Singly-Linked List 02

1 Operations at Middle

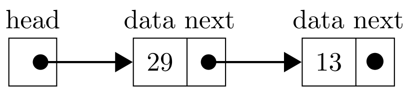
Previously, we have discussed how to insert/delete data at head and at tail of the list. Now what if we want to insert/delete in the middle of the list?

Base Code for the Discussions Below:

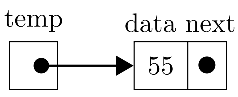
```c
#include <stdio.h>
#include <stdlib.h> //NOTE: Remember to add this for malloc

//Each node created for the linked list will follow this structure
typedef struct node_tag {
    int data;
    struct node_tag * next;
} NODE;

int main() {
    //Create a head pointer to maintain the linked list. Ensure that it is
        initialized to NULL to denote that it is empty.
    NODE *head = NULL;
}
```

1.1 Insert at Middle

A linked list created using definition above is illustrated below:

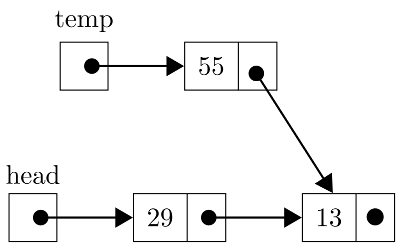


Let's try to insert the number 55 between 29 and 13. First, we need to create the node using a temporary pointer, let's say `temp`.

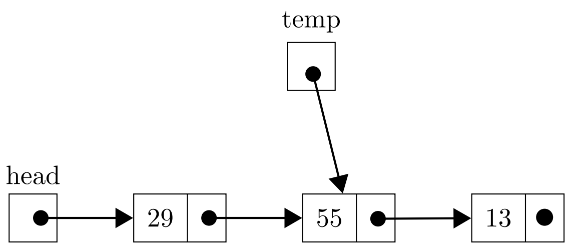


To insert the new node, we need to do the following:

1. Point the `next` pointer of 55 to the node after 29:



2. Point the `next` pointer of 29 to the new node (being pointed by `temp`):

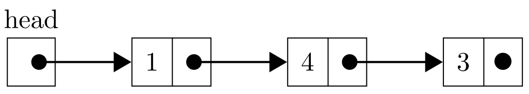


The new node (55) is successfully inserted to the list.

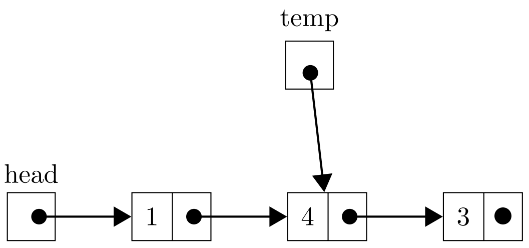
Can the same steps be used when the linked list is empty? How about when there is only one node on the list? How about larger lists?

1.2 Delete at Middle

Consider the linked list below:

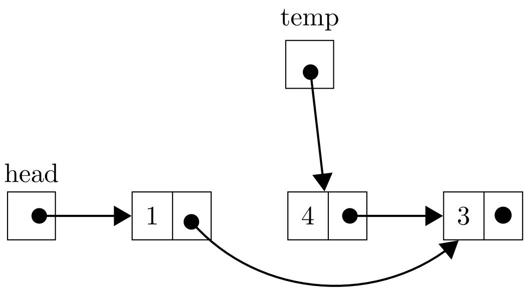


Let us try to delete the node containing 4. First, we need a temporary pointer (`temp`) to point at the node we want to delete:

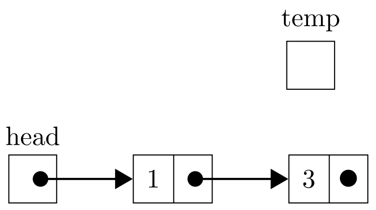


Then, we need to do the following:

1. Point the `next` pointer of 1 (node before the one to be deleted) to 3 (node after the one to be deleted):



2. Delete `temp` (using `free()`):



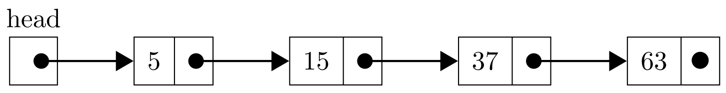
`temp` didn't vanish, it just doesn't point to anything anymore

The node (4) is successfully deleted from the last.

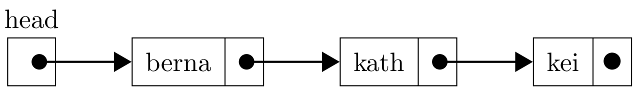
Can the same steps be used when the linked list is empty? How about when there is only one node on the list? How about larger lists?

2 Sorted Operations

Most of the time, nodes in a linked list must be arranged in a certain order. For example the nodes can be arranged in ascending order:



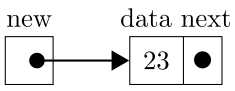
Or in alphabetical order:



In order to achieve this, we can insert the data at the head/tail of the list and then sort it. However, this process can be tedious and requires a lot of operations since we need to sort the list after every insertion. The more logical way to do this is by inserting each node in its **proper place** every time we try to add a node.

