List

- A list is a linear data structure with homogeneous components (called list items or list elements) that can only be accessed sequentially, one after the other.

- We say the first item in the list is at the head or front of the list, and the last item in the list is at the tail of the list.

To implement the List ADT

The programmer must

1) choose a concrete data representation for the list, and

2) implement the list operations.

How to implement a list?

- **Use a built-in array stored in contiguous memory locations, implementing operations by using [ ] and moving list items around in the array, as needed for insertions and deletions.**

- **Use a linked list (to avoid excessive data movement from insertions and deletions) not necessarily stored in contiguous memory locations, but rather on the heap or free store.**

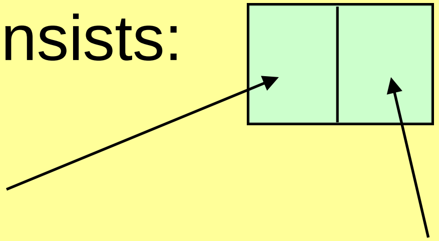
A Linked List

- **A linked list is a collection of nodes.**

- **The nodes are `struct`s (or `class` objects).**

- **Each node contains at least one member (field) that gives the location of the next node in the list.**

Each node consists:

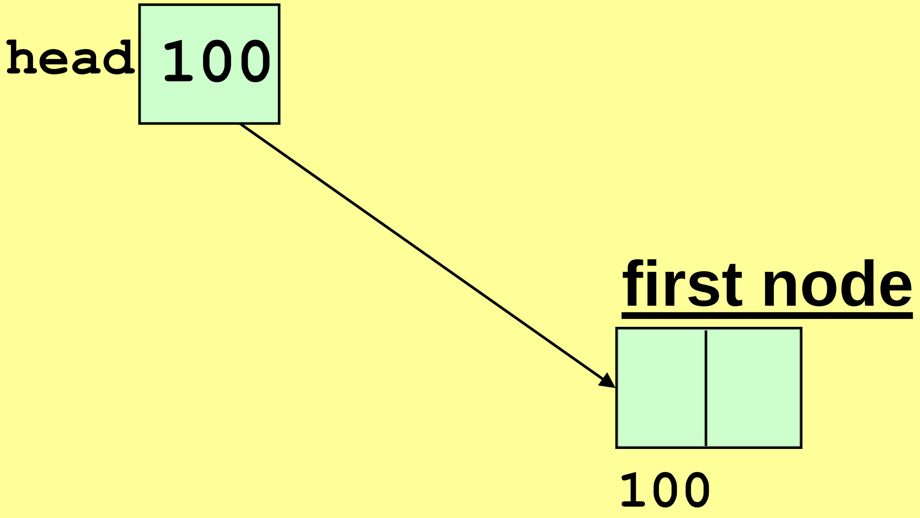


- A data item
- Link Member holds an **address of another node**

A Linked List

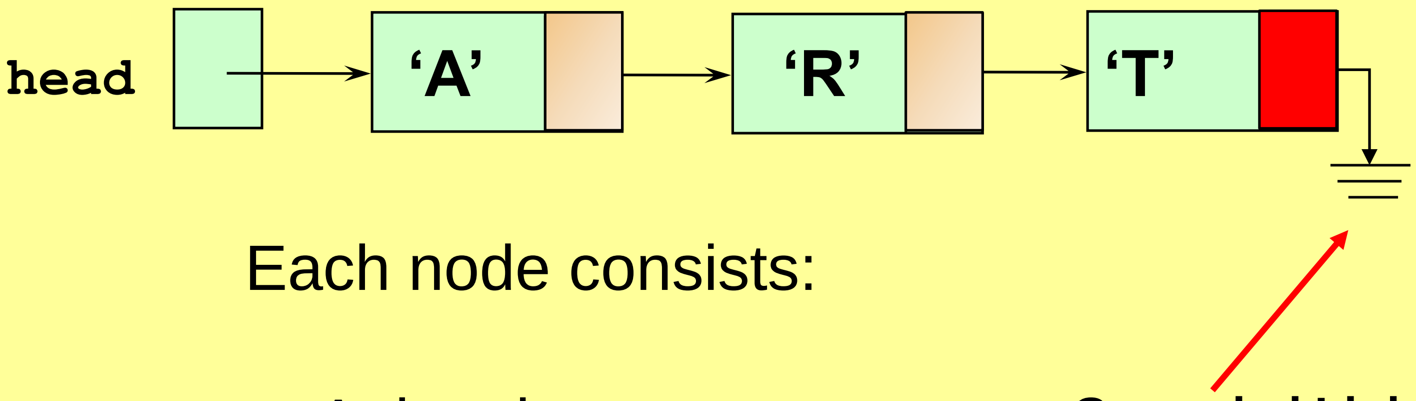
- A linked list is a collection of nodes and an external pointer to the very first node.

```
struct node *head;
```



