01418231
Data Structures

LECTURE-2-ADT & LINKED-LIST

Powered by
Dr. Jirawan Charoensuk

Agenda

- What is ADT?
- Linked-List
- Types of link-list
- Operations of Linked-list
  - Traverse an item in the list
  - Insert an item in the list
  - Delete an item from the list
- Summary

What is ADT?

Abstract Data Types

Abstract data type (ADT) is an abstract of a data structure

An ADT is composed of

- A collection of data
- A set of operations on that data

Specifications of an ADT indicate

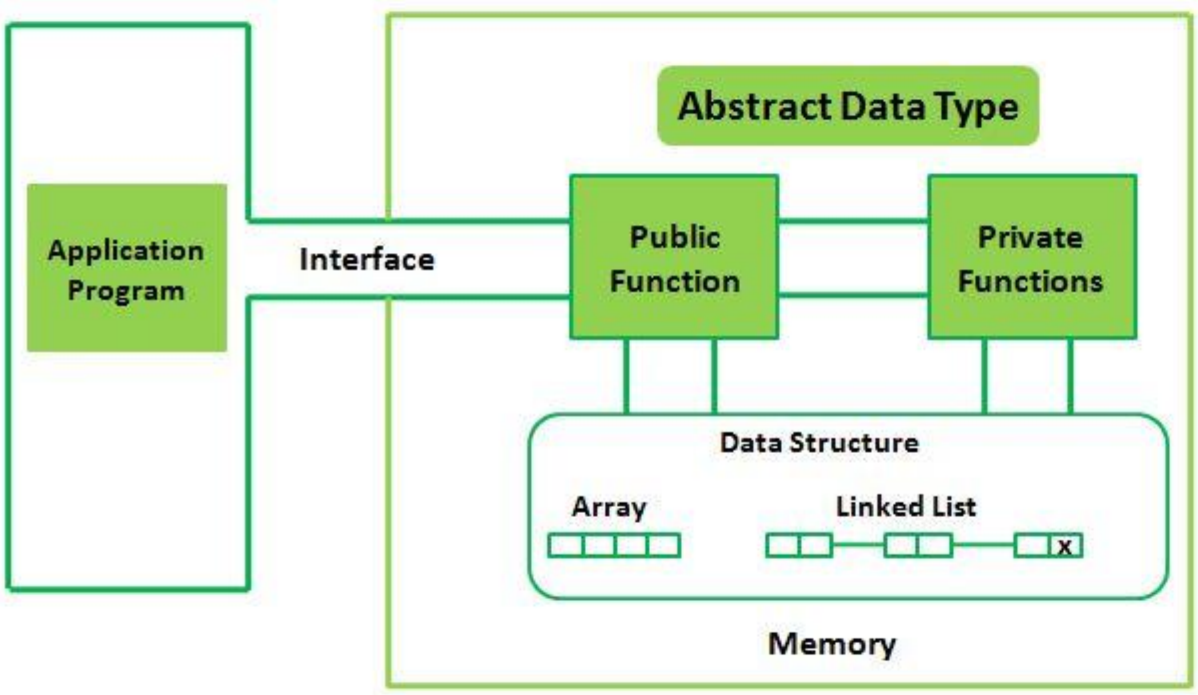
- What the ADT operations do, not how to implement them

Implementation of an ADT

- Includes choosing a particular data structure

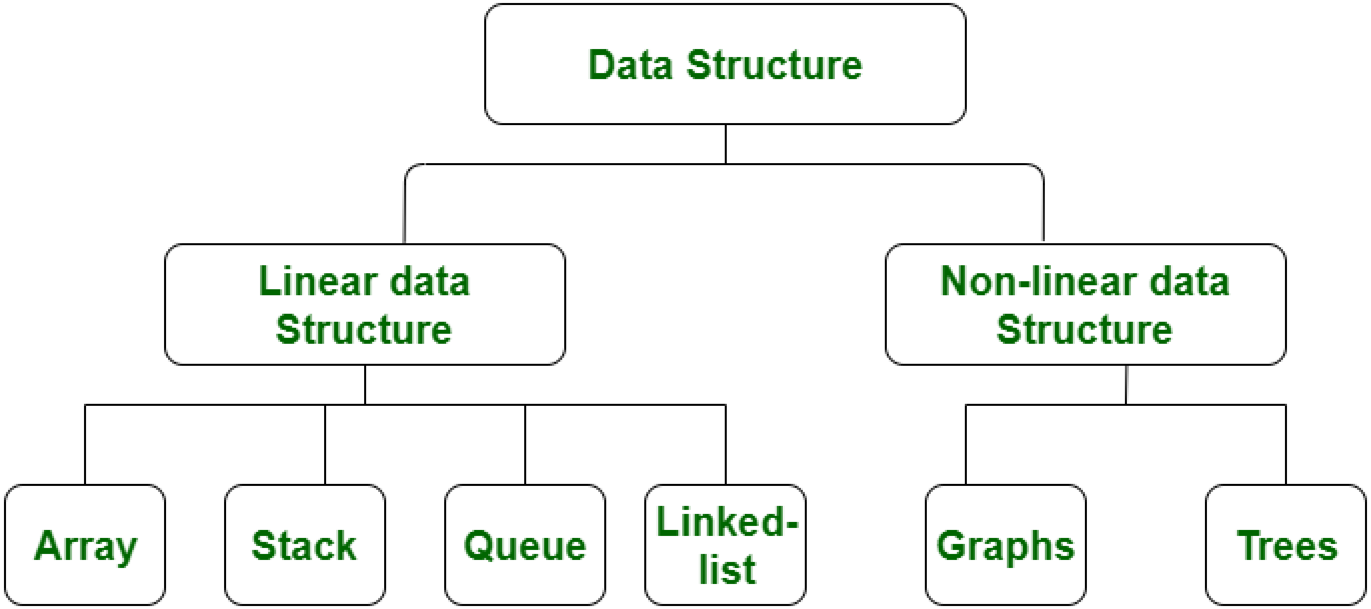
Abstract Data Types

ADT = Type + Function names + Behaviour of each Function



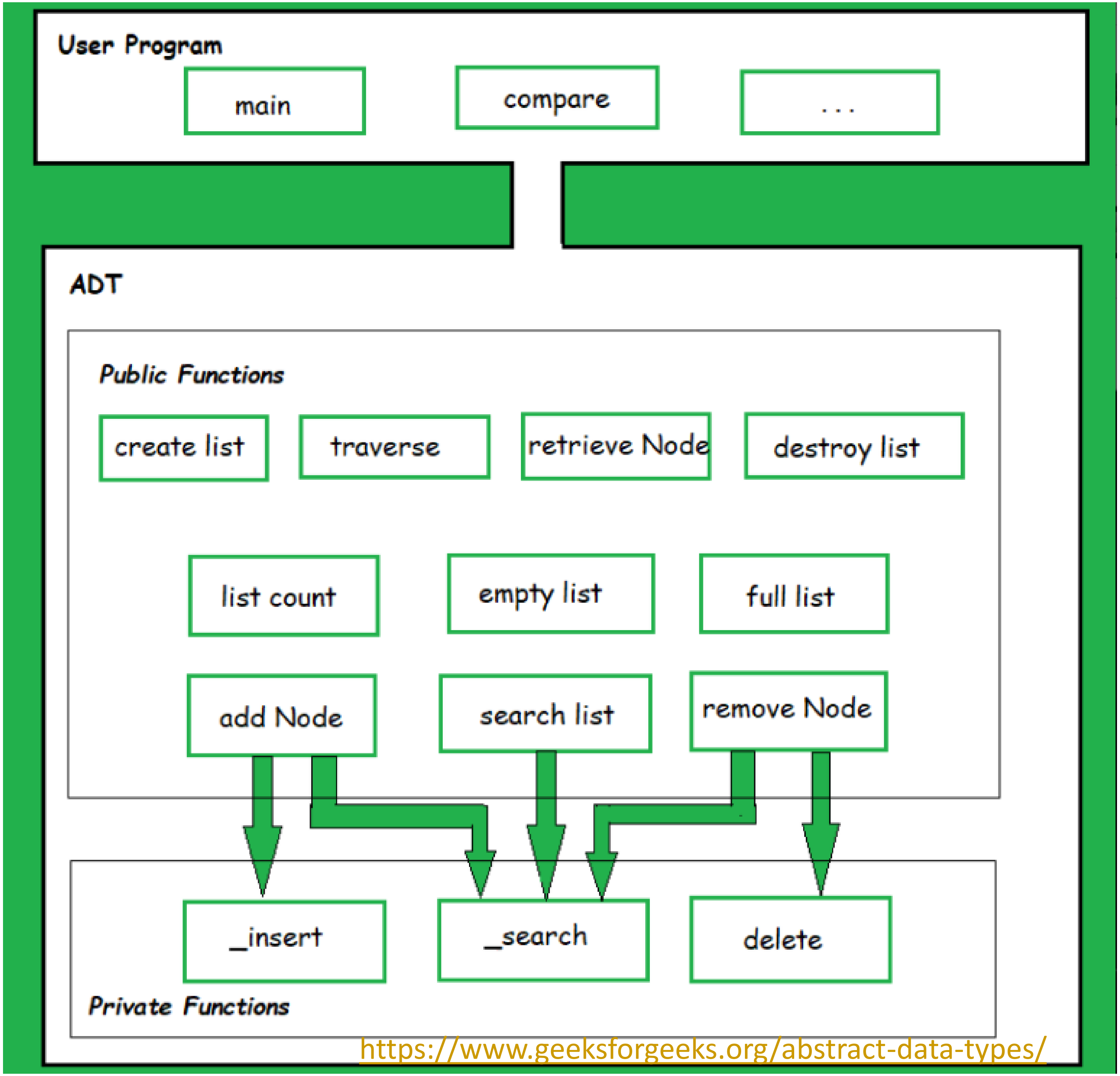
https://www.tutorialscan.com/data_structure/abstract-data-types/

Abstract Data Types



https://www.geeksforgeeks.org/difference-between-linear-and-non-linear-data-structures/

Linked-List

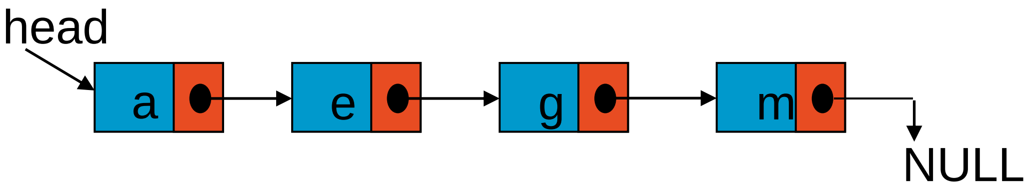


https://www.geeksforgeeks.org/abstract-data-types/

ADT

1. Linear data structures

- Array
- Linked-List
  - What is linked-list ?
  - Types of linked-list?
  - Operations of Linked-list
- Stack
- Queue



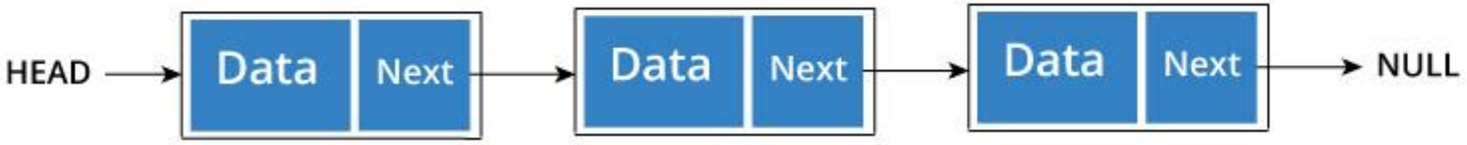
What is Linked-list?

