Singly-Linked List 01

1 SELF-REFERENTIAL STRUCTURES

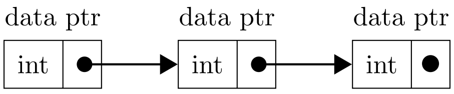
Self-referential structures are structures that has a pointer to a structure similar to itself as member. Below are examples of self-referential structures:

```c
struct node_tag {
  int data;
  struct node_tag * ptr;
};
```

```c
typedef struct node_tag {
  int data;
  struct node_tag * friend_one;
  struct node_tag * friend_two;
} NODE;
```

Self-referential structures are used to create chains of structures that are of the same type. The pointer member acts as a link from one structure instance to another structure instance.

To make the connections easier to visualize, you should draw it. Each structure variable (NODE) would be a box that has several divisions depending on the number of members, and each pointer variable would be a box. A dot inside a box area would be a NULL value, and you can use arrows to connect the pointers to other nodes.

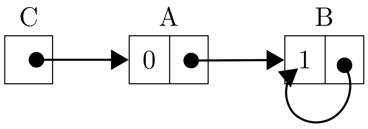


TRY THIS!

```c
1  #include<stdio.h>
2
3  typedef struct node_tag { //structure definition
4    int x;
5    struct node_tag *ptr;  //pointer to another struct node_tag
6  }NODE;
7
8  int main(){
9      NODE A, B, *C;
10     A.x = 0, B.x = 1;
11     A.ptr = &B, B.ptr = &B;
12     C = &A;
13
14     // what will be the output?
15     printf("\nA: %d\tB: %d\tC: %d\n", A.x, B.x, C->x);
16
17     //this is where the magic starts
18     (*(B.ptr->ptr)).ptr->x += (*C).ptr->x;
19     (*(A.ptr->ptr)).ptr = C;
20     C->ptr->x = (*(A.ptr->ptr)).ptr->x * C->ptr->ptr->x;
21
22     // what will be the output?
23     printf("\nA: %d\tB: %d\tC: %d\n", A.x, B.x, C->x);
24 }
```

RECALL: `<pointer_name> -> <member_name>` is equivalent to `(*<pointer_name>).<member_name>`

The diagram below represents the variables declared and initialized above (Line 9 to 12):



For Line 15, the output would be `"A: 0  B: 1  C: 0"`. C is 0 since it just points to structure variable A.

In Line 18, we get `(*(B.ptr->ptr)).ptr->x += (*C).ptr->x;`. Now, how does this work?

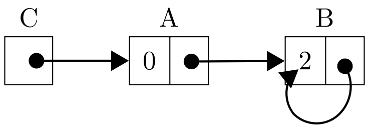
On the left-hand side of the operator `(+=)`, we get `(*(B.ptr->ptr)).ptr->x`:

- We can convert the `(*(B.ptr->ptr)).ptr->x.` to `B.ptr->ptr->ptr->x` to make it easier to understand.
- `B.ptr` holds the address of B. We can replace it with `'&B'` to make the code `'&B'->ptr->ptr->x`.
- Just trace it again for the next parts. Replacing `'&B'->ptr` will lead to `'&B'->ptr->x` and then `'&B'->x.`
- Now, we can see that the one we need to replace is the x member of the structure variable B.

On the right-hand side, we get `(*C).ptr->x`:

- Again, we can convert the `(*C).ptr->x` to `C->ptr->x` to make it easier to understand.
- `C` pointer just holds the address of A so we can replace it with that. Now, we have `'&A'->ptr->x`.
- `'&A'->ptr` just holds the address of B. Now, we have `'&B'->x`.
- Based on the initialization, B's x member is equal to 1.

Given this tracing, the code will become: `'&B'->x += 1`, meaning B's x member will now become 2.



In Line 19, we get `(*(A.ptr->ptr)).ptr = C;`. Now, how does this work?

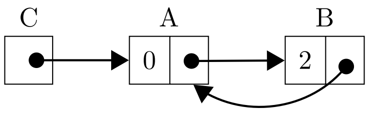
On the left-hand side of the operator `(=)`, we get `(*(A.ptr->ptr)).ptr`:

- We can convert the `(*(A.ptr->ptr)).ptr.` to `A.ptr->ptr->.ptr` to make it easier to understand.
- `A.ptr` holds the address of B. We can replace it with `'&B'` to make the code `'&B'->ptr->.ptr`.
- Just trace it again for the next parts. Replacing `'&B'->ptr` will lead to `'&B'->ptr`.
- Now, we can see that the one we need to replace is the ptr member of the structure variable B.