Singly Linked List

- **In a singly linked list, each node contains only one link member.**



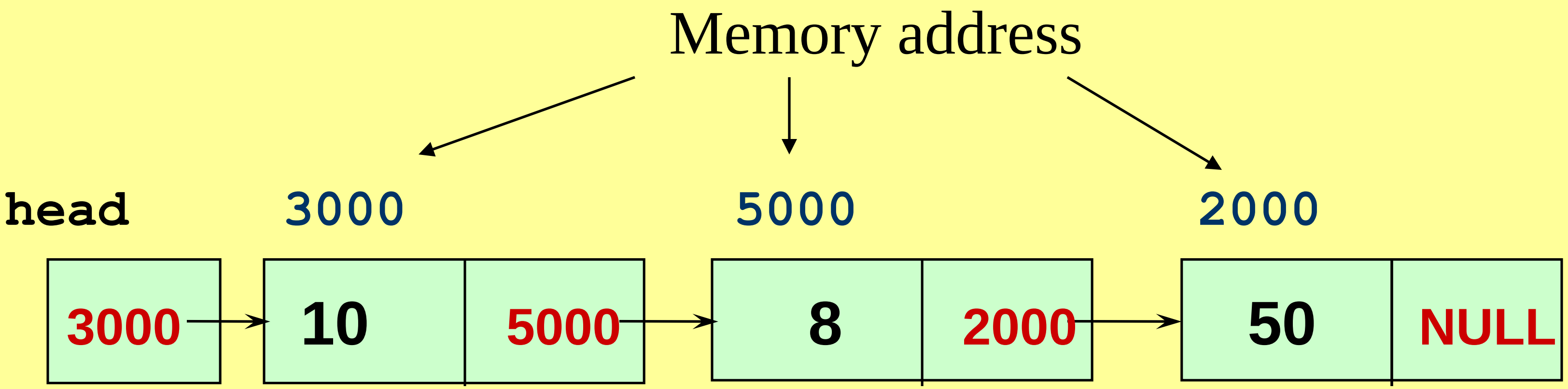
Each node consists:

- A data item
- Link Member

Grounded Link List
The last node link member contains NULL

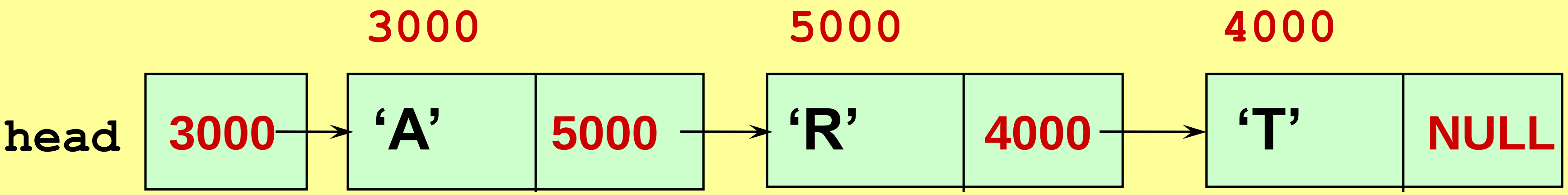
Nodes can be located anywhere in memory

the link member holds the memory address of the next node in the list



Nodes can be located anywhere in memory

- **The <u>link member</u> holds the memory address of the next node in the list.**



