Linked Lists

Arrays demand a contiguous memory location. Lengthening of an array isn't possible. We would have to copy the whole array to another memory location to lengthen it's size. Similarly inserting or deleting a element causes the elements to shift right or left.

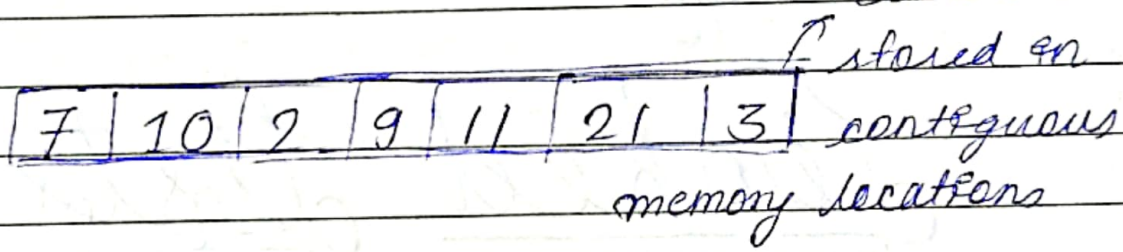
But linked lists are stored in non-contiguous memory locations. To add a new element, we just have to create a node somewhere in the memory & get it pointed by the previous element. And deleting also as easy as that. We just have

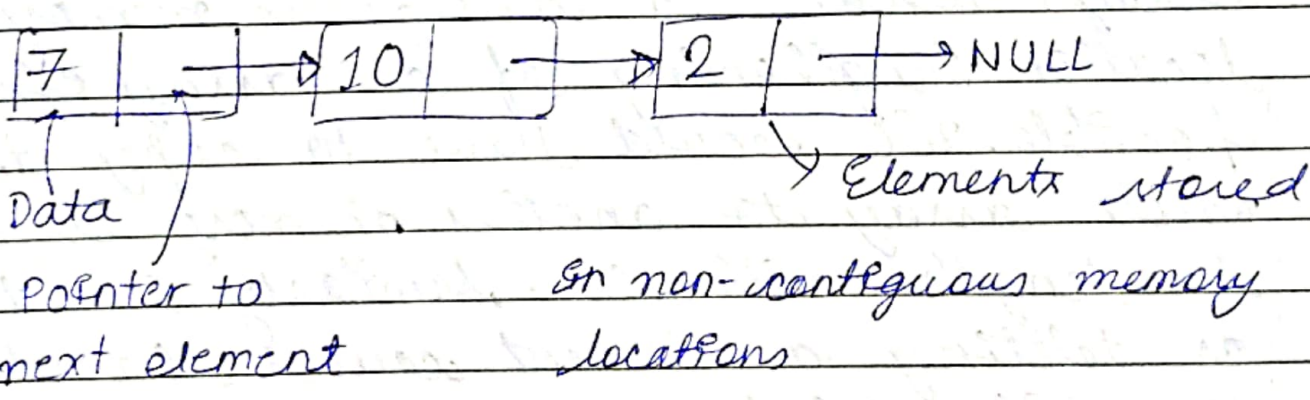
to skip pointing to a particular node.

Structure of a Linked List

Every element in a linked list is called a node & consists of 2 parts, the data part & the pointer part. The data part stores the values, while the pointer part stores the pointer pointing to the address of the next node. Both of these structures (arrays & linked lists) are linear data structures.

Linked Lists Vs Arrays





~~Why Linked Lists~~

Why Linked Lists

Memory & capacity of an array remains fixed, while in linked lists we can

keep on adding & removing elements without any capacity constraint.

Drawbacks of Linked Lists

→ Extra memory space for pointers is required (for every node, extra space for a pointer is needed)
→ Random access is not allowed as elements are not stored in contiguous memory locations.

Implementation

Linked lists can be implemented using a structure in C language.

```
struct Node {                    // → self
    int data;                    // referencing
    struct Node *next;           //   structure
};
```

→ We constructed a structure named node.
→ Define 2 of it's members, an integer data which holds the node's data, & a structure pointer, next, which points to the address of next structure node.