https://www.programiz.com/dsa/linked-list

Linked Lists

- Definition
  - a list of items, called nodes
- Every node in a linked list has two components
  - one to store the information (data)
    - Integer, Float, Char, String
  - one to store the address of the next node in the list, or called the next

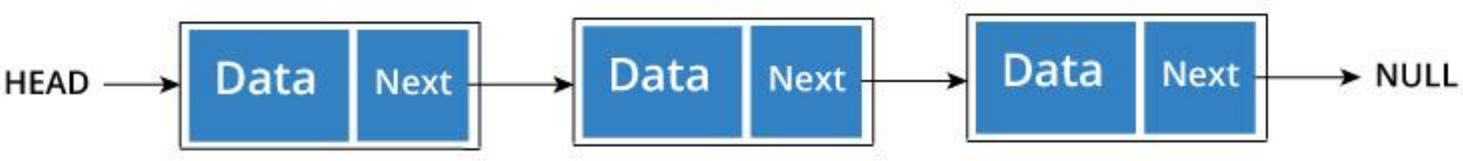
Structure of a node



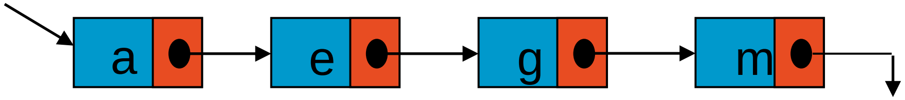
https://www.programiz.com/dsa/linked-list

Linked Lists

- The address of the first node in the list is stored in a separate location, called the head (or first)
  - head should always point to the first node
  - last node point to the end node (Null)



https://www.programiz.com/dsa/linked-list



Linked Lists

- Why Linked List?
  - The size of the arrays is fixed, Linked list allocated memory is equal to the upper limit irrespective of the usage.
  - Inserting a new element and ordered in an array of elements is expensive, Linked list only crated elements and shifted it.
- For example, in a system if we maintain a sorted list of IDs in an array id[].
  - id[] = [100, 110, 115, 120, 125]
- And if we want to insert a new ID 105, then to maintain the sorted order, we have to move all the elements after 100 (excluding 100)

https://www.geeksforgeeks.org/linked-list-set-1-introduction/

Linked Lists

Advantages over arrays

- **1)** Dynamic size
- **2)** Ease of insertion/deletion

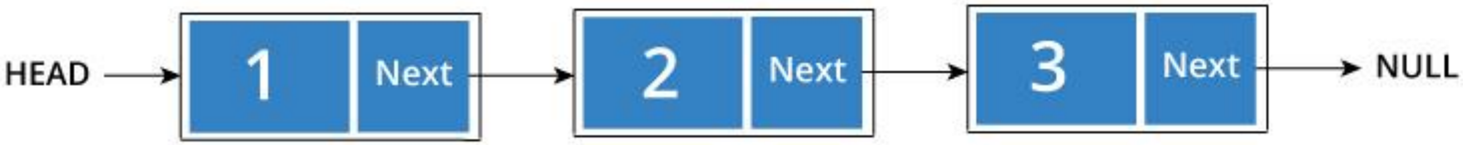
Drawbacks:

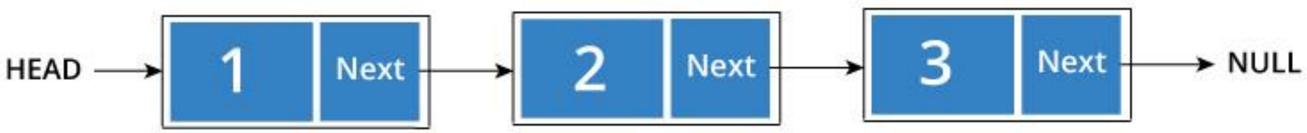
- 1) Random access is not allowed. We have to access elements sequentially starting from the first node.
- 2) Extra memory space for a pointer is required with each element of the list.
- 3) Not cache friendly. Since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.

https://www.geeksforgeeks.org/linked-list-set-1-introduction/

Example of Linked Lists in C

```
struct node
{


};
```





```
struct node
{
    int data;
    struct node *next;
};
```

```
1)/* Initialize nodes */

2)struct node *head;

3)struct node *one = NULL;

4)struct node *two = NULL;

5)struct node *three = NULL;


6)/* Allocate memory */

7)one = malloc(sizeof(struct node));

8)two = malloc(sizeof(struct node));

9)three = malloc(sizeof(struct node));
```

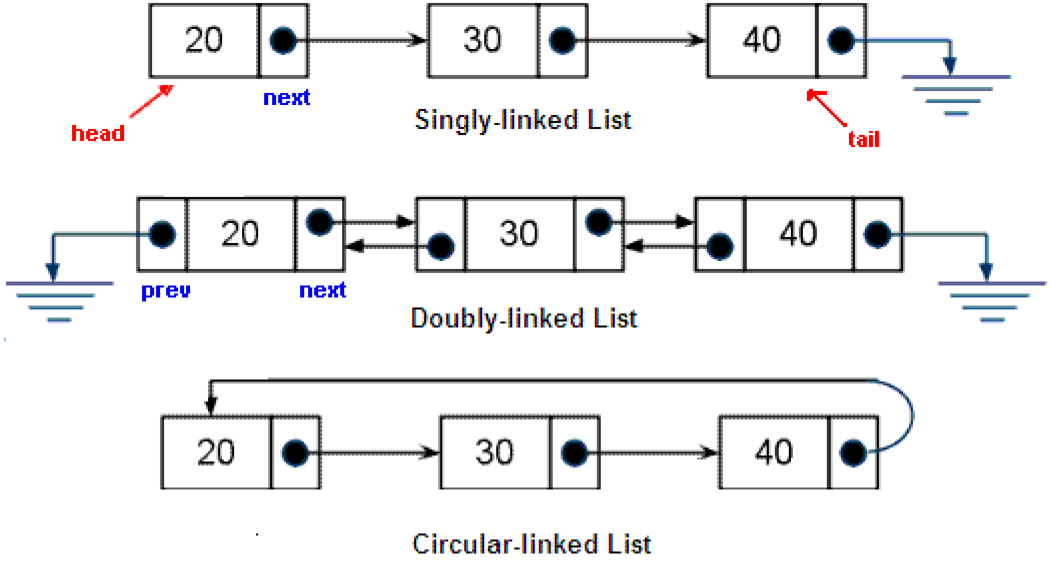
```
10)  /* Assign data values */

11)  one->data

12)  two->data

13)  three->data


14)  /* Connect nodes */

15)  one->next

16)  two->next

17)  three->next


18)  /* Save address of first node in head */

19)  head = one;
```

Types of linked-list?

HTTPS://WWW.PROGRAMIZ.COM/DSA/LINKED-LIST-TYPES#SINGLY%20LINKED%20LIST

Types of linked-list?

1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List



https://sites.google.com/site/sarvasite/algorithms/fund-algo/linked-list1

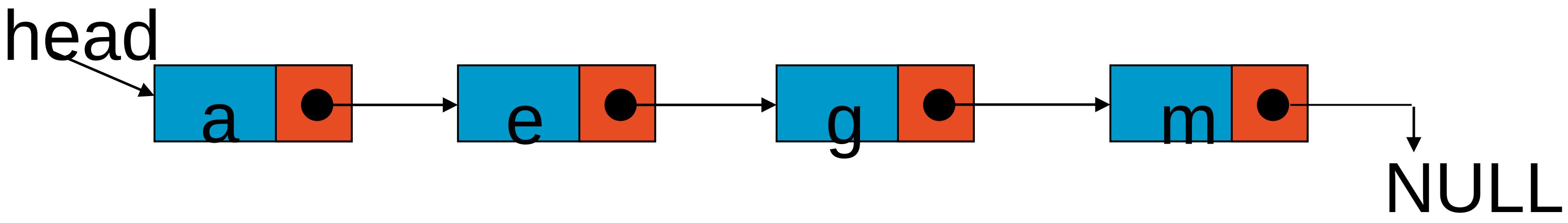
Singly Linked List

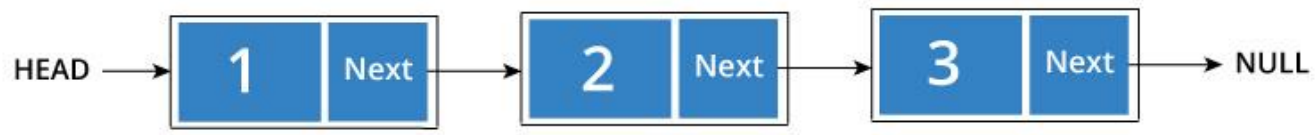
Singly linked list

Begins with a pointer to the first node

Terminates with a null pointer

Only traversed in *one direction*





```
struct node
{
    int data;
    struct node *next;
};
```

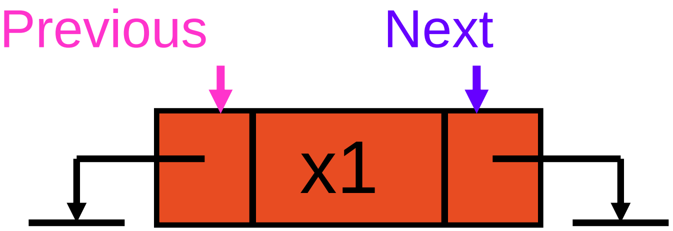
```
/* Initialize nodes */
struct node *head;
struct node *one = NULL;
struct node *two = NULL;
struct node *three = NULL;

/* Allocate memory */
one = malloc(sizeof(struct node));
two = malloc(sizeof(struct node));
three = malloc(sizeof(struct node));
```

```
/* Assign data values */
one->data = 1;
two->data = 2;
three->data=3;

/* Connect nodes */
one->next = two;
two->next = three;
three->next = NULL;

/* Save address of first node in head */
head = one;
```