Each node consists:

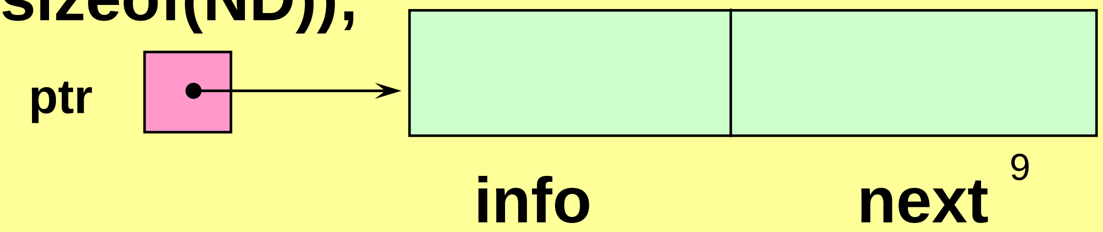
- A data item
- Link Member holds an **address of another node**

Declarations for a Singly Linked List

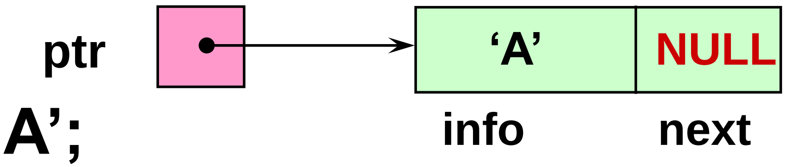
```
// Type DECLARATIONS

struct Node  {
   char         info;
   struct Node    *next;
};

typedef  struct Node  ND;

// Variable DECLARATIONS

ND   *head;
ND   *ptr;
```

Pointer variables of type structure Node

ptr = (ND *) malloc (sizeof(ND));

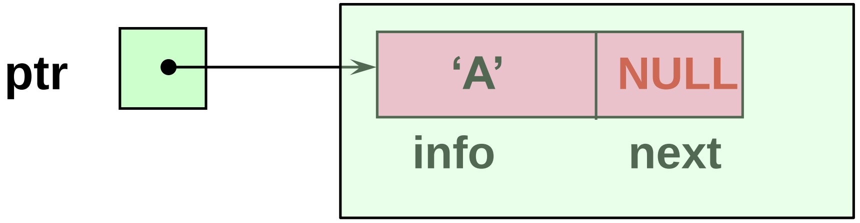


Pointer Dereferencing and Member Selection

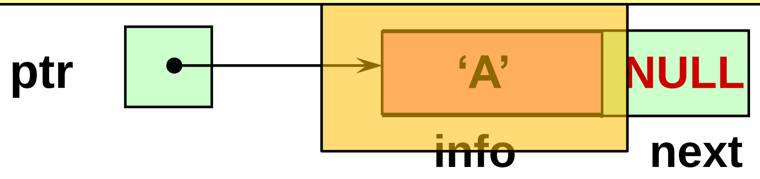


ptr ->info = 'A';

ptr->next=NULL;