On the right-hand side, we get `C`:

- Based on the initialization, C is equal to the address of A.

Given this tracing, the code will become: `'&B'->ptr = &A`, meaning B's ptr member will now point to A.



In Line 20, we get `C->ptr->x = (*(A.ptr->ptr)).ptr->x * C->ptr->ptr->x;`. Now, how does this work?

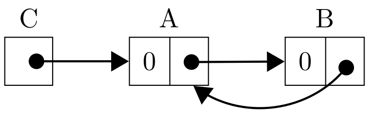
On the left-hand side of the operator `(=)`, we get `C->ptr->x`:

- `C` pointer just holds the address of A so we can replace it with that. Now, we have `'&A'->ptr->x`.
- `'&A'->ptr` just holds the address of B. Now, we have `'&B'->x`.
- Now, we can see that the one we need to replace is the x member of the structure variable B.

On the right-hand side, we get `(*(A.ptr->ptr)).ptr->x * C->ptr->ptr->x`:

- Convert the `(*(A.ptr->ptr)).ptr->x` to `A.ptr->ptr->ptr->x` to make it easier to understand.
- By tracing, we get `'&B'->ptr->ptr->x` followed by `'&A'->ptr->x`. The final form will be `'&B'->x`.
- By tracing, we get `C->ptr->ptr->x` followed by `'&A'->ptr->ptr->x`. Then, `'&B'->ptr->x`. The final form will be `'&A'->x`.
- B's x member is 2, and A's x member is 0.

Given this tracing, the code will become: `'&B'->x = 2 * 0`, meaning B's x member will now become 0.



For Line 23, the output would be `"A: 0  B: 0  C: 0"`. C is 0 since it just points to structure variable A.

2 LINKED LISTS

A linked list consists of several structures linked together, forming a chain of structures. It allows you to create **flexible data structures that can expand or shrink dynamically** to accommodate varying data sizes. Additionally, it serves as an **alternative to arrays**, which some prefer because it is more efficient in the memory and easier to manipulate.

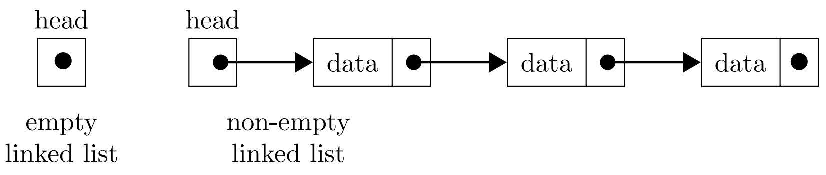
A linked list is usually composed of:

1. **structure variables** that contain the data, creating the chain of information, and
2. a **'HEAD' structure pointer** which points to the first element of the list.

To expand or shrink linked lists, you have to remember how to use the functions **malloc() and free()**.

2.1 Head Pointer

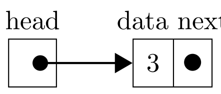
To start creating a linked list, you need to have a pointer to a structure of the desired data type. We call this the **head pointer**. The head pointer points to the **first element** of the list if not empty, and must contain NULL if the list is empty. This is essential because it is used to maintain and access the linked list.



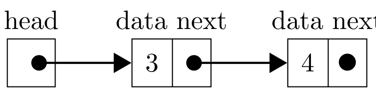
TRY THIS!

```c
#include<stdio.h>
#include<stdlib.h> //library that contains malloc() and free()

typedef struct node_tag { //structure definition
  int data;
  struct node_tag * next; //pointer to the next struct node_tag
} NODE;

int main() {
    NODE *head, *p;

    //what happened here?
    head = (NODE *) malloc(sizeof(NODE));
    head->data = 3;
    head->next = NULL;

    //how about here?
    head->next = (NODE *) malloc(sizeof(NODE));
    head->next->data = 4;
    head->next->next = NULL;

    //what happened here?
    p = (NODE *) malloc(sizeof(NODE));
    p->data = 5;
    p->next = NULL;
    head->next->next = p;

    //how about here?
    p->next = (NODE *) malloc(sizeof(NODE));
    p = p->next;
    p->next = NULL;
    p->data = 6;
}
```

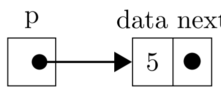
In Lines 13-15, we get to see a new node being created. The address of the space created by malloc is connected to the head pointer.