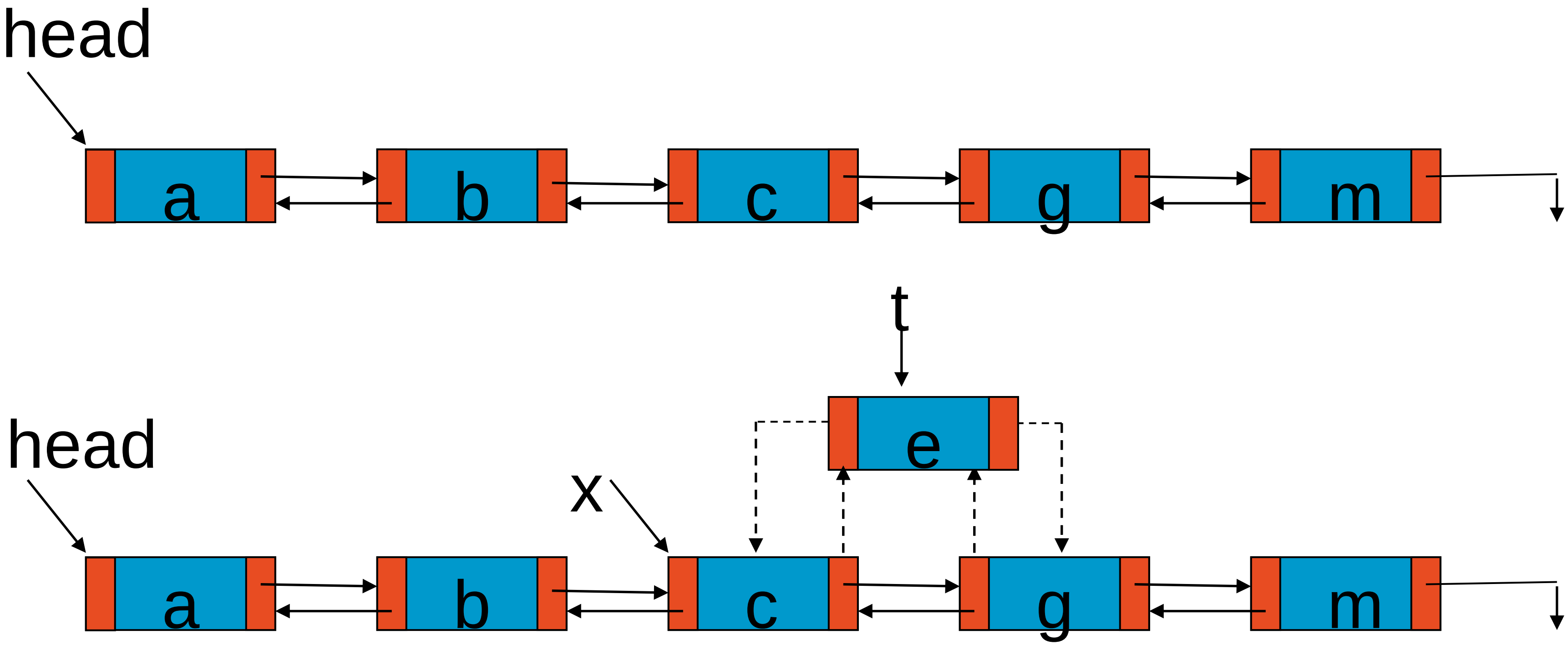
Doubly Linked List

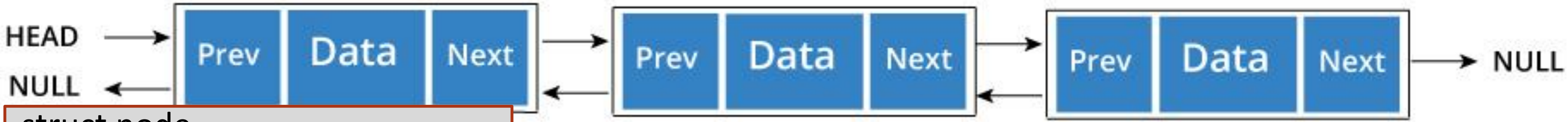
Doubly linked list

- Two “start pointers” – first element and last element
- Each node has a previous pointer and a next pointer
- Allows traversals both forwards and backwards
- Compared to single list inserting and deleting nodes is a bit slower as both the links had to be updated
- Requires the extra storage space for the second list



Doubly Linked List





```
struct node
{
    int data;
    struct node *next;
    struct node *prev;
};
```

```
1)  /* Initialize nodes */
2)  struct node *head;
3)  struct node *one = NULL;
4)  struct node *two = NULL;
5)  struct node *three = NULL;

6)  /* Allocate memory */
7)  one = malloc(sizeof(struct node));
8)  two = malloc(sizeof(struct node));
9)  three = malloc(sizeof(struct node));
```

```
10) /* Assign data values */
11) one->data = 1;
12) two->data = 2;
13) three->data=3;

14) /* Connect nodes */
15) one->next = two;  one->prev = NULL;
16)
17)

18) /* Save address of first node in head */
19) head = one;
```

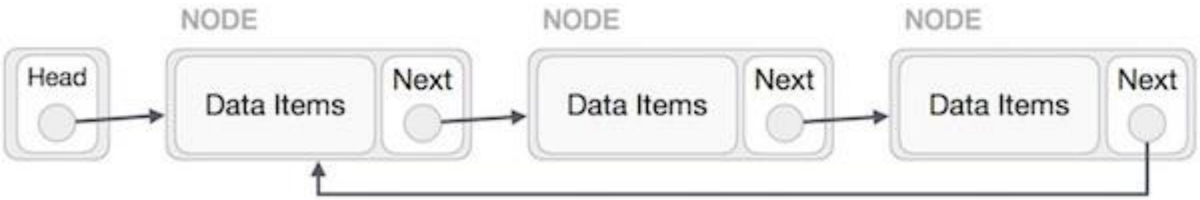
https://www.programiz.com/dsa/linked-list-types

Circular Linked List

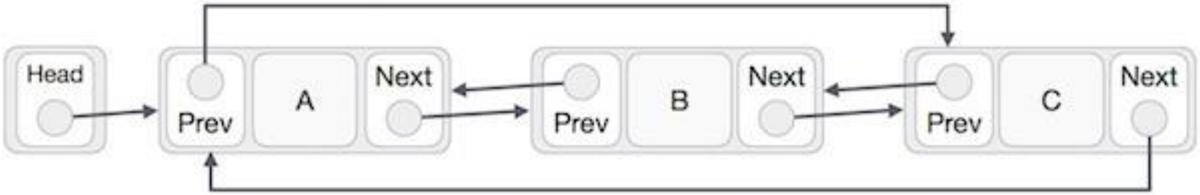
SINGLY LINKED LIST, DOUBLY LINKED LIST

Circular Linked List

Circular Singly Linked List



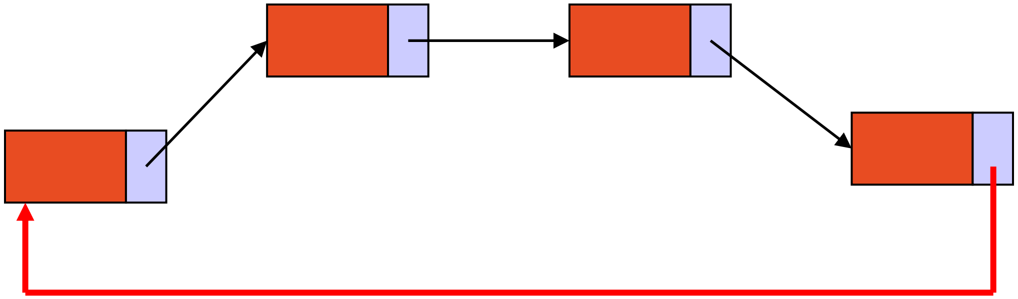
Circular Doubly Linked List



https://www.tutorialspoint.com/data_structures_algorithms/circular_linked_list_algorithm.htm

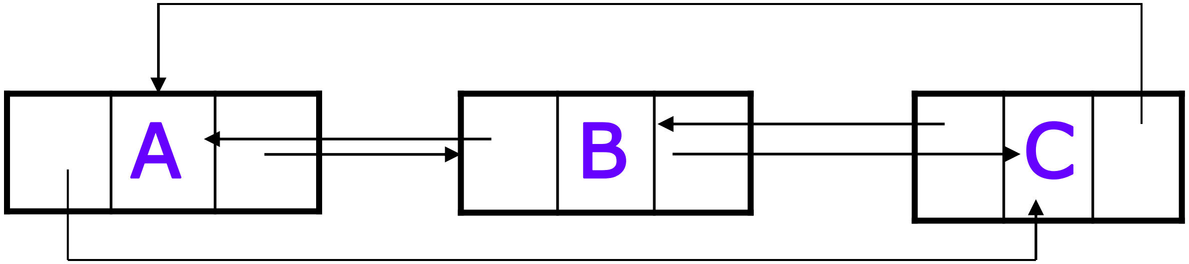
Circular, singly linked

- Pointer in the last node points back to the first node

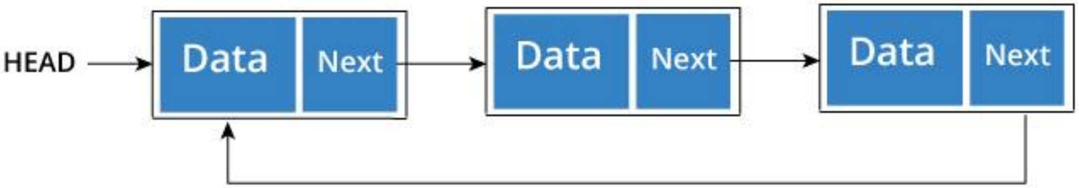


Circular, doubly linked list

- Similar to the Double linked
- But, the previous pointer of the last node points to the first node
- The next pointer of the first node points to the last node
- Advantage is that we can make head to refer to any node without destroying the list



Circular Singly Linked List



```
struct node
{
     int data;
     struct node *next;
};
```

```
1)   /* Initialize nodes */
2)   struct node *head;
3)   struct node *one = NULL;
4)   struct node *two = NULL;
5)   struct node *three = NULL;

6)   /* Allocate memory */
7)   one = malloc(sizeof(struct node));
8)   two = malloc(sizeof(struct node));
9)   three = malloc(sizeof(struct node));
```

```
10)  /* Assign data values */
11)  one->data = 1;
12)  two->data = 2;
13)  three->data = 3;

14)  /* Connect nodes */
15)  one->next = two;
16)
17)

18)  /* Save address of first node in head */
19)  head = one;
```

ıttps://www.programiz.com/dsa/linked-list-types

Operations of Linked-list

Linked Lists: Operations

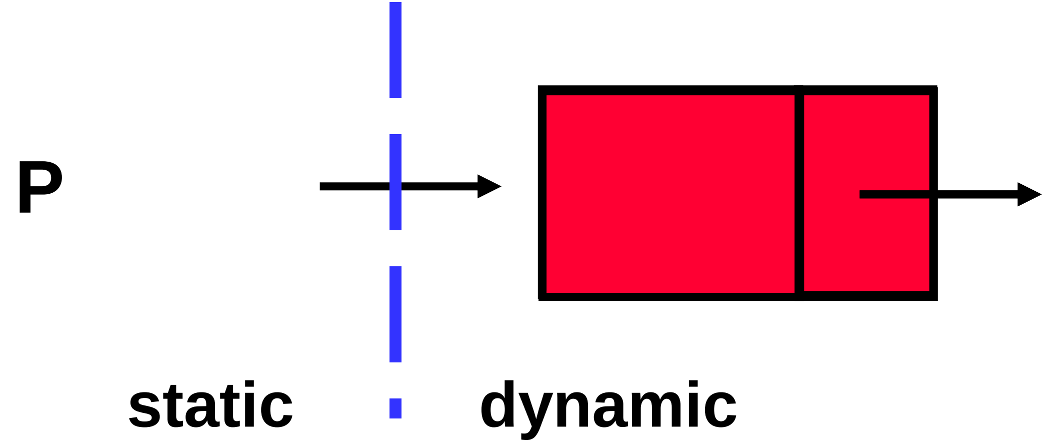
Linked list basic operations:

- Traverse an item in the list
- Insert an item in the list
- Delete an item from the list

Traverse

Pointers and Linked Lists

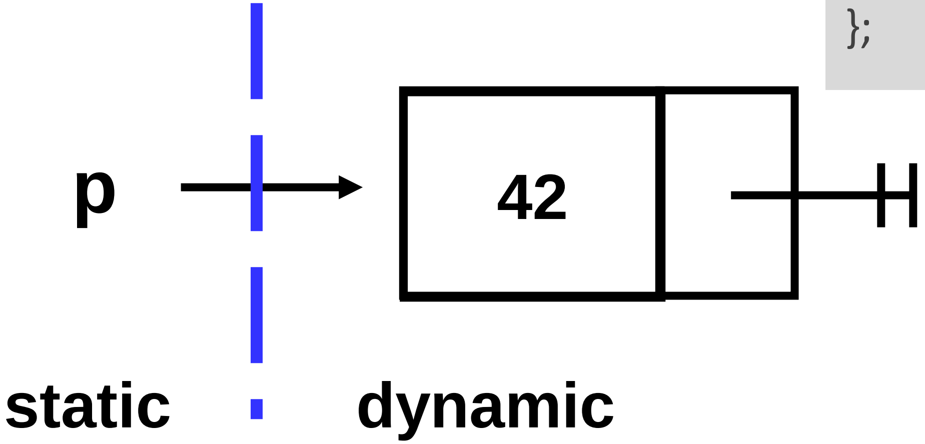
```
struct node
{
  int data;
  struct node *next;
};
```



```
P->
P->data
P->next
```

