the new head** (list not empty), then use a modified insert at head.

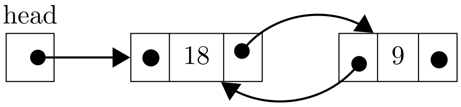
      i. Point the `next` pointer of `new` to `head`.



ii. Point the `prev` pointer of `head` to `new`.



iii. Point `head` to `new`.



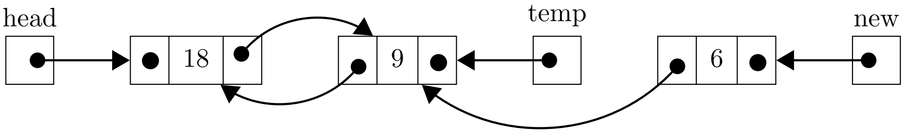
```
//In this example, node will be inserted as head if it is greater
    than the current head (descending order)
else if (new->data > head->data) {
   new->next = head;
   head->prev = new;
   head = new;
}
```

(c) If not inserted at the head, use a temporary pointer to locate the node that will precede the new node.

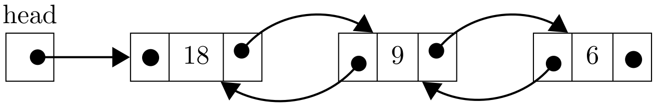
```
else {
   NODE *temp = head;

   /* the loop will only stop if:
      (1) temp is at the last node
      (2) new data is greater than the temp->next */
   while(temp->next != NULL && temp->next->data > new->data) {
      temp = temp->next;
   }
```

i. If insert is **<u>at the end</u>** of the list.

A. Point the `prev` pointer of `new` to node pointed by `temp`.



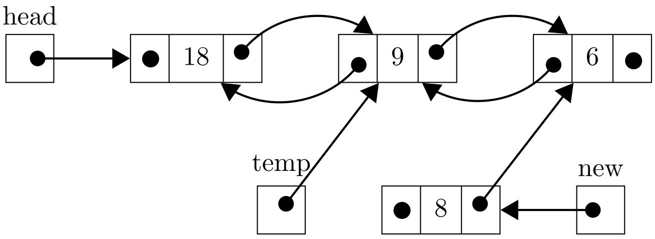
B. Point the `next` pointer of `temp` to `new`.



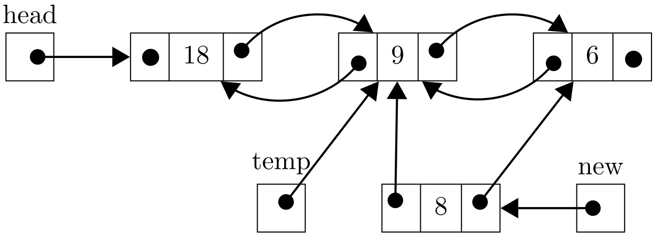
```
   //Put after the loop for finding the node before the new node
   if (temp->next == NULL) {
      new->prev = temp;
      temp->next = new;
   }
```

ii. If insert is **<u>at the middle</u>** of the list.

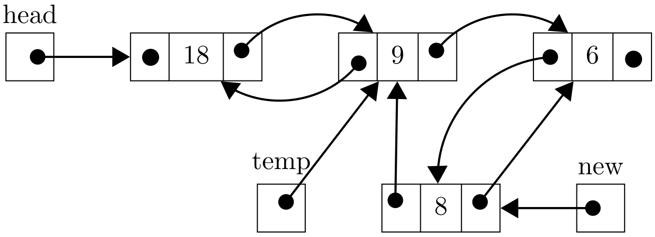
A. Point the `next` pointer of `new` to the node pointer by the `next` pointer of `temp`.



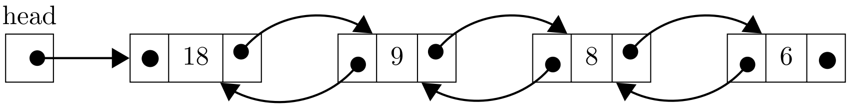
B. Point the `prev` pointer of `new` to the node pointed by `temp`.



C. Point the `prev` pointer of the node pointed by the `next` pointer of `temp` to new.



D. Point the `next` pointer of `temp` to `new`.



```
    //Put after the loop for finding the node before the new node
    else {
        new->next = temp->next;
        new->prev = temp;
        temp->next->prev = new;
        temp->next = new;
    }
}
```

2.2 DELETE

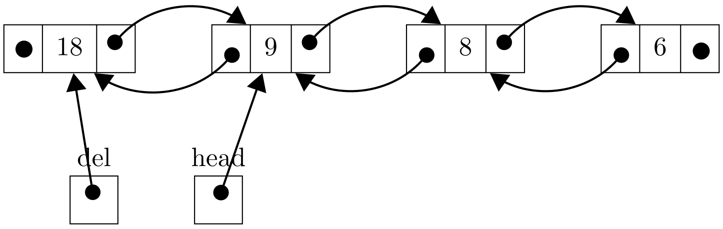
1. Locate the node to be deleted using a pointer. Pointer `del` will point to the node to be deleted.

```
int x;
printf("Enter data to be deleted: ");
scanf("%d", &x);

NODE *del = head;
while(del != NULL) { //Traverse the loop until the end
    if (del->data == x) { //If you find the same data, stop the loop
        break;
    }

    del = del->next;
}
```

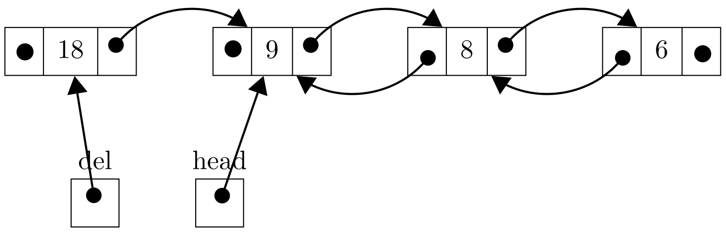
NOTE: If data to be deleted does not exist in the linked list, or if the list is empty, simply say so.