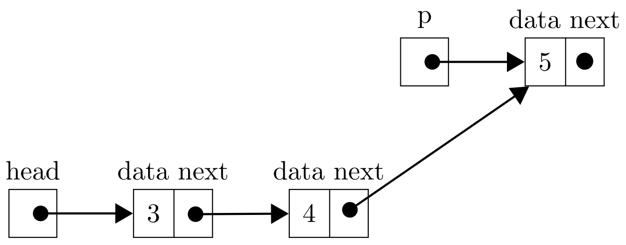
In Lines 18-20, a second node is created. This time, the address of the space created by malloc is connected to the next pointer of the head, meaning it will be put after 3.



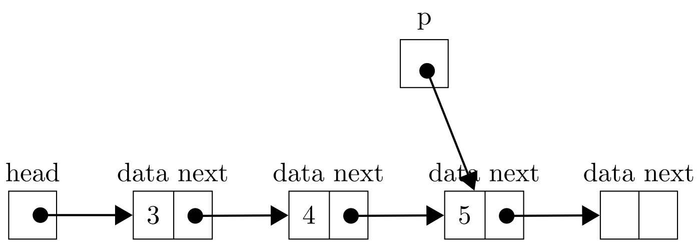
In Lines 23-25, another node is create, but this time the address of the space created by malloc is connected to another pointer `p`.



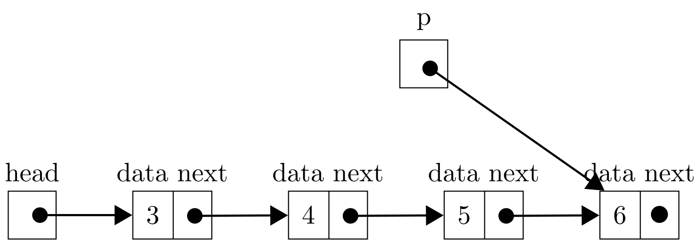
In Line 26, the next pointer of list's second node will also point to the new node pointed to by p.
NOTE: This doesn't mean that p will lose it's connection to the new node.



In Line 29, a new node is created. This time, the address of the space created by malloc is connected to the next pointer of the node held by pointer p, meaning it will be put after 5.



In Line 30-32, the `p` pointer is now moved to the last node, and it is used change the data there. In this way, `p` pointer serves as a temporary pointer that holds the last current node to connect new nodes easier since it is inefficient to use `head->next->next->next`.



3 BASIC OPERATIONS ON A LINKED LIST

NOTE: Linked List is easier to trace if you draw the nodes and their connections. So, try to have a pen and paper beside you when studying this part.

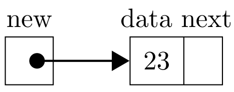
```c
#include <stdio.h>
#include <stdlib.h> //NOTE: Remember to add this for malloc and free.

//Each node created for the linked list below 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;
}
```

3.1 Adding Data

3.1.1 Adding data at the beginning (insert at head)

1. Using a pointer of the same data type, dynamically allocate a new structure variable using `malloc()` and initialize the members of the structure.

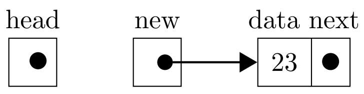
```c
//Use malloc to create an instance of the node.
NODE * new = (NODE *) malloc (sizeof(NODE));

//Initialize the members of the structure:
new->data = 23; //You can assign a value directly or use scanf()

//NOTE: We used (->) operator because 'new' is a pointer.
```



2. Point the `next` pointer of the newly allocated structure to the new first element of the list.

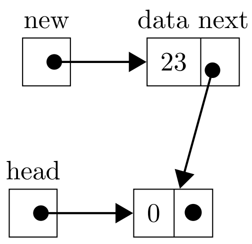
```c
//The new first element of the list is held by the head pointer.
//Since 'new' will become the new head value, you have to make sure that
   the original head will be connected to it.
new->next = head;
```

IF HEAD IS EMPTY:



`new->next` will become `NULL`.