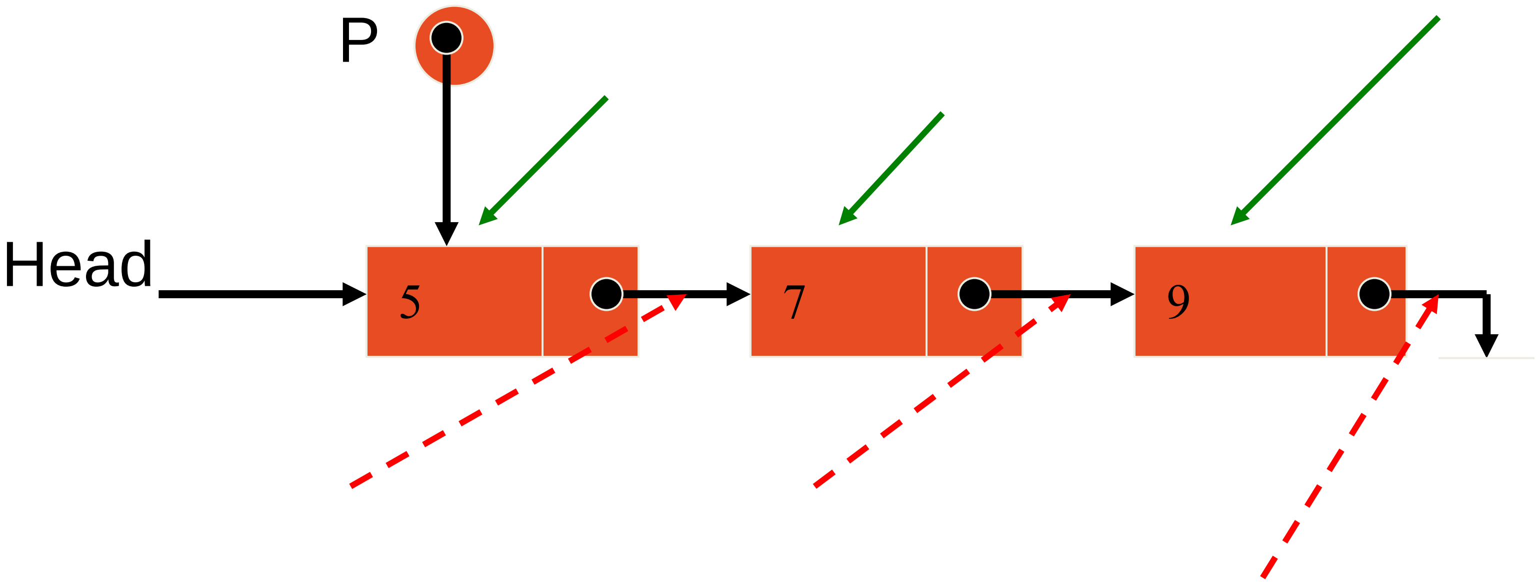
Accessing the Data Field of a Node

```
struct node
{
  int data;
  struct node *next;
};
```



```
P->data

P->next
```

Example: Traverse



Node Definition

```
struct node
{
  int data;
  struct node *next;
};
```

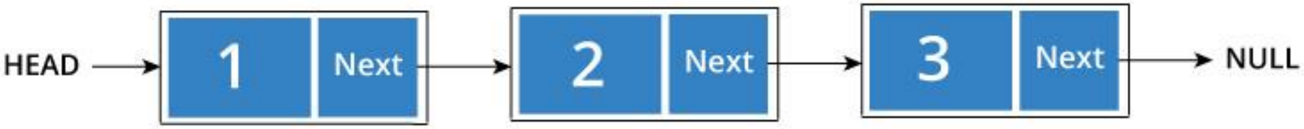
```
/* Initialize nodes */
struct node *head;
struct node *one = NULL;
struct node *two = NULL;
struct node *three = NULL;

/* Allocate memory */
one = malloc(sizeof(struct node));
two = malloc(sizeof(struct node));
three = malloc(sizeof(struct node));
```

```
/* Assign data values */
one->data = 1;
two->data = 2;
three->data = 3;

/* Connect nodes */
one->next = two;
two->next = three;
three->next = NULL;

/* Save address of first node in head */
head = one;
```

Linked Lists: Traverse

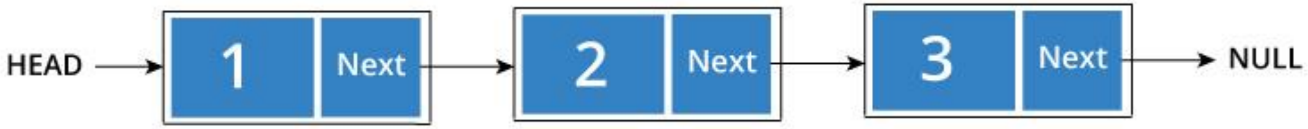
- Traverse: given a pointer to the first node of the list, step through each of the nodes of the list
- Traverse a list using a pointer of the same type as head
- Example:
  - assume temp is a pointer of nodeType and head points to the first node in the linked list

```
struct node *temp = head;
while(temp != NULL)
{
        printf("%d --->",temp->data);
        temp = temp->next; // Handle the node pointed to by temp
}
```



```
struct node
{
  int data;
  struct node *next;
};
```

```
/* Assign data values */
one->data = 1;
two->data = 2;
three->data = 3;

/* Connect nodes */
one->next = two;
two->next = three;
three->next = NULL;

/* Save address of first node in head */
head = one;
```



```
struct node
{
  int data;
  struct node *next;
};
```

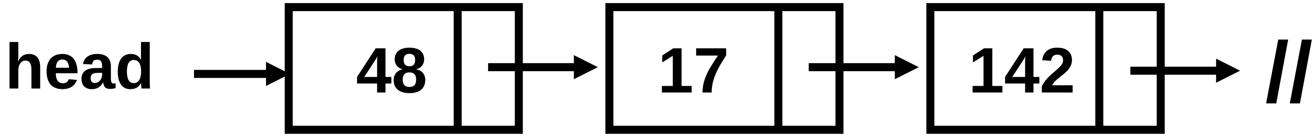
```
/* Assign data values */
one->data = 1;
two->data = 2;
three->data = 3;

/* Connect nodes */
one->next = two;
two->next = three;
three->next = NULL;

/* Save address of first node in head */
head = one;
```

```
/* Assign temp pointer point to head*/
struct node *temp = head;
printf("\n\nList elements are - \n");
while(temp != NULL)
{
     printf("%d --->",temp->data);
     temp = temp->next;
}
```

Insert

The Scenario

- If you have a linked list , check linked list
  - Empty
  - Not empty



Adding an Element to a Linked-List

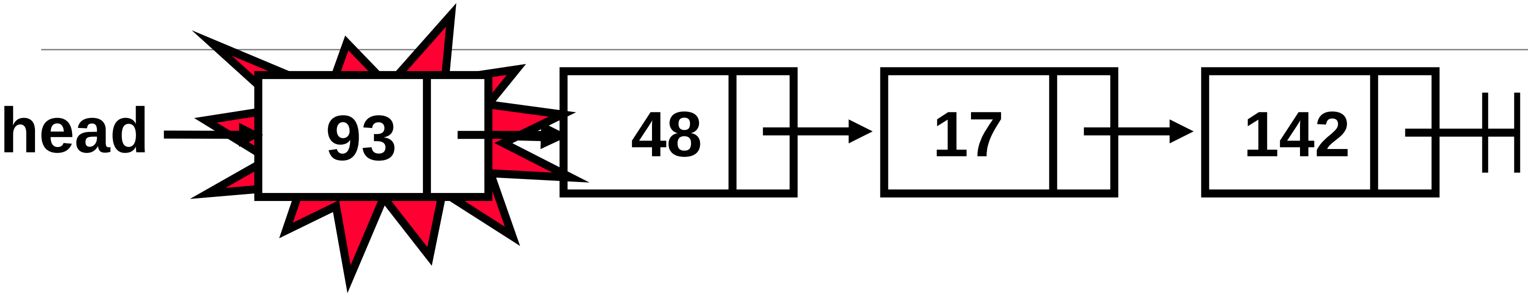
Involves two steps:

1. Finding the correct location
   - Three possible positions:
     - The front
     - The end
     - Somewhere in the middle
2. Add the node

Inserting at the Front of a Linked List

https://www.programiz.com/dsa/linked-list-operations

Inserting to the Front (93)



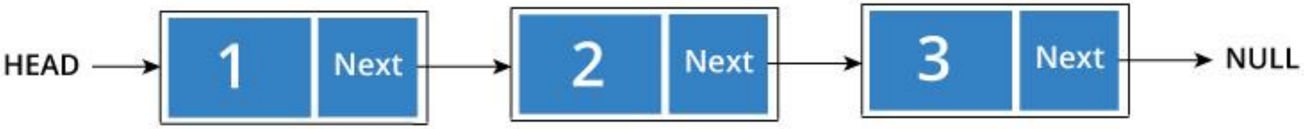
Using head to find the correct location

Empty or not, head will point to the right location

Inserting at the Front of a Linked List

1. Create newNode
2. Allocate memory for new node
3. Store data
4. Change next of new node to point to head
5. Change head to point to recently created node

```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;
newNode->next = head;
head = newNode;
```

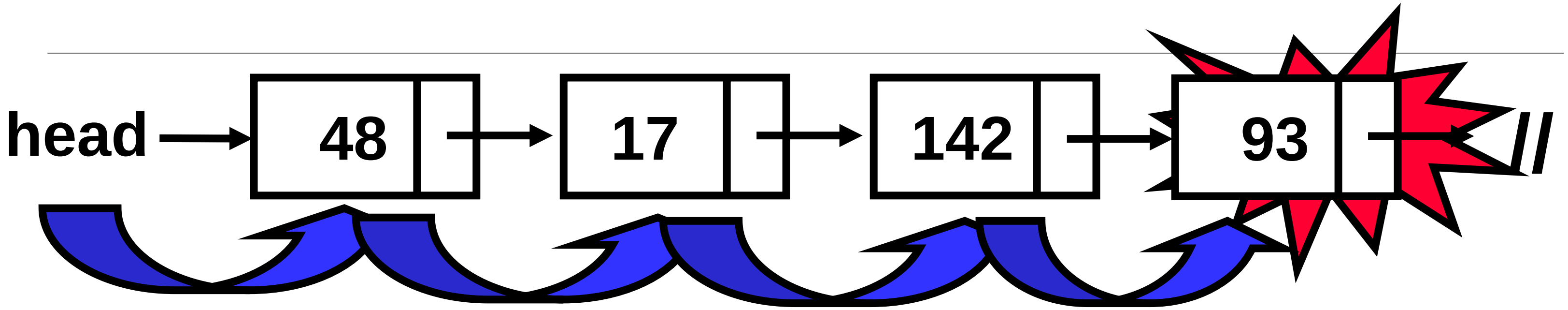


```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;
newNode->next = head;
head = newNode;

struct node *temp = head;
printf("\n\nList elements are - \n");
while(temp != NULL)
{
    printf("%d --->",temp->data);
    temp = temp->next;
}
```

Inserting at the End of a Linked List

https://www.programiz.com/dsa/linked-list-operations

Inserting to the End (93)



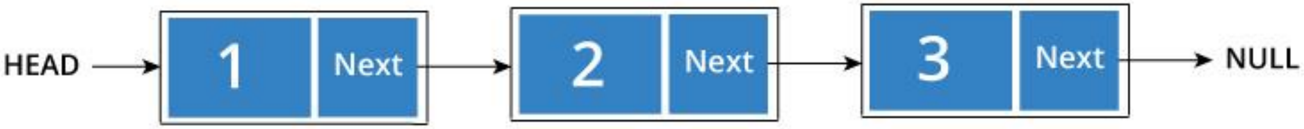
Find the end of the list

- when at NULL
  - Insert after NULL

Inserting at the End of a Linked List

1. Create newNode
2. Allocate memory for new node
3. Store data
4. Set pointer to NULL
5. Traverse to last node
6. Change next of last node to recently created newNode

```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;
newNode->next = NULL;

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

temp->next = newNode;
```