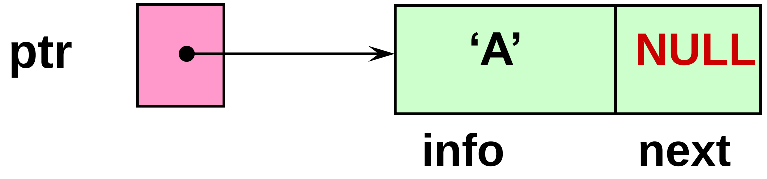


*ptr



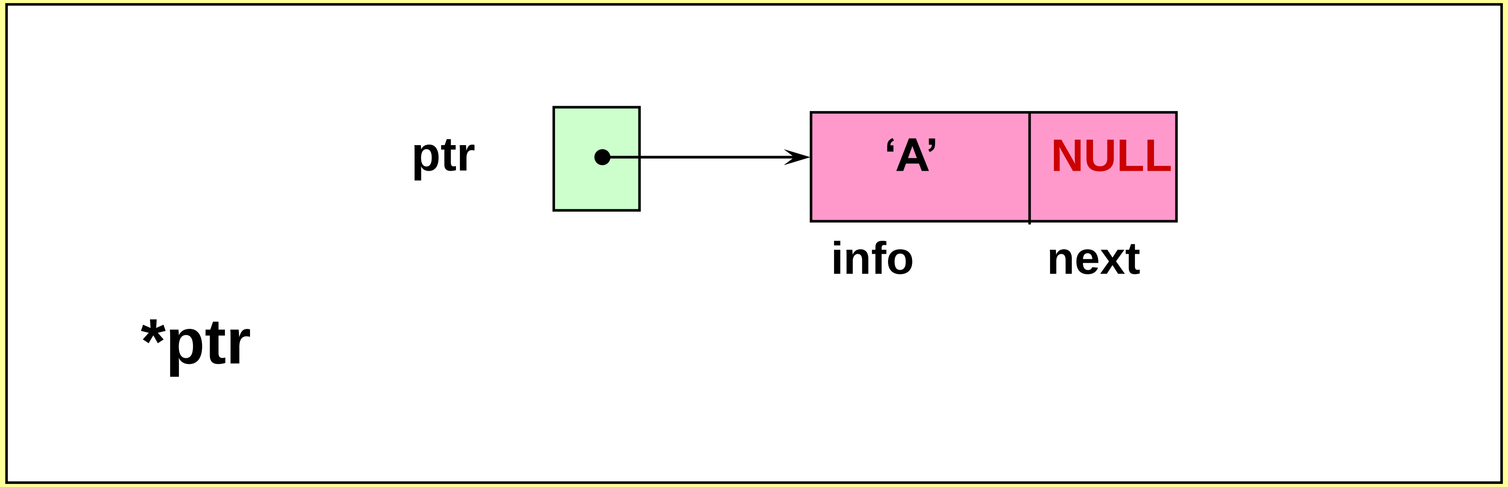
ptr->info

`ptr` is a pointer to a node

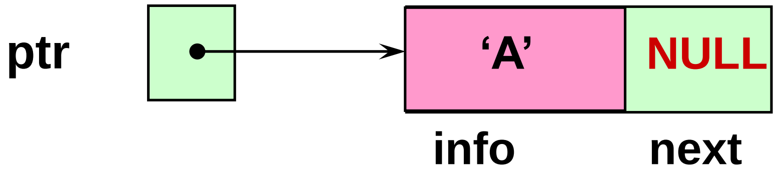


ptr

`*ptr` is the node pointed to by ptr

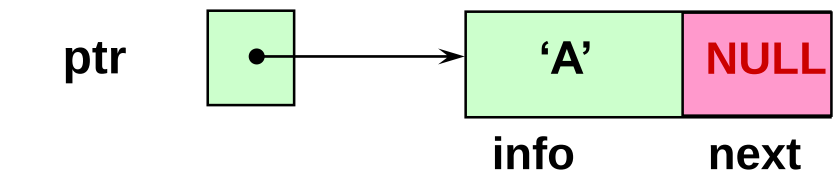


`ptr->info`
is a node member



ptr->info

`ptr->next` is a node member



ptr->next

Operations on Linked Lists

- Insertion: Adding a new node to a linked list involves adjusting the pointers of the existing nodes to maintain the proper sequence. **Insertion can be performed at the beginning, end, or any position within the list**
- Deletion: Removing a node from a linked list requires adjusting the pointers of the neighboring nodes to bridge the gap left by the deleted node. **Deletion can be performed at the beginning, end, or any position within the list.**
- Searching: Searching for a specific value in a linked list involves traversing the list from the head node until the value is found or the end of the list is reached.

Linked Lists

Advantages of Linked Lists

- **Dynamic Size:** Linked lists can grow or shrink dynamically, as memory allocation is done at runtime.
- **Insertion and Deletion:** Adding or removing elements from a linked list is efficient, especially for large lists.
- **Flexibility:** Linked lists can be easily reorganized and modified without requiring a contiguous block of memory.