2.1 Sorted Insertion

1. Create a new node and initialize it's members.



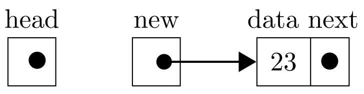
```
NODE * new = (NODE *) malloc (sizeof(NODE));

printf("Enter data: ");
scanf("%d", &new->data); //23 will be the sample input value

new->next = NULL;
```

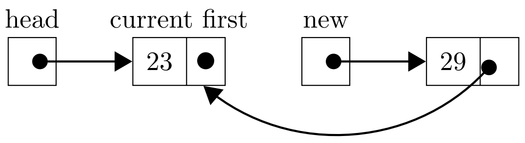
2. Several Cases to Consider when Inserting a Node:

   (a) If the new node will be inserted as the new `head`, then use insert at `head`.

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

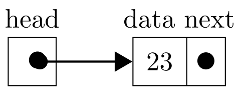
INSERT AT EMPTY: **INSERT AT HEAD:**

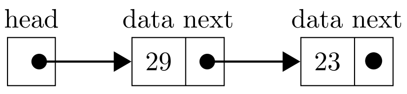




      ii. Point `head` to `new`.

INSERT AT EMPTY: **INSERT AT HEAD:**



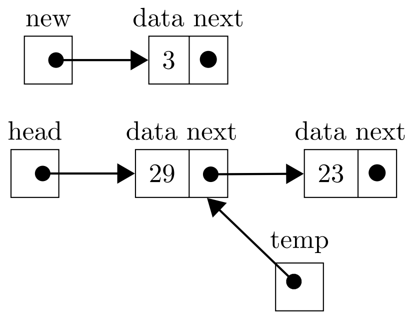


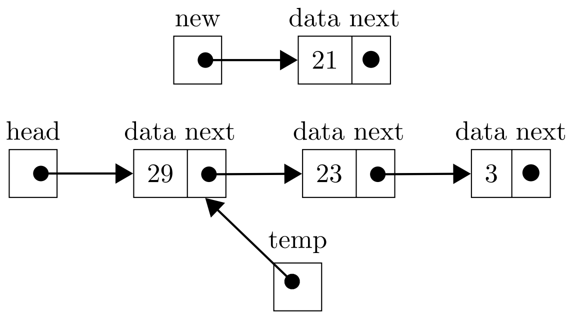
```
//In this example, sorting is done in descending order

//Insertion at head will occur if:
  //the list is empty
  //if the new data is greater than the current head (meaning list
      already has a value)
if (head == NULL || (head != NULL && head->data < new->data)) {
    new->next = head;
    head = new;
}
```

NOTE: This is only a base version, you can use other ways that is easier for you to understand.

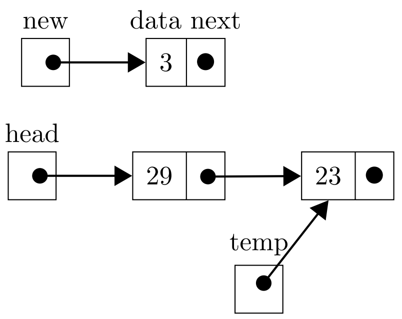
(b) Else, insert at middle or tail. You need to use a loop to locate the node that will immediately precede the new node.

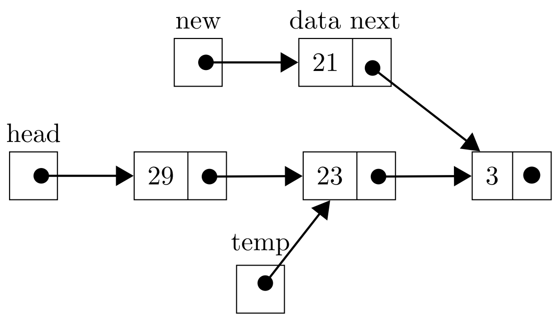
INSERT AT TAIL: **INSERT AT MIDDLE:**



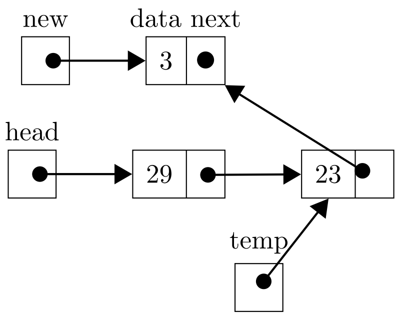


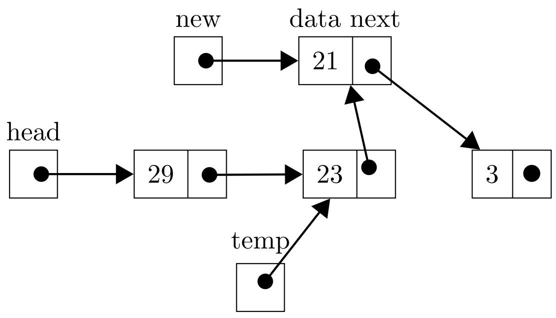
i. Point the `next` pointer of `new` to where the `next` pointer of `temp` is pointing.