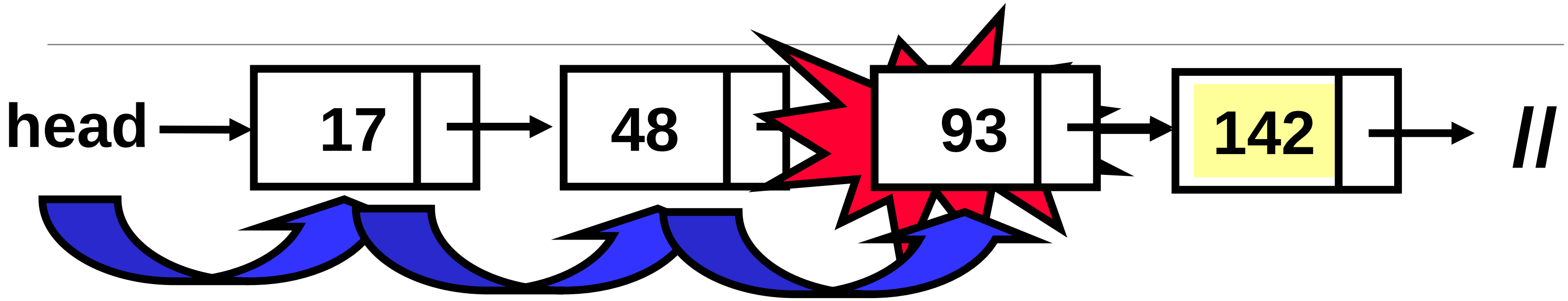


```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;
newNode->next = NULL;


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

```
//print
temp = head;
printf("\n\nList elements are - \n");
while(temp != NULL)
{
     printf("%d --->",temp->data);
     temp = temp->next;
}
```

Inserting in Order into a Linked List

Insert ordering to the Middle (93)



Used when order is important

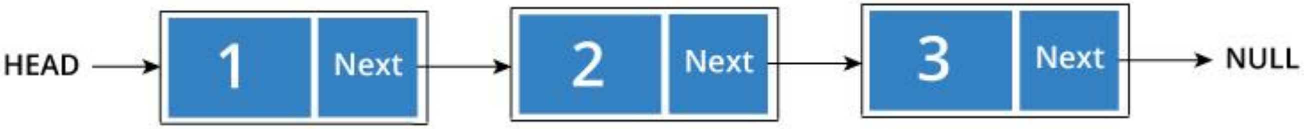
Go to the node that should follow the one to add

Using transverse and compare data

Inserting in Order into a Linked List

1. Create newNode
2. Allocate memory and store data for new node
3. Traverse to node just before the required position of new node
4. Change next pointers to include new node in between

```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 55;

struct node *temp = head;
for(i=2; i < position; i++) {
    if(temp->next != NULL) {
        temp = temp->next;
    }
}
newNode->next = temp->next;
temp->next = newNode;
```



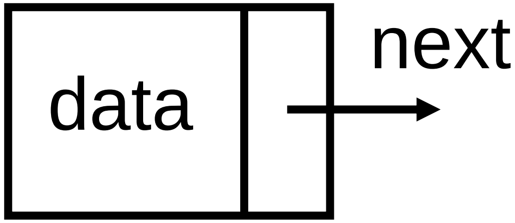
```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 55;
int i,position =3;

struct node *temp = head;
for(i=2; i < position; i++) {
    if(temp->next != NULL) {
        temp = temp->next; }
}
newNode->next = temp->next;
temp->next = newNode;
```

```
//print
temp = head;
printf("\n\nList elements are - \n");
while(temp != NULL)
{
    printf("%d --->",temp->data);
    temp = temp->next;
}
```

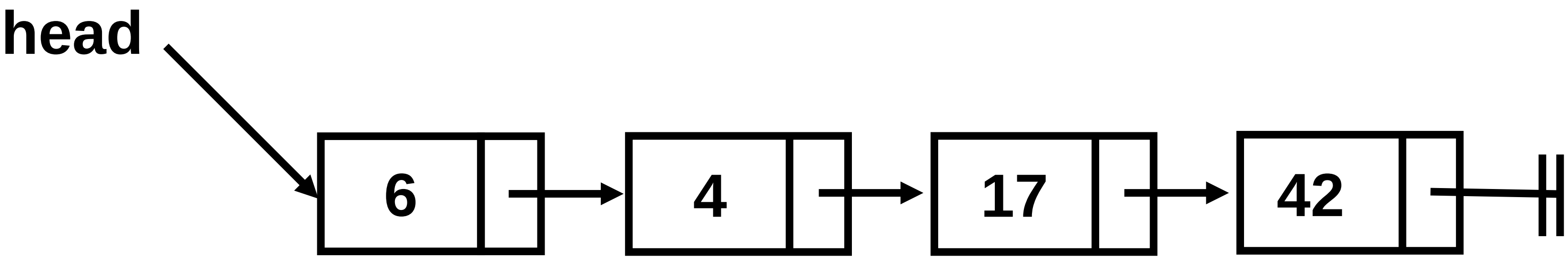
Delete
(Recursive procedure)

The Node Definition

```
Node defines a struct
  data isoftype num
 next isoftype ptr toa Node
endstruct
```



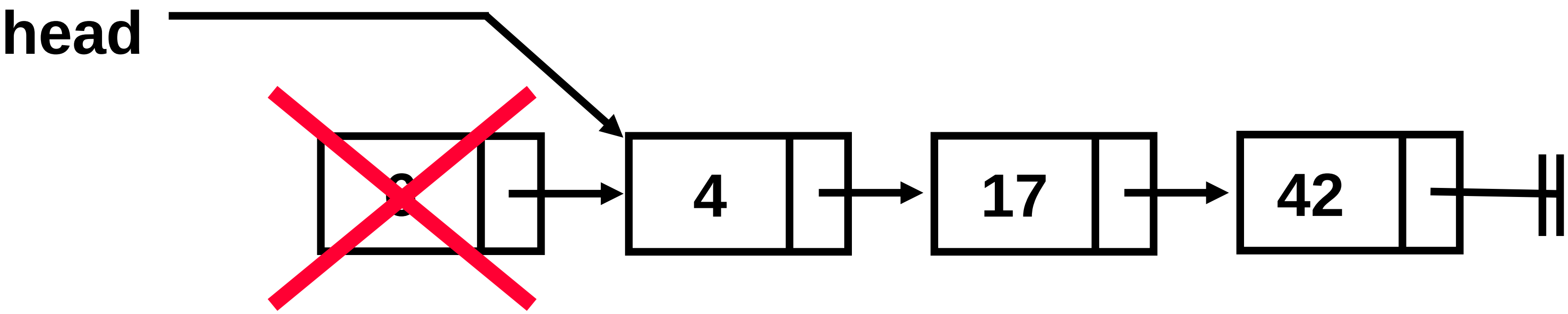
The Scenario



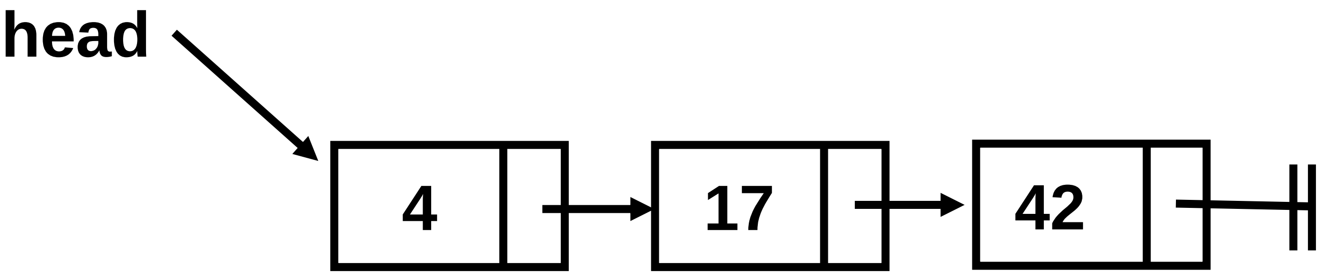
Begin with an existing linked list

- Could be empty or not
- Could be ordered or not

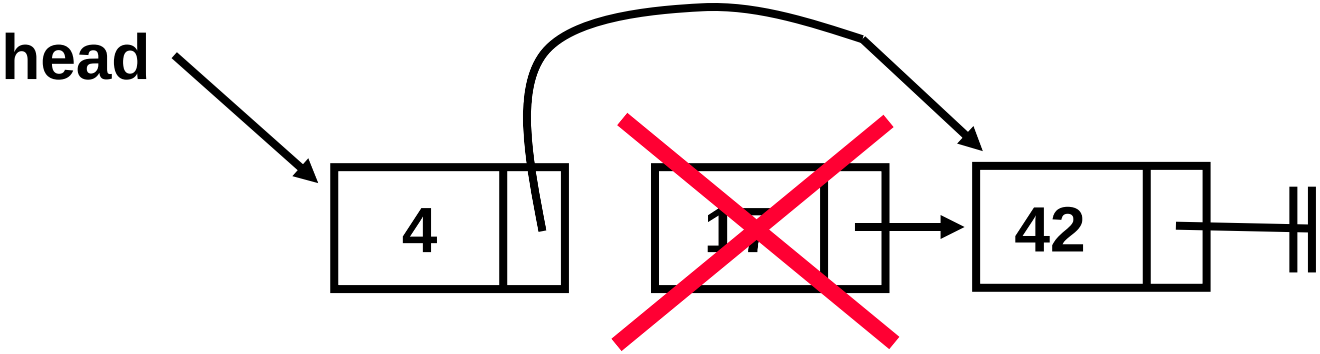
The Scenario



The Scenario



The Scenario



Finding the Match

Three situations for delete:

- Delete the first element
- Delete the first occurrence of an element
- Delete all occurrences of a particular element

Deleting the First Element

- This can be done without any traversal/searching
- Requires an in/out pointer

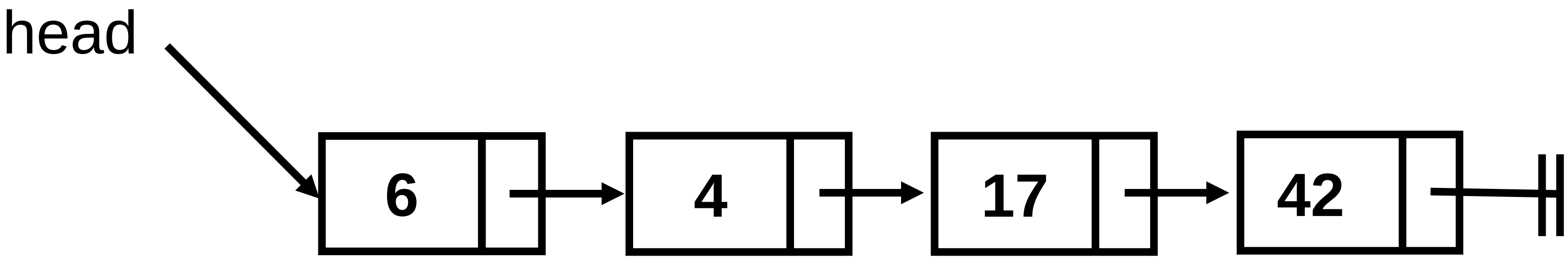
```
procedure DeleteFront
(current iot in/out ptr toa Node)
  // deletes the first node in the list
  if (current <> NULL) then
    current = current->next
  endif
endprocedure
```

Deleting from a Linked List

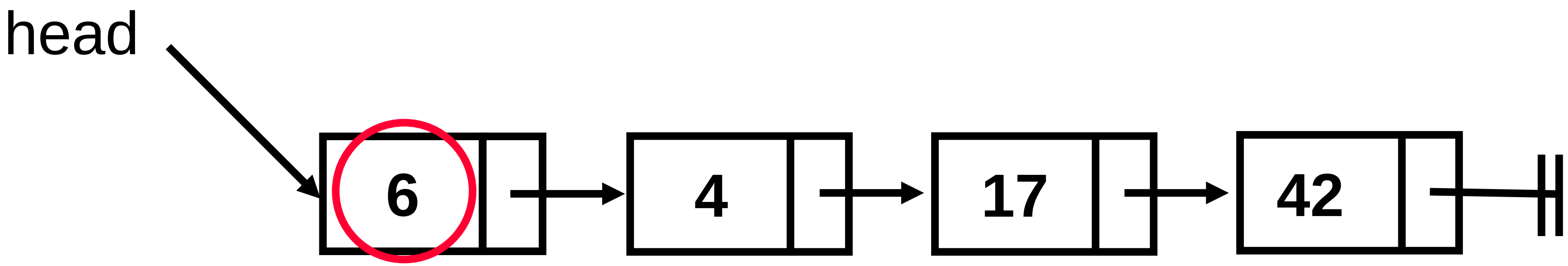
- Deletion from a linked list involves two steps:
  - Find a match to the element to be deleted (traverse until NULL or found)
  - Perform the action to delete
- Performing the deletion is trivial:

  ```
  current = current->next
  ```

  - This removes the element, since nothing will point to the node.

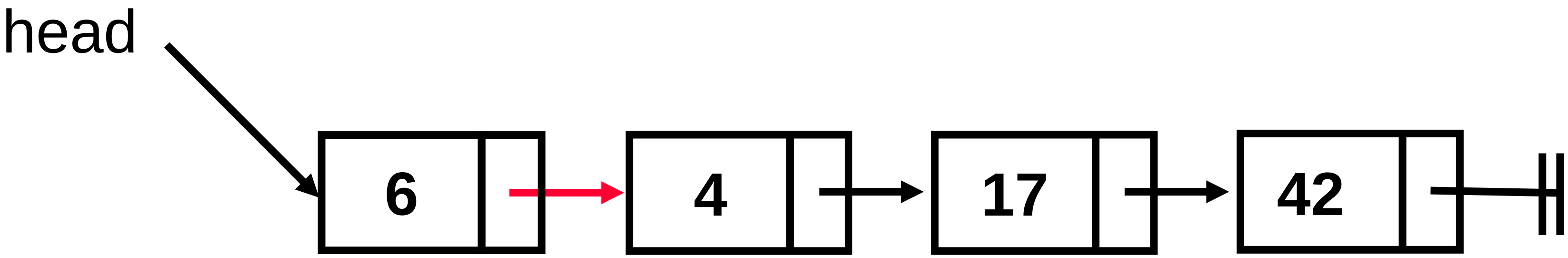


```
.
.
Delete(head, 4)
.
.
```



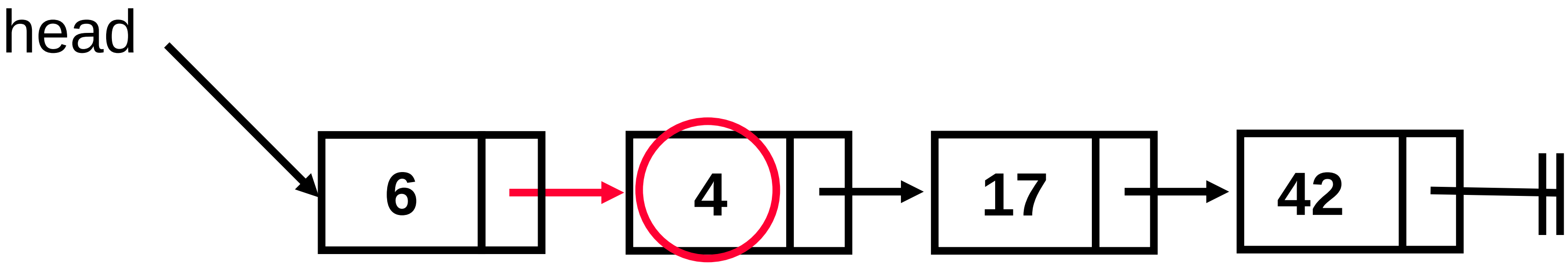
```
procedure Delete(cur iot in/out ptr toa Node,
           target isoftype in num)
// Delete single occurrence of a node.
  if(cur <> NULL) then
    if(cur -> data = target) then
   cur = cur -> next
    else
   Delete(cur -> next, target)
    endif
  endif
endprocedure
```

Target = 4



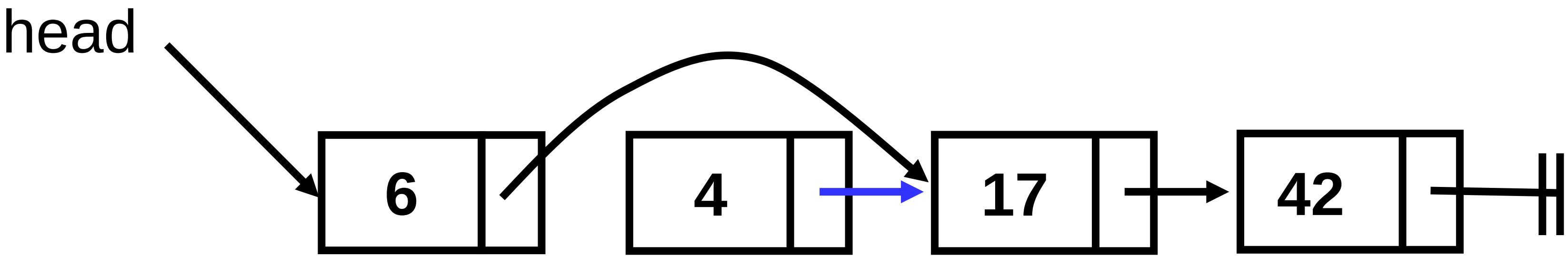
```
procedure Delete(cur iot in/out ptr toa Node,
           target isoftype in num)
// Delete single occurrence of a node.
  if(cur <> NULL) then
    if(cur -> data = target) then
   cur = cur->next
    else
   Delete(cur->next, target)
    endif
  endif
endprocedure
```

Target = 4



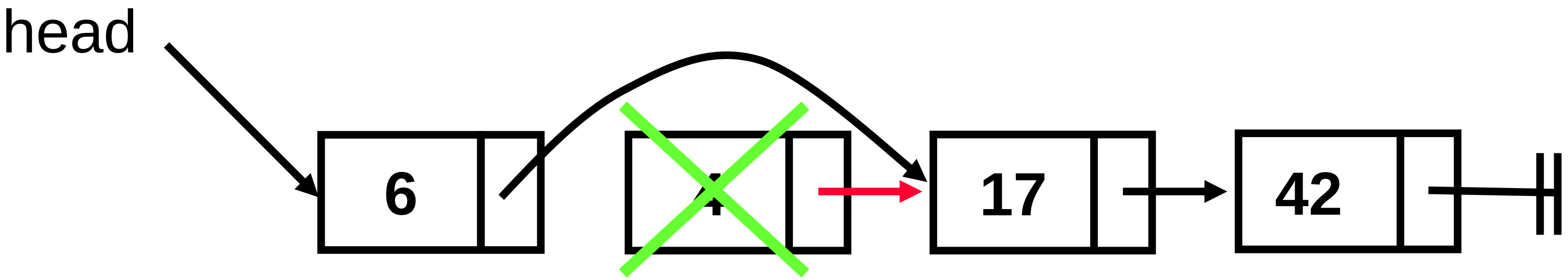
```
procedure Delete(cur iot in/out ptr toa Node,
           target isoftype in num)
// Delete single occurrence of a node.
  if(cur <> NULL) then
    if(cur -> data = target) then
   cur = cur->next
    else
   Delete(cur->next, target)
    endif
  endif
endprocedure
```

Target = 4



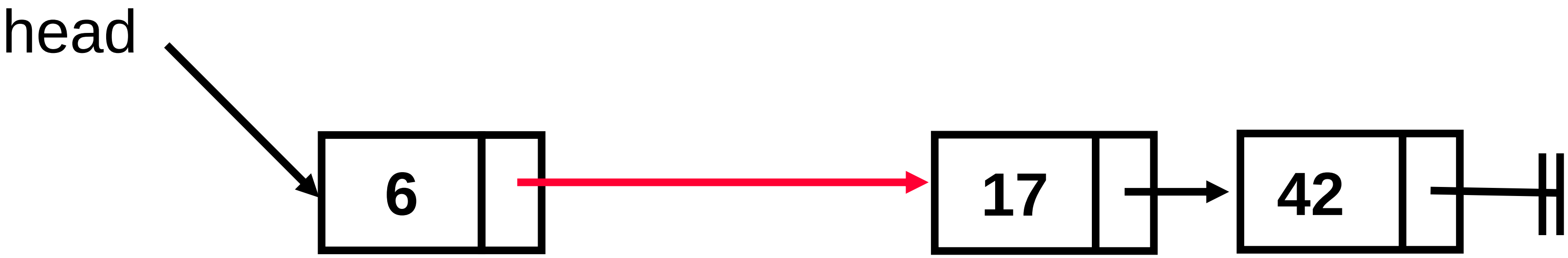
```
procedure Delete(cur iot in/out ptr toa Node,
           target isoftype in num)
// Delete single occurrence of a node.
  if(cur <> NULL) then
    if(cur -> data = target) then
   cur = cur->next
    else
   Delete(cur->next, target)
    endif
  endif
endprocedure
```

Target = 4



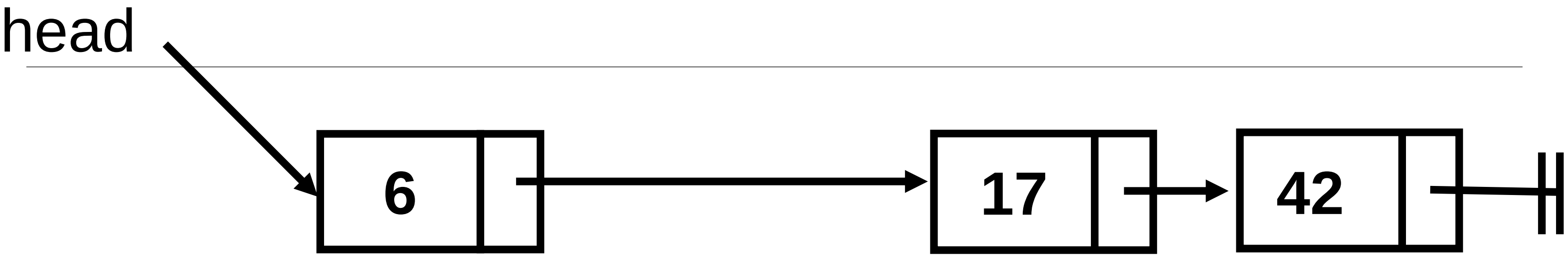
```
procedure Delete(cur iot in/out ptr toa Node,
             target isoftype in num)
// Delete single occurrence of a node.
  if(cur <> NULL) then
    if(cur -> data = target) then
   cur = cur->next
    else
   Delete(cur->next, target)
    endif
  endif
endprocedure
```

Target = 4