Code for creation & Traversal of a Linked List

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

struct Node
{
  int data;
  struct Node *next;
};

void LinkedListTraversal(struct Node *ptr)
{
  while (ptr != NULL)
  {
    printf("Element: %d\n", ptr->data);
    ptr = ptr->next;
  }
}

int main()
{
  struct Node *head;
  struct Node *second;
  struct Node *third;
  struct Node *fourth;

// Allocate memory for nodes in the
// Linked List in Heap
```

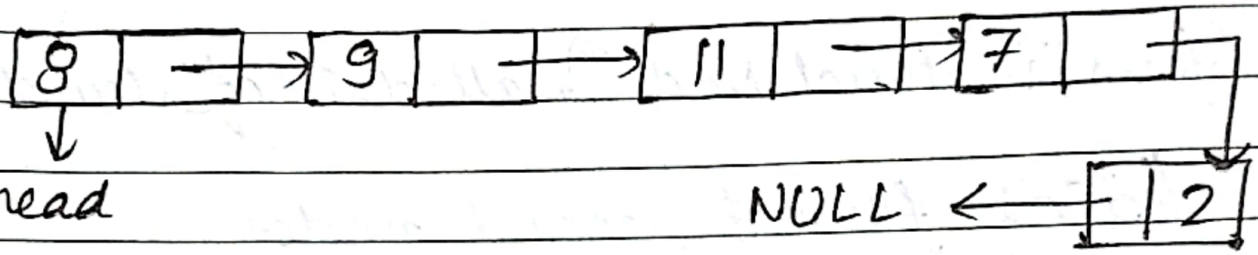
```
head=(struct Node*)malloc(sizeof(struct
                                Node));
second=(struct Node*)malloc(sizeof(struct Node));
third=(struct Node*)malloc(sizeof(struct Node));
fourth=(struct Node*)malloc(sizeof(struct Node));

//Link first & second nodes
head->data = 7;
head->next = second;

//Link second & third nodes
second->data = 11;
second->next = third;

// Link 3rd & 4th nodes
third->data= 41;
third->next = NULL; -> fourth
    fourth->data = 66
linkedlist fourth->next = NULL;
linkedListTraversal(head);
return 0;
}
```

o/p

Element : 7
" : 11
" : 41
" : 66

Insertion of a Node in Linked List

Consider the following Linked List



Insertion can be divided into the following categories

1) Insert at the beginning
2) Insert in between
3) Insert at the end
4) Insert after the node

Firstly, we would need to create a extra node. And then, we overwrite the current connection & make new connections

Syntax for creating a Node