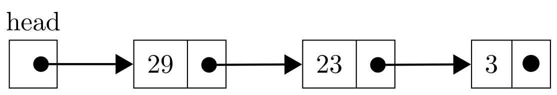


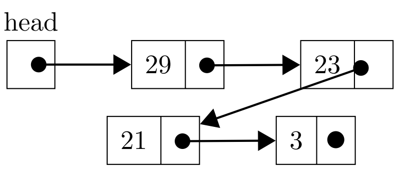


ii. Point the `next` pointer of `temp` to `new`.









```
else {
    //Make temp point to the first node
    NODE *temp = head;

    //Traverse the list until you reach the last node or until the value of
       new data is less than the next node
    while(temp->next != NULL && temp->next->data > new->data) {
        temp = temp->next;
    }

    new->next = temp->next;
    temp->next = new;
}
```

NOTE: This is only a base version, you can use other ways that is easier for you to understand.

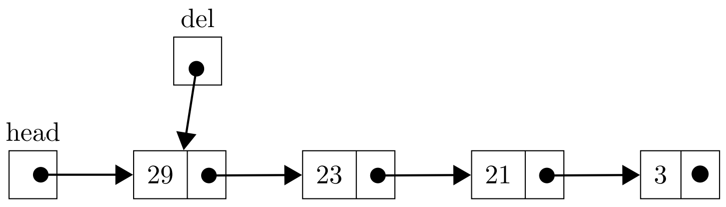
2.2 Deletion

1. Using a temporary pointer (`del`), locate the node that will be deleted.

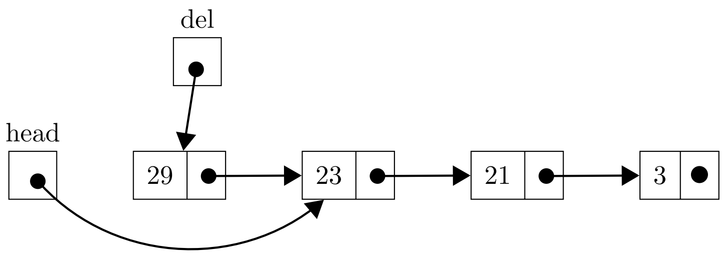
```
int x;
printf("Enter data to be deleted: ");
scanf("%d", &x);

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.

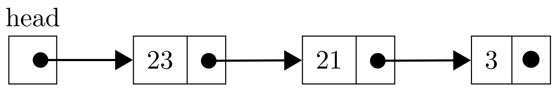
(a) If the node to be deleted is at the head, use delete at head:



i. Point `head` to where the `next` pointer of `del` is pointing.

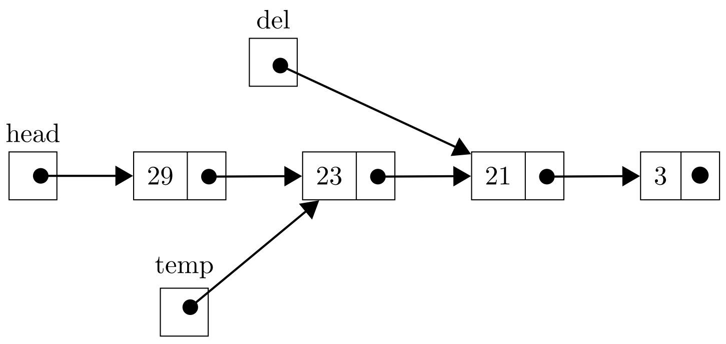


ii. Free or delete `del`.

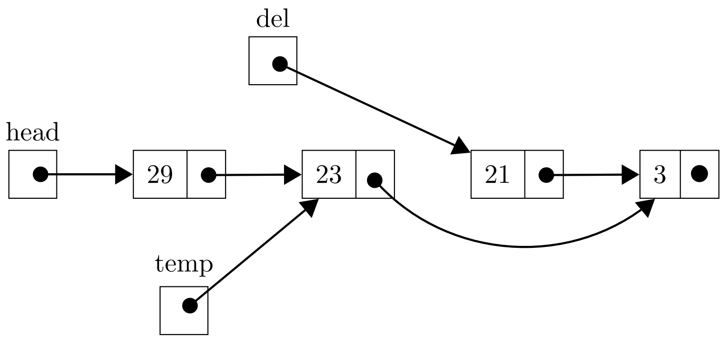


```
if (del == head) {
    head = del->next; //OR: head = head -> next;
    free(del);
}
```

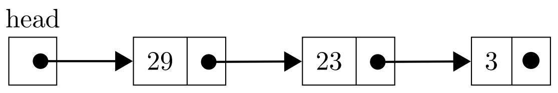
(b) Else, use another pointer (`temp`) to locate the node that will immediately precede the node to be deleted (held by `del`).



i. Point the `next` pointer of `temp` to where the `next` pointer of `del` is pointing.



ii. Free or delete `del`.



```
else {
    NODE * temp = head;
    //Traverse the loop to find the node before the node to be deleted
    while(temp -> next != del) {
        temp = temp->next;
    }

    temp->next = del->next;
    free(del);
}
```