```
procedure Delete(cur iot in/out ptr toa Node,
           target isoftype in num)
// Delete single occurrence of a node.
  if(cur <> NULL) then
    if(cur -> data = target) then
   cur = cur->next
    else
   Delete(cur->next, target)
    endif
  endif
endprocedure
```

Target = 4

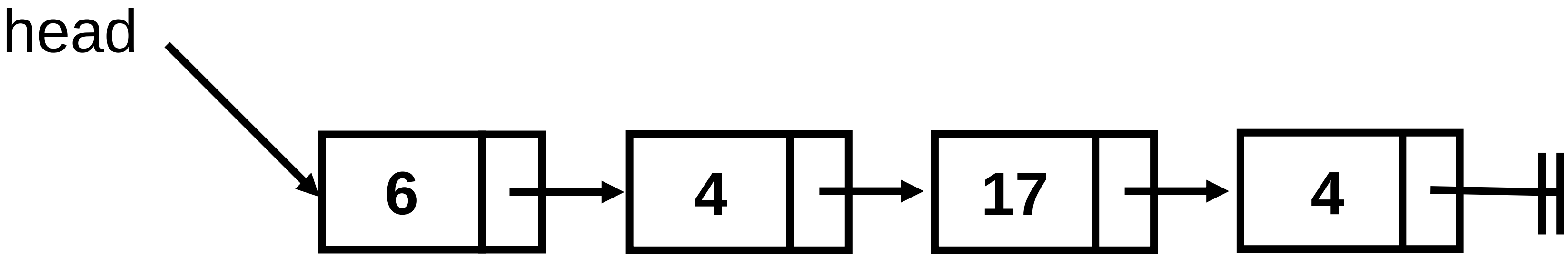


```
.
.
Delete(head, 4)
.
.
```

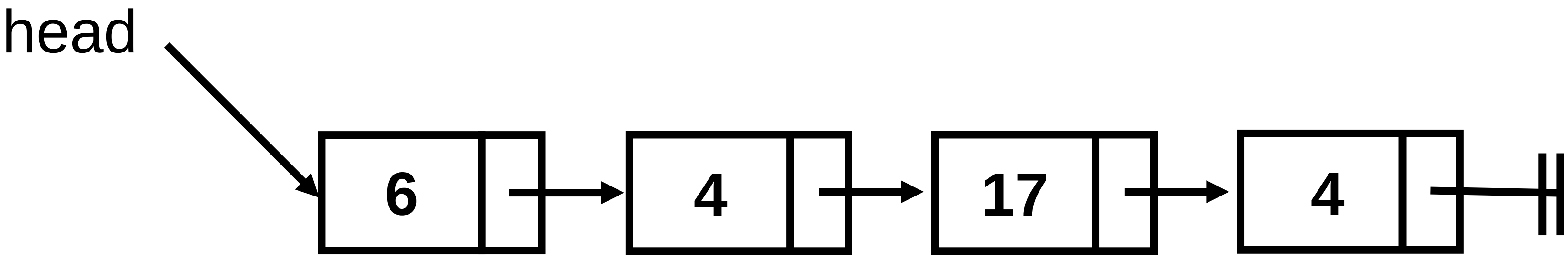
Linked List Deletion (All Occurrences)

Deleting All Occurrences

- Deleting all occurrences is a little more difficult.
- Traverse the entire list and don’t stop until you reach NULL.
- If you delete, recurse on current
- If you don’t delete, recurse on current->next

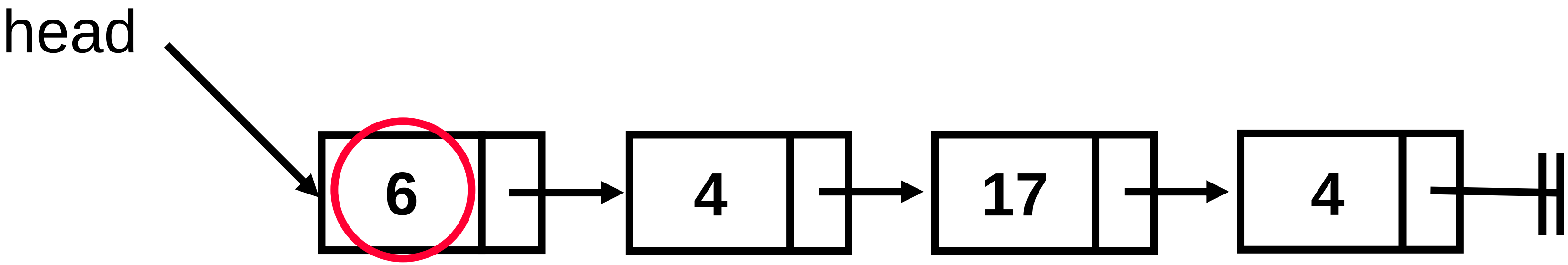