(a) If the node to be **deleted is the head**, then use delete at head.

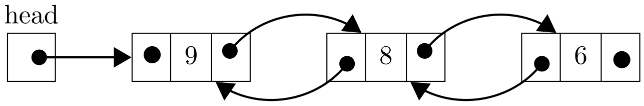
i. Point the `head` to the `next` (second) node.



ii. Make the previous pointer of head to `NULL`.



iii. Free pointer `del`.



```
if (del == head) {
    head = head->next; //head = del->next;
    head->prev = NULL;
    free(del);
}
```

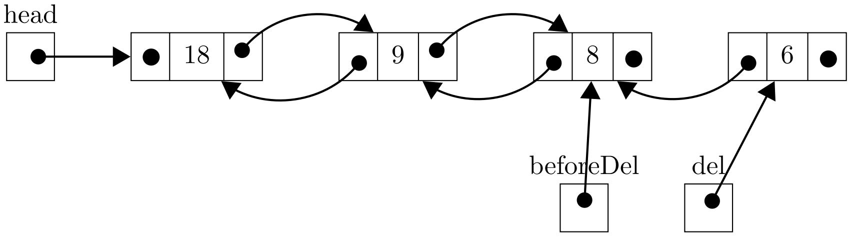
NOTE: `head->prev` will result in a segmentation fault if there is only one node left. Make sure to add how to correct that. Also, you have to make sure to reset the head to `NULL` if it becomes empty.

(b) Else, use another pointer (let's use `beforeDel`) to point to the node before the node to be deleted.

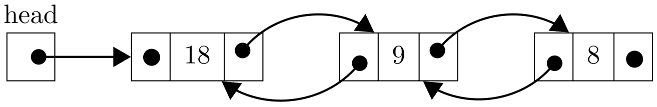
```
else {
    NODE * beforeDel = head;
    while(beforeDel->next != del) {
        beforeDel = beforeDel->next;
    }
```

i. If the node to be deleted is at the **end of the list**.

A. Make the `next` pointer of `beforeDel` to `NULL`.



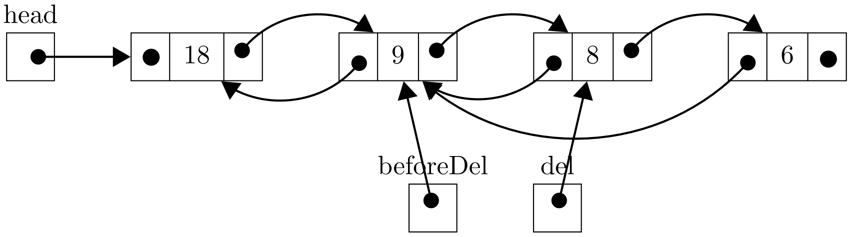
B. Free pointer `del`.



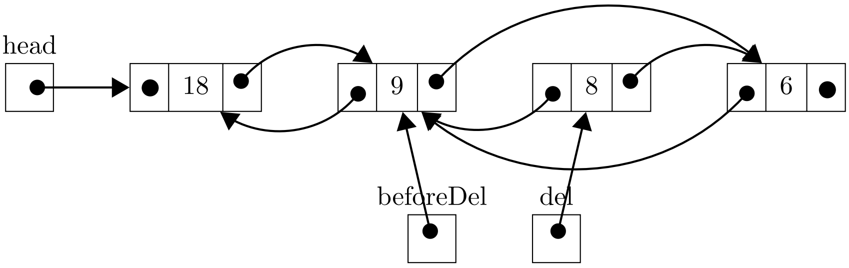
```
    if (beforeDel->next->next == NULL) {
        beforeDel->next = NULL;
        free(del);
    }
```

ii. If the node to be deleted is at the **middle of the list**.

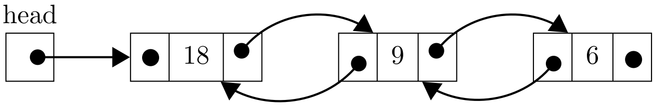
A. Point the `prev` pointer pointed by the `next` pointer of pointer `del` to `beforeDel`.



B. Point the `next` pointer of `beforeDel` to the `next` pointer of `del`.



C. Free pointer `del`.



```
    else {
        del->next->prev = beforeDel;
        beforeDel->next = del->next;
        free(del);
    }
}
```

2.3 PRINTING

2.3.1 USING THE NEXT POINTER

It is like printing in a singly linked list.

1. Using a temporary pointer of the same data type, point it to the first element of the linked list.

```
NODE *temp = head; //points temp to the first node
```

2. Until the pointer does not point to `NULL`, print the data of the current node and then move the pointer to the next node.

```
while(temp != NULL) { //prints forward
    printf("%d\t", temp->data);
    temp = temp->next;
}
```

2.3.2 USING THE PREV POINTER (REVERSE)

1. Using a temporary pointer of the same data type. Position the pointer to the last node.

```
NODE *temp = head;
while(temp->next != NULL) { //points temp to the last node
    temp = temp->next;
}
```

2. Until the pointer does not point to `NULL`, print the data of the current node and then move the pointer to the previous node.

```
while(temp != NULL) { //prints backward
    printf("%d\t", temp->data);
    temp = temp->prev;
}
```