IF HEAD IS NOT EMPTY:

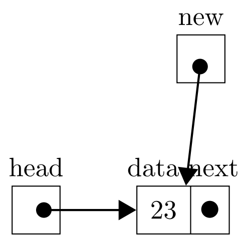


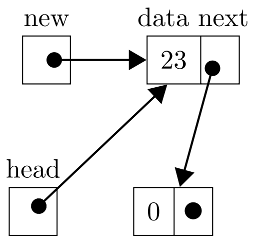
`new->next` will point to the node currently held by the head pointer.

3. Point the **head** pointer to the newly allocated structure.

```
//new will become the new first element
head = new;
```

IF HEAD IS EMPTY: **IF HEAD IS NOT EMPTY:**

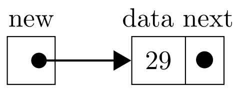




3.1.2 Adding data at the end (insert at tail)

1. Create a new node using `malloc()` and make its `next` pointer point to `NULL`.

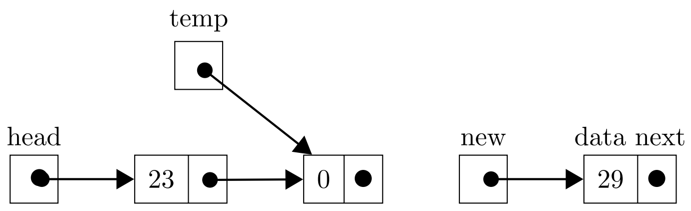
```
NODE *new = (NODE *) malloc (sizeof(NODE));

new->data = 29;

//To indicate that nothing follows (since last node), put NULL.
new->next = NULL;
```



2. Traverse the list using a pointer to find the current last node of the list (the node with `next` pointer equal to `NULL`).

```
//You need a loop to traverse the list.
//You need to create a temporary pointer because you cannot use the head
   for traversing the nodes (if you do that, you will lose track of the
   first element and will not be able to maintain the linked list).

NODE *temp = head;
while(temp->next != NULL) {
    temp=temp->next;
}

//'temp->next != NULL' is used as the condition to search for the
   current last node since the current last node's next value is NULL
//You can't use 'temp != NULL' as the condition, you will not be at the
   last node, but the place after the last node (which is NULL).
```

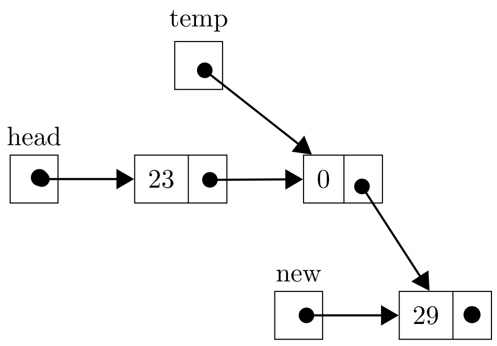
NOTE: This part will cause a segmentation fault if the head is `NULL` because if it is `NULL` it cannot have a next. If the head is still empty, just use add at head.



`temp` will stop at node 0 since `temp->next != NULL` will become `FALSE` there.

3. Make the `next` pointer of the current last node point to the newly allocated node.

```
temp->next = new; //temp holds the original last node
```

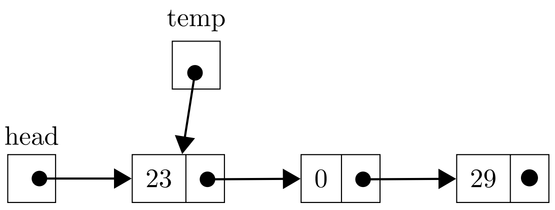


3.2 Deleting Data

3.2.1 Deleting data at the beginning (delete at head)

1. Use a temporary pointer of the same data type to point at the current first element of the list.

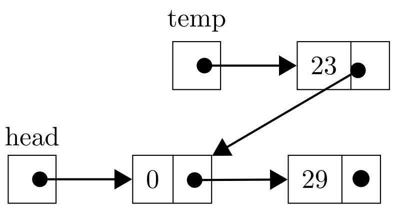
```
NODE *temp = head;
```



2. Move `head` pointer to point to the current second data of the list.

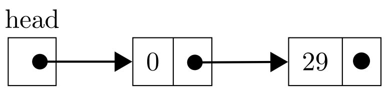
```
head = head->next; //OR: head = temp->next;
```

NOTE: This part will cause a segmentation fault if the head is NULL because if it is NULL it cannot have a next. If the head is still empty, just state that there is nothing to delete.