```
struct Node* ptr = (struct Node*) malloc
                   (sizeof(struct Node));
```

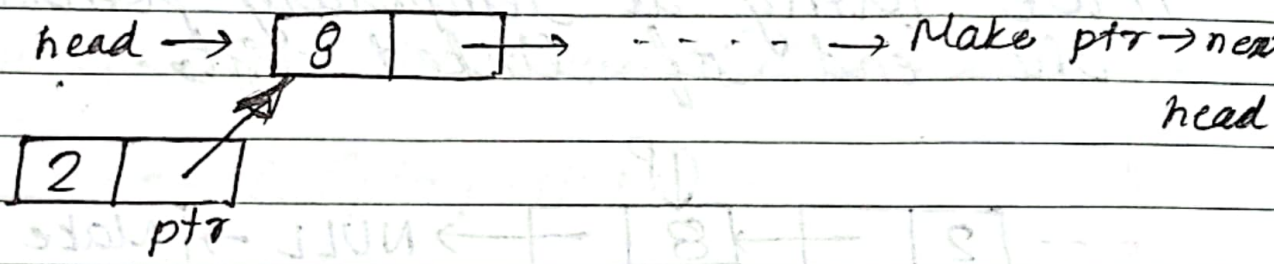
:

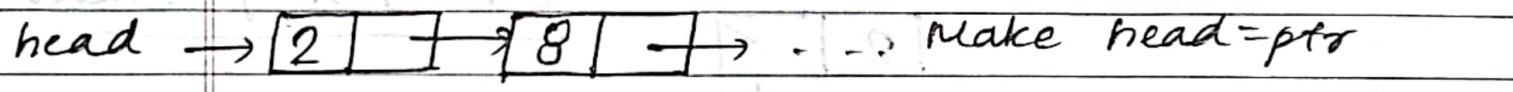
```
ptr -> data = 9
```

This will set data.

Case 1: Insert at the beginning (TC → O(1))

In order to insert at the beginning, firstly we need to have the head pointer pointing to the new node & the new node's pointer to the current head.

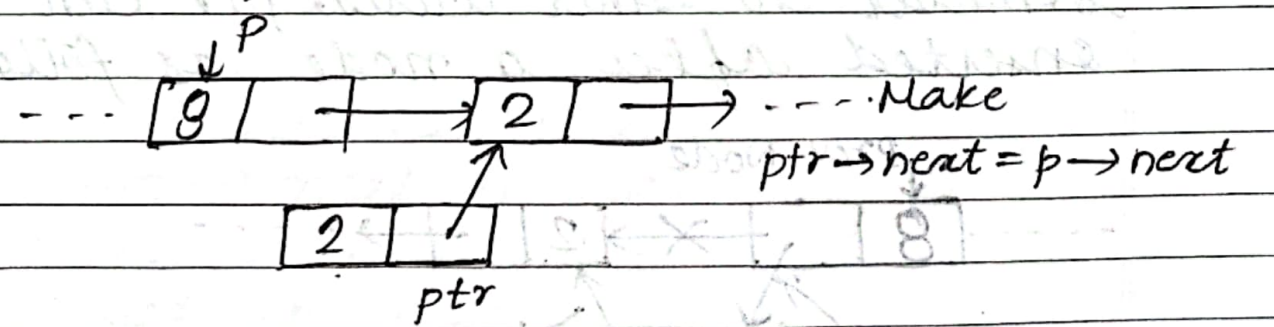


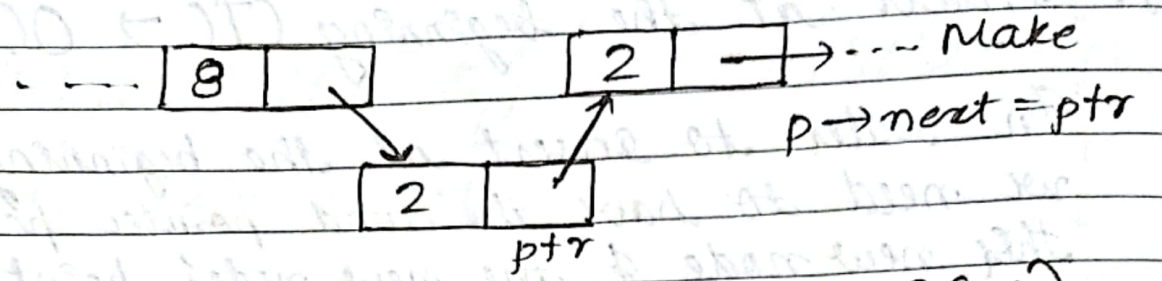


Case 2: Insert in between (TC → O(n))

~~Assuming~~ 9!

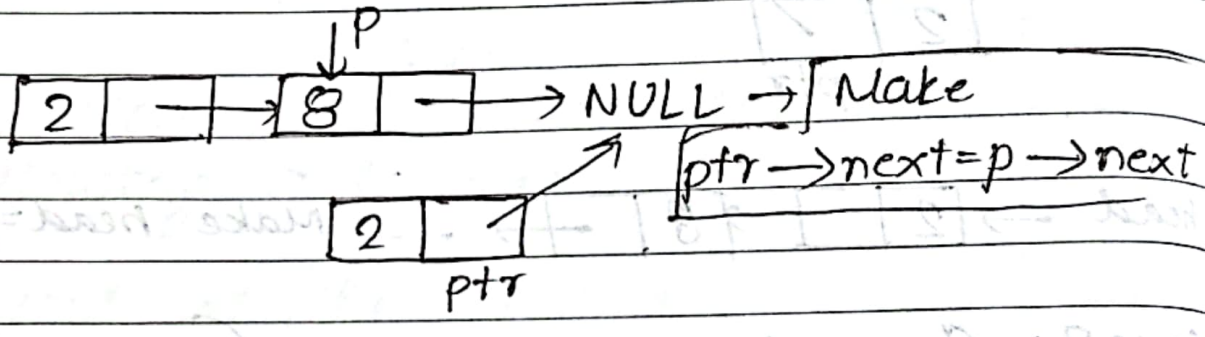
1) Bring a temporary pointer p pointing to the node before the element you want to insert in the linked list
2) Since, we want to insert between 8 & 2, we bring p to 8

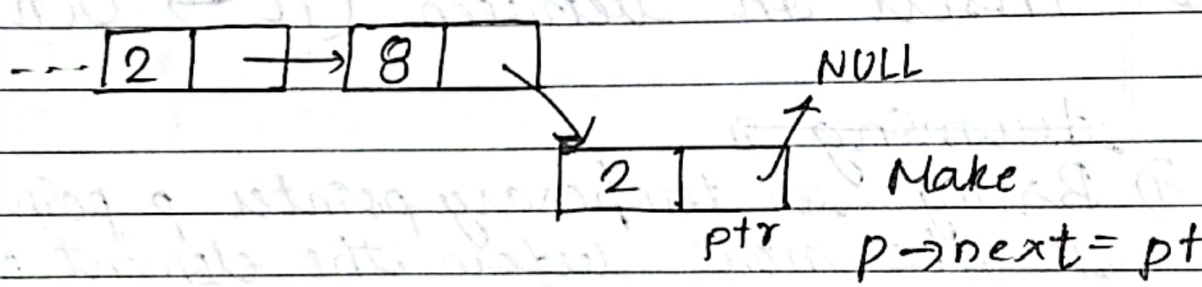




Case 3: Insert at the end (TC → O(n))

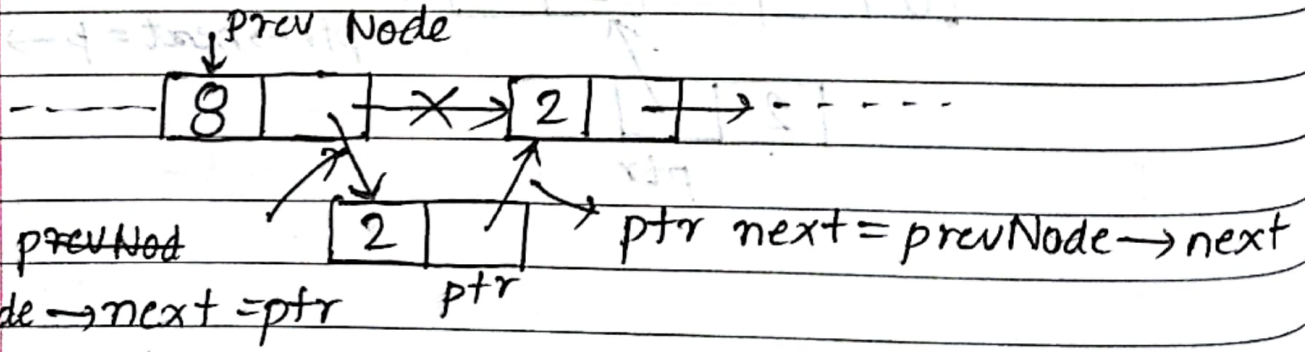
1) We bring a temporary pointer to the end of linked list.





Case 4: Insert after a node (TC → O(1))

Similar to other cases, ptr can be inserted after a node as follows.



Code

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

struct Node {
    int data;
    struct Node *next;
};

void LinkedListTraversal(struct Node* ptr)
{
    while (ptr != NULL)
    {
        printf("Element : %d \n", ptr->data);
        ptr = ptr->next;
    }
}

// Case 1

struct Node * InsertAtFirst(struct Node *head,
                            int data)
{
struct Node *ptr = (struct Node *)malloc
                   (sizeof(struct Node));

    ptr->data = data;
    ptr->next = head;
    head = ptr;
    return ptr;

}
```