```
.
.
Delete(head, 4)
.
.
```



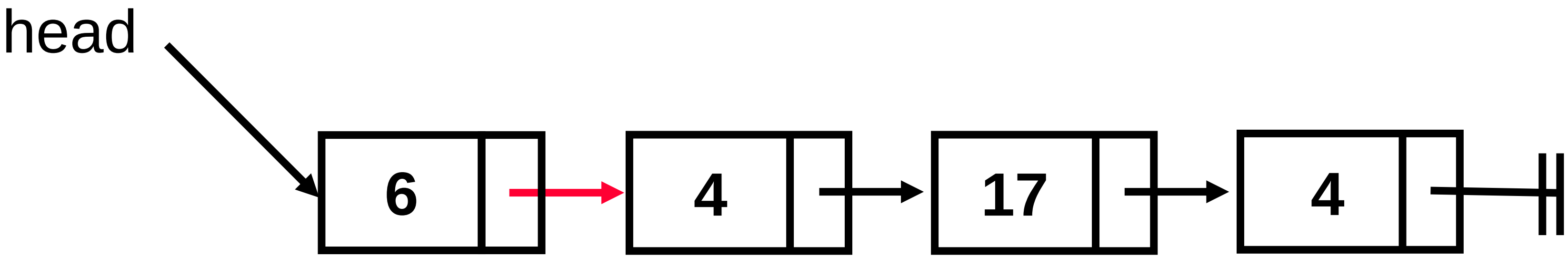
```
procedure Delete(cur iot in/out ptr toa Node,
           target isoftype in num)
// Delete all occurrences of a node.
  if(cur <> NULL) then
    if(cur -> data = target) then
   cur = cur->next
      Delete(cur, target)
    else
   Delete(cur->next, target)
    endif
  endif
endprocedure
```

Target = 4



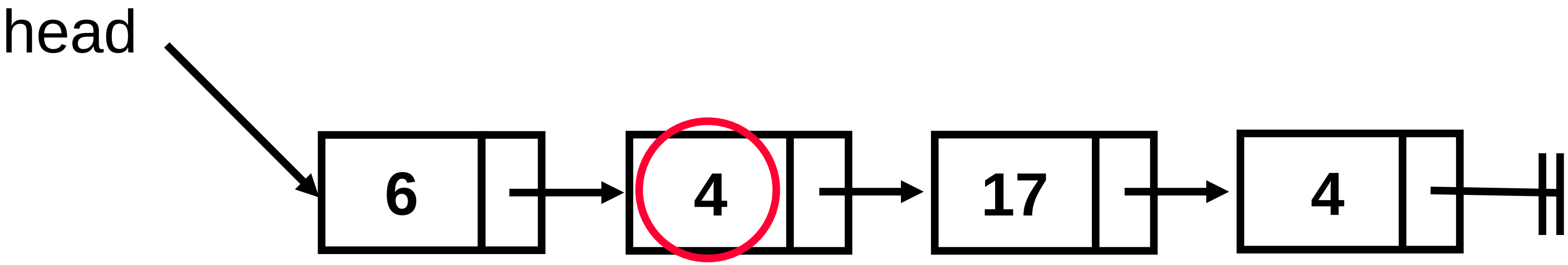
```
procedure Delete(cur iot in/out ptr toa Node,
           target isoftype in num)
// Delete all occurrences of a node.
  if(cur <> NULL) then
    if(cur -> data = target) then
   cur = cur->next
      Delete(cur, target)
    else
   Delete(cur->next, target)
    endif
  endif
endprocedure
```

Target = 4



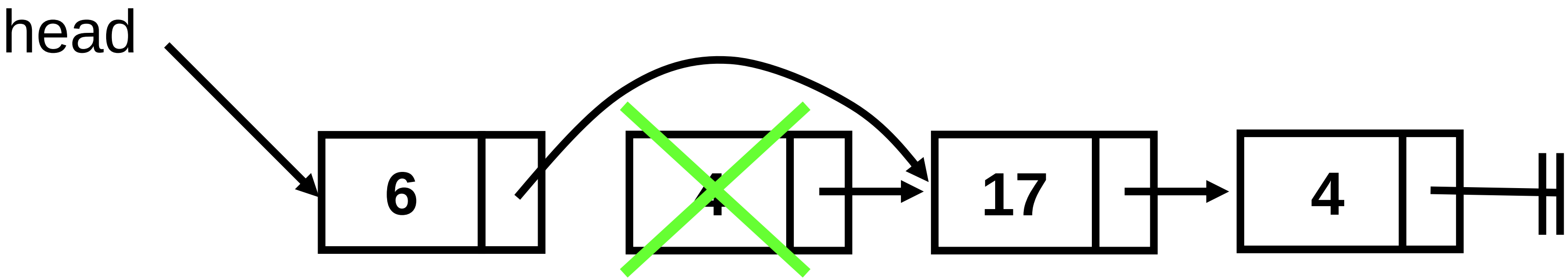
```
procedure Delete(cur iot in/out ptr toa Node,
           target isoftype in num)
// Delete all occurrences of a node.
  if(cur <> NULL) then
    if(cur -> data = target) then
   cur = cur->next
      Delete(cur, target)
    else
   Delete(cur->next, target)
    endif
  endif
endprocedure
```

Target = 4



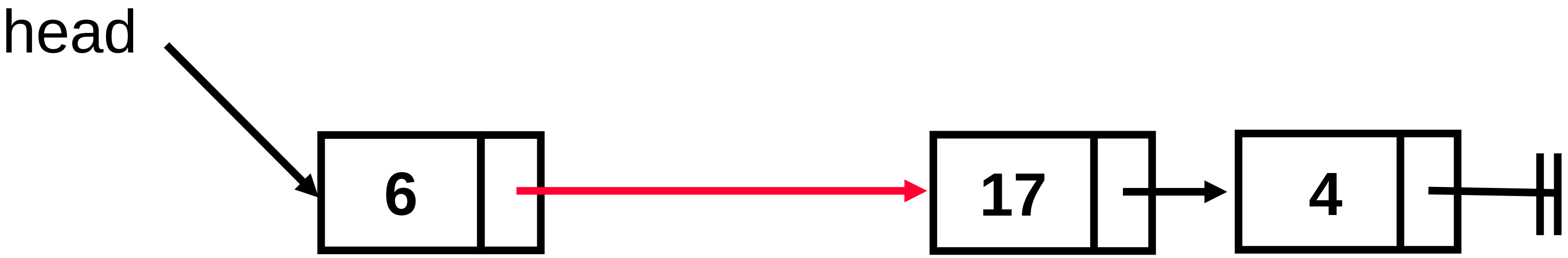
```
procedure Delete(cur iot in/out ptr toa Node,
           target isoftype in num)
// Delete all occurrences of a node.
  if(cur <> NULL) then
    if(cur -> data = target) then
   cur = cur->next
      Delete(cur, target)
    else
   Delete(cur->next, target)
    endif
  endif
endprocedure
```

Target = 4



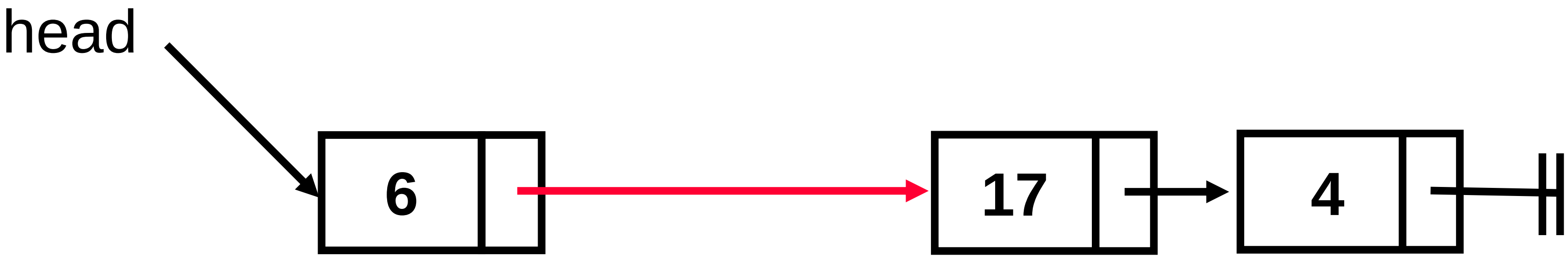
```
procedure Delete(cur iot in/out ptr toa Node,
           target isoftype in num)
// Delete all occurrences of a node.
  if(cur <> NULL) then
    if(cur -> data = target) then
   cur = cur->next
      Delete(cur, target)
    else
   Delete(cur->next, target)
    endif
  endif
endprocedure
```

Target = 4



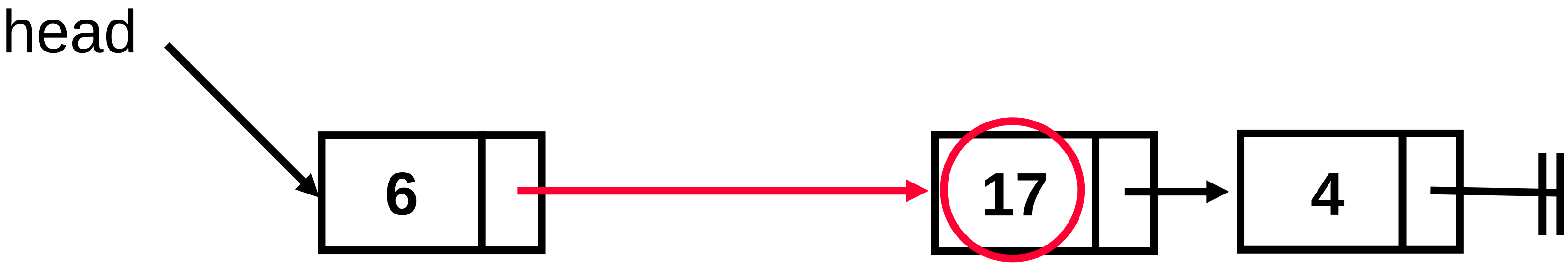
```
procedure Delete(cur iot in/out ptr toa Node,
           target isoftype in num)
// Delete all occurrences of a node.
  if(cur <> NULL) then
    if(cur -> data = target) then
   cur = cur->next
      Delete(cur, target)
    else
   Delete(cur->next, target)
    endif
  endif
endprocedure
```

Target = 4



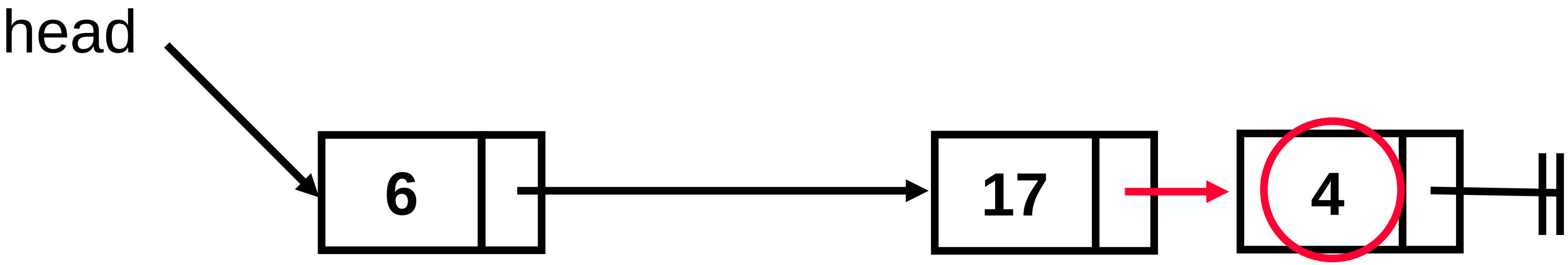
```
procedure Delete(cur iot in/out ptr toa Node,
           target isoftype in num)
// Delete all occurrences of a node.
  if(cur <> NULL) then
    if(cur -> data = target) then
   cur = cur->next
      Delete(cur, target)
    else
   Delete(cur->next, target)
    endif
  endif
endprocedure
```

Target = 4



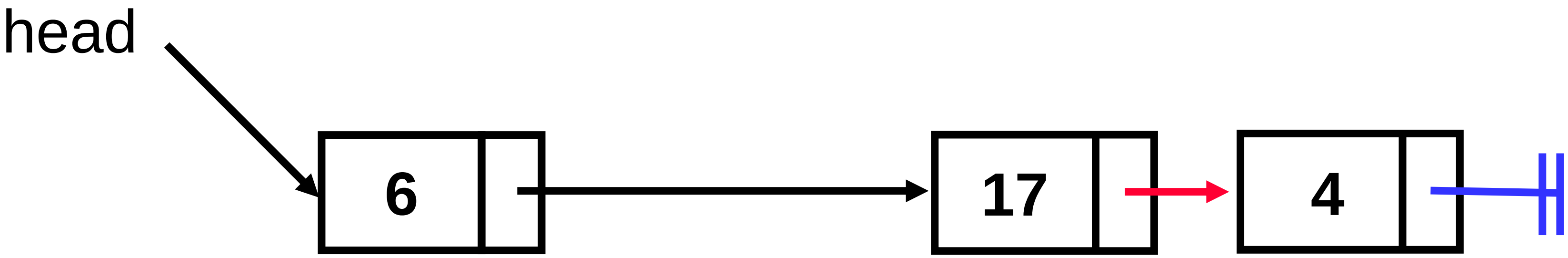
```
procedure Delete(cur iot in/out ptr toa Node,
           target isoftype in num)
// Delete all occurrences of a node.
  if(cur <> NULL) then
    if(cur -> data = target) then
   cur = cur->next
      Delete(cur, target)
    else
   Delete(cur->next, target)
    endif
  endif
endprocedure
```

Target = 4



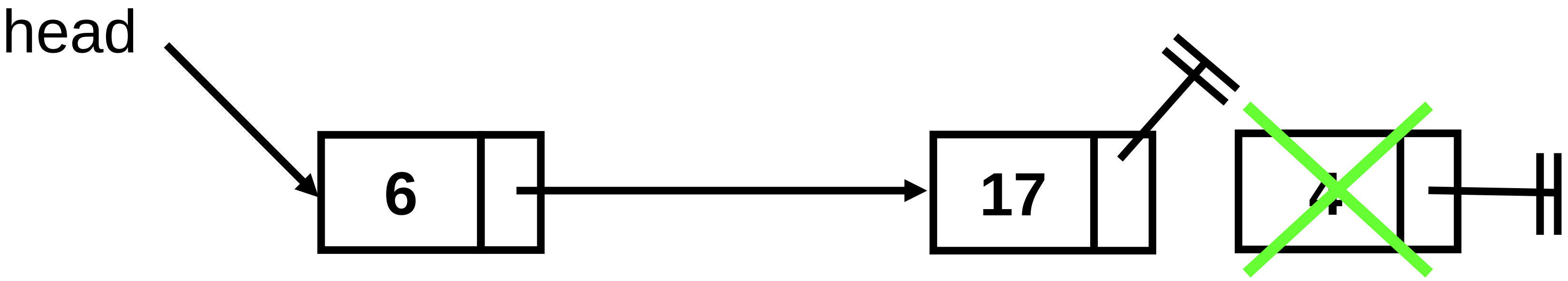
```
procedure Delete(cur iot in/out ptr toa Node,
           target isoftype in num)
// Delete all occurrences of a node.
  if(cur <> NULL) then
    if(cur -> data = target) then
   cur = cur->next
      Delete(cur, target)
    else
   Delete(cur->next, target)
    endif
  endif
endprocedure
```

Target = 4



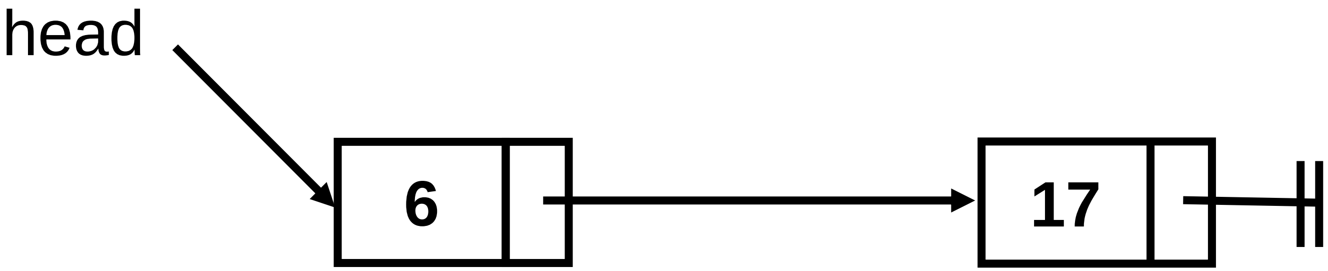
```
procedure Delete(cur iot in/out ptr toa Node,
           target isoftype in num)
// Delete all occurrences of a node.
  if(cur <> NULL) then
    if(cur -> data = target) then
   cur = cur->next
      Delete(cur, target)
    else
   Delete(cur->next, target)
    endif
  endif
endprocedure
```

Target = 4



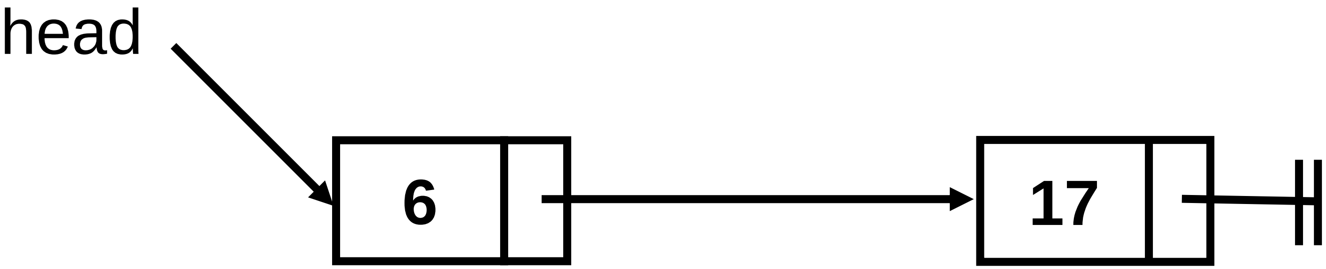
```
procedure Delete(cur iot in/out ptr toa Node,
           target isoftype in num)
// Delete all occurrences of a node.
  if(cur <> NULL) then
    if(cur -> data = target) then
   cur = cur->next
      Delete(cur, target)
    else
   Delete(cur->next, target)
    endif
  endif
endprocedure
```

Target = 4



```
procedure Delete(cur iot in/out ptr toa Node,
           target isoftype in num)
// Delete all occurrences of a node.
  if(cur <> NULL) then
    if(cur -> data = target) then
   cur = cur->next
      Delete(cur, target)
    else
   Delete(cur->next, target)
    endif
  endif
endprocedure
```

Target = 4



```
.
.
Delete(head, 4)
.
.
```

Summary

Summary

The basic operations of linked-list

- traverse, insert, delete

Location of linked lists (traverse/insert/ delete)

- Front, End, Somewhere in the middle (to preserve order)

Types of linked lists

- Singly linked list
- Circular, singly linked
- Doubly linked list
- Circular, doubly linked list

Question

This Photo by Unknown Author is licensed under CC BY-NC