3. Use the `free()` function to delete the node pointed by the temporary pointer.

```
free(temp);
```

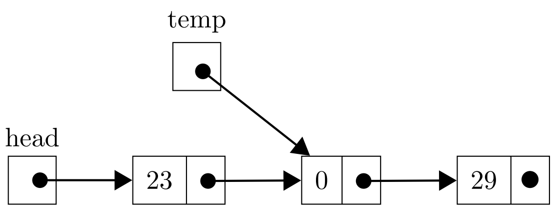


3.2.2 Deleting data at the end (delete at tail)

1. Traverse the list using a temporary pointer to find the second to the last node of the list.

```
//Since 'temp->next != NULL' stops at the last node you can use
    'temp->next->next != NULL' to get the second to the last.

NODE *temp = head;
while(temp->next->next != NULL) {
    temp = temp->next;
}
```

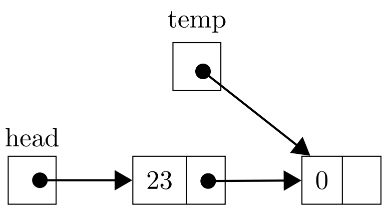
   NOTE: Make sure that the list contains more than one nodes. If it doesn't, the code above will result in a segmentation fault. If the head is still empty, just state that there is nothing to delete, and if it only has one node, simply delete at head.



   `temp` will stop at node 0 since `temp->next->next != NULL` will become `FALSE` there.

2. Deallocate the node at the end of the list using `free()`.

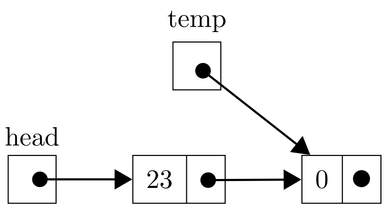
```
free(temp->next); //temp->next holds the last node
```



3. Update the `next` pointer of the new last element to `NULL`.

```
temp->next = NULL; //Ensures that the node temp holds becomes the new
    last node
```

   NOTE: If you forget to put this, your loops for searching nodes may cause an infinite loop because it will never encounter NULL.



3.2.3 Delete All Nodes

1. Repeatedly execute *`Delete At Head`* until `head` points to `NULL`.

```
while(head != NULL) {
    NODE *temp = head;
    head = head->next;
    free(temp);
}
```

   NOTE: This part is important because remember that if you have a dynamic data, you have to free them before exiting the program to ensure that you will not use all of the memory of your system.

3.3 Printing the Contents of a List

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

```
//Create a temp pointer for traversing the list. You can't use 'head'
   directly because you won't be able to keep track of the first element.
NODE *temp = head;
```

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

```
//Use a loop and traverse the linked list until you encounter a NULL
   (which signifies that you are at the end)
while(temp != NULL) {
    printf("%d\t", temp->data); //Prints the data
    temp = temp->next;          //Updates the loop
}

//NOTE: You can also use a for loop.
for(NODE *temp = head; temp != NULL; temp=temp->next) {
    printf("%d\t", temp->data);
}
```

4 PASSING LINKED LISTS AS PARAMETERS TO FUNCTIONS

You just need to remember that if you have to modify the linked list, you have to use pass by reference. This usually applies to adding or deleting nodes.

Sample Code for Pass by Reference:

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

void deleteAllNodes(NODE **head) {
  //code for deleting all nodes
}

int main() {
  NODE *head = NULL;
  //insert some data on the linked list

  deleteAllNodes(&head);
}
```

On the other hand, if you simply want to search for a value, or print some data, edit some data in the node, or anything that doesn't need to modify the linked list, you can just use pass by value.

Sample Code for Pass by Value:

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

void printAllData(NODE *head) {
  //code for printing nodes
}

int main() {
  NODE *head = NULL;
  //insert some data on the linked list

  printingAllData(head);
}
```