// Case 2

```
struct Node * insertAtIndex(struct Node* he
                int data, int index)
    {
struct Node *ptr = (struct Node*) malloc
                (sizeof(struct Node));
struct Node *p = head;
 int i=0;

 while (i!=index-1)
   {
     p = p -> next;
       i++;
   }
    ptr -> data = data;
   ptr -> next = p -> next;
   p->next = ptr;

   return head;
     }
```

// Case 3

```
struct Node * insertAtEnd(struct Node*hea
                int data)
  {
struct Node *ptr = (struct Node*) malloc
              (sizeof (struct Node));
 ptr -> data = data;
 struct Node * p = head;
```

We use p→next & not p because we want to reach upto the last node & not NULL

```
while (p->next != NULL)
    {
      p = p->next;
    }
  p->next = ptr;
  ptr->next = NULL;
  return head;
}
```

// case 4

```
struct Node * insertAfterNode(
  struct Node *head, struct Node* prevNode,
                                int data)
        {
struct Node *ptr = (struct Node*) malloc
          (sizeof (struct Node));

  ptr -> data = data;
  ptr -> next = prevNode -> next;
  prevNode -> next = ptr;

  return head;

  }

int main () {

struct Node *head;
- // ---------- * second;
-// ---------- * third;
- // ---------- * fourth;
```

// Allocate memory for Nodes in the Linked List on Heap

```
head =
second =
third =
head = (struct Node*) malloc (sizeof (struct Node));
second = - // ---------- //
third = - // ---------- //
fourth = - // ---------- //
```

// Link first & second node

```
head → data = 7;
head → next = second;
```

// Link second & third nodes

```
second → data = 11;
second → next = third;
```

// Link third & fourth nodes.

```
third → data = 41;
third → next = fourth;
```

// Terminate the list at the 4th Node

```
fourth → data = 66;
fourth → next = NULL;
```

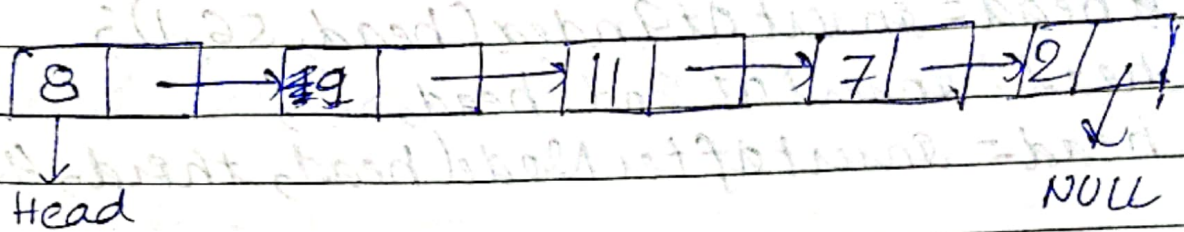
```
printf("Linked list before insertion:\n");
LinkedListTraversal(head);
head= insertAtFirst(head, 56);
head=insertAtIndex(head, 56, 1);
head=insertAtEnd(head, 56);
head= insertAfterNode(head, third, 45);
printf("\nLinked list after insertion\n");
LinkedListTraversal(head);
return 0;
}
```

O/P

```
Linked list before insertion
Element: 7
 11    : 11
 11    : 41
 11    : 66

Linked list after insertion
Element : 7
Element : 11
 11    : 41
 11    : 45
 11    : 66
```

Deletion in a Linked List

Consider the foll. LL:



Deletion can be done for foll. cases:

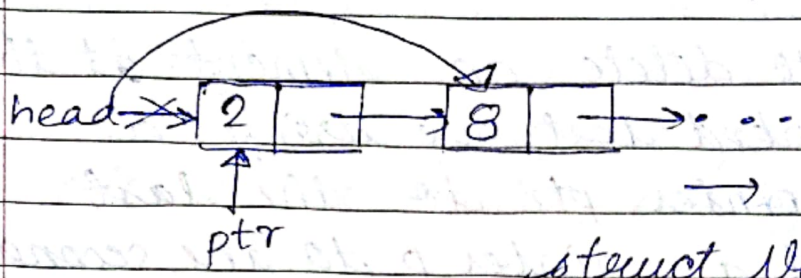
1) Deleting the first Node
2) Deleting the Node at an index
3) Deleting the last Node
4) Deleting the first Node with a given value

We would just need to free the extra node left after we disconnect it from the list. Before that we overwrite the current connection & make new connections. & that is how we delete a node from desired place.

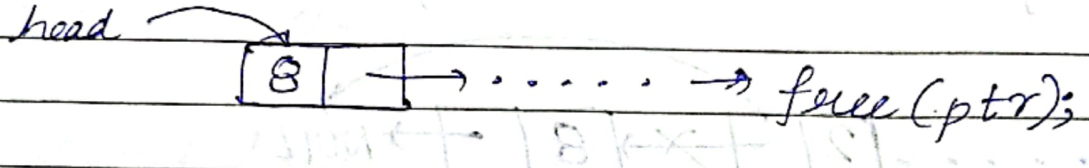
Syntax for freeing a node:

```
free(ptr);
```

Case 1: Delete at beginning

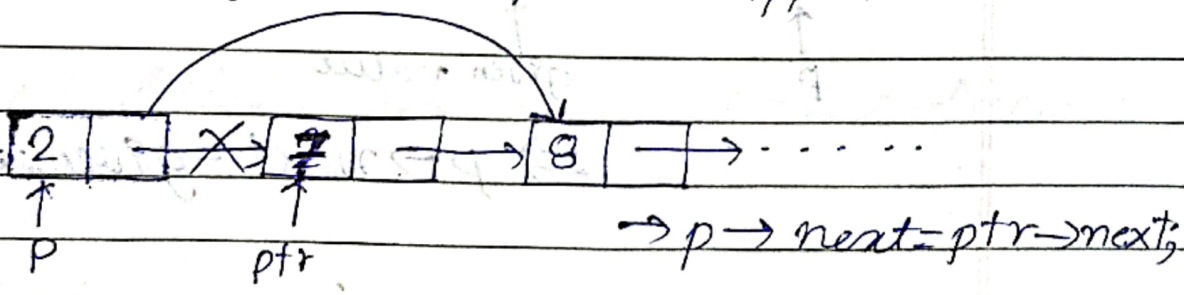


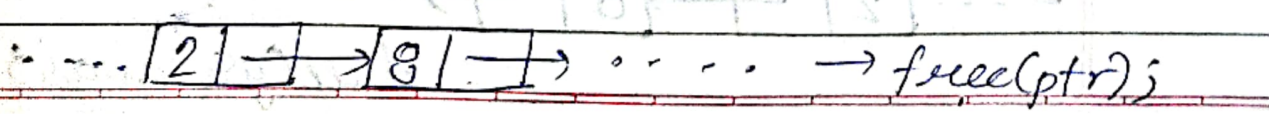
struct Node * ptr = head
head = head → next



Case 2: Deleting at some index in between

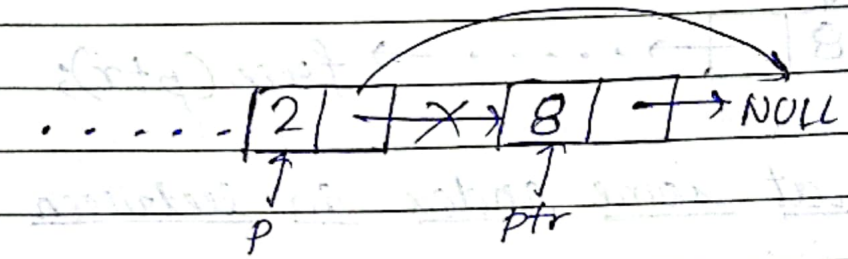
Assuming index starts from 0, we can delete an element from index i > 0 as follows:
Using a temp. pointer p pointing to the node before the element you want to delete in the linked list
Since we want to delete element between 2 & 8, we bring p to 2
Ass. ptr points at element we want to delete
We make pointer p to the next node after pointer ptr skipping ptr.
We can now free the pointer skipped





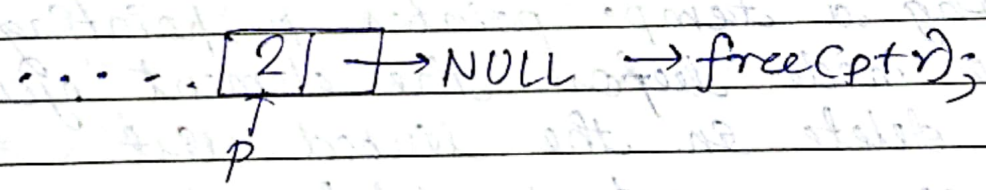
Case 3: Deleting at end

In order to delete an element at the end of linked list, we using a temporary pointer ptr to the last element. And a pointer p to the second last. We make second last element to point at NULL & free ptr.



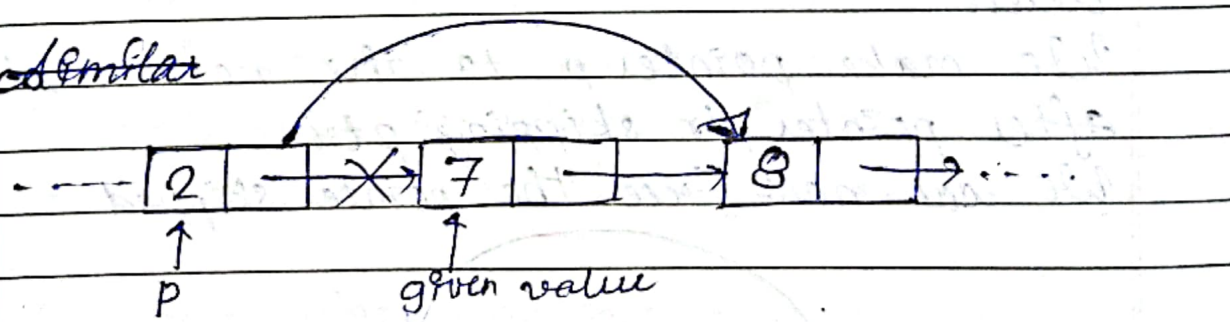
~~→ptr~~

→ p→next = NULL

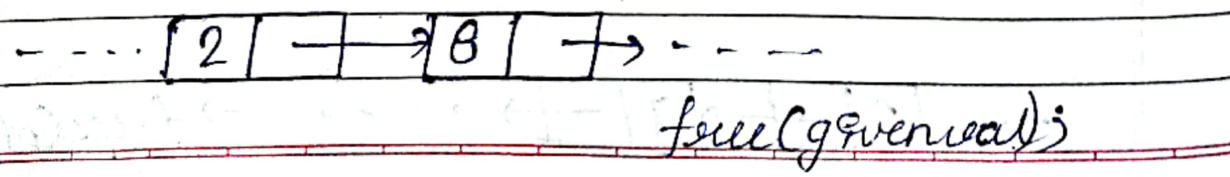


Case 4: Delete the first node with a given value

~~Similar~~



⇒ p→next = givenval→next



free(givenval);

We found that deleting element at beginning completes in O(1). Deleting an index in between or at end or case 4 needs the pointer to reach the node to be deleted, causing it to follow O(n).

Code:

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

struct Node{

    int data;
    struct Node *next;

};

void LinkedListTraversal(struct Node *ptr)
{

while (ptr != NULL)
{
    printf("Element = %d \n", ptr->data);
    ptr = ptr->next;
}
}
// Case 1: Delete the first element from
// Linked List
```

```
struct Node* deleteFirst(struct Node* head)
{
struct Node* ptr = head;
 head = head -> next;
 free(ptr);
 return head;
}

// Case 2: Deleting the element at a given
// index from the Linked List

struct Node* deleteAtIndex(
 struct Node* head, int index) {

struct Node* p = head;
struct Node* q = head -> next;

for(int i=0; i<index-1; i++){

    p = p -> next;
    q = q -> next;
}
p -> next = q -> next
   free(q);
   return head;
}

// Case 3: Deleting the last element
```

In case 4: q->next != NULL is to check whether the element is present in the LL or not

Date
Page

```
struct Node* deleteAtLast(struct Node* head)
{
  struct Node* p = head;
  struct Node* q = head->next;

  while(q->next != NULL){

    p = p->next;
    q = q->next;
  }

  p->next = NULL;
  free(q);
  return head;
}

// Case 4: Deleting an element with a given
// value from Linked List

struct Node* deleteAtIndex(struct Node* head,
                           int value)
{
struct Node* p = head;
struct Node* q = head->next;

while(q->data != value && q->next != NULL)
  {
p = p->next;
  q = q->next;

  }
```

```
if(q->data == value)
{
    p->next = q->next;
    free(q);
}
return head;
}

int main()
{
struct Node *head;
struct Node *second;
 "     "  * third;
 "     "  * fourth;

// Allocate memory for nodes in Linked List
in Heap

head = (struct Node*)malloc(sizeof(struct Node));
second = " "  "  "  "  "  "  "  "  "  "  "  "
third = " "  "  "  "  "  "  "  "  "  "  "  "
fourth = " "  "  "  "  "  "  "  "  "  "  "  "

//Link first & second nodes
 head->data = 4;
 head->next = second;

//Link second & third nodes
 second->data = 3;
 second->next = third;
```

```
// Link third & fourth nodes

    third -> data = 8;
    third -> next = fourth;

// Terminating the list

    fourth -> data = 1;
    fourth -> next = NULL;

printf("Linked list before deletion \n");
linkedListTraversal(head);

x   // head = deleteFirst(head);
y   // head = deleteAtIndex(head, 2);
z   head = deleteAtLast(head);

printf("Linked list after deletion\n");
linkedListTraversal(head);

    return 0;
}
```

o/p

```
Linked list before deletion
Element: 4
--11--: 3
--11--: 8
--11--: 1

Linked list after deletion
Element: 4
--11--: 3
--11--: 8
```