Disadvantages of Linked Lists

- **Random Access:** Unlike arrays, linked lists do not allow direct access to elements by index. Traversal is required to reach a specific node.
- **Extra Memory:** Linked lists require additional memory for storing the pointers, compared to arrays.

Types of linked lists

- There are Four main types of linked lists in C:

1. Singly Linked Lists.
   - Example: Stack, queue, linked list implementation of a dynamic array
2. Doubly Linked Lists.
   - Example: LRU cache, undo/redo history, doubly linked list implementation of a binary tree
3. Circular Linked Lists
   - Example: Circular buffer, circular queue, circular linked list implementation of a hash table.
4. Header Linked List
   - The header linked lists are frequently used to maintain the polynomials in memory. The *header node* is used to represent the *zero polynomial*.
   - E.g: **F(x) = $5x^5 – 3x^3 + 2x^2 + x^1 +10x^0$**

Types of linked lists

1. **Singly Linked Lists (Grounded).**

- Singly linked lists in C are the simplest type of linked list.
- Each node in a singly linked list contains a data field and a pointer to the next node in the list.
- The last node in the list points to null, indicating the end of the list.
- Example: Stack, queue, linked list implementation of a dynamic array

Types of linked lists

2. **Doubly Linked Lists.**

- Doubly linked lists in C are more complex to implement than singly linked lists, but they are more efficient for certain operations, such as insertion, deletion, and traversal in both directions.
- Each node in a doubly linked list contains a data field, a pointer to the next node in the list, and a pointer to the previous node in the list.
- Example: LRU cache, undo/redo history, doubly linked list implementation of a binary tree

Types of linked lists

3. **Circular Linked Lists**

- Circular linked lists are the most complex type of linked list to implement, but they can be very efficient for certain operations, such as traversal and queueing.
- In a circular linked list, the last node in the list points back to the first node in the list, forming a loop.
- Example: Circular buffer, circular queue, circular linked list implementation of a hash table.

Types of linked lists

4. Header Linked List

- A header linked list is a type of linked list that uses a special header node to represent the beginning of the list.
- A ***header node*** is a special node that is found at the beginning of the list.
- For example, suppose there is an application in which the number of items in a list is often calculated. Usually, a list is always traversed to find the length of the list. However, if the current length is maintained in an additional header node that information can be easily obtained.

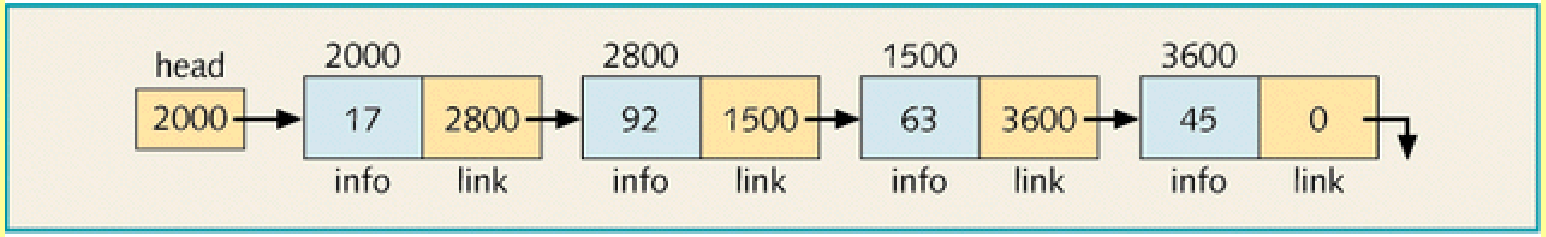
Types of Header Linked List

- Grounded Header Linked List:
  - It is a list whose last node contains the NULL pointer.
  - In the header linked list the start pointer always points to the header node. start -> next = NULL indicates that the grounded header linked list is empty.
  - The operations that are possible on this type of linked list are Insertion, Deletion, and Traversing.

Types of Header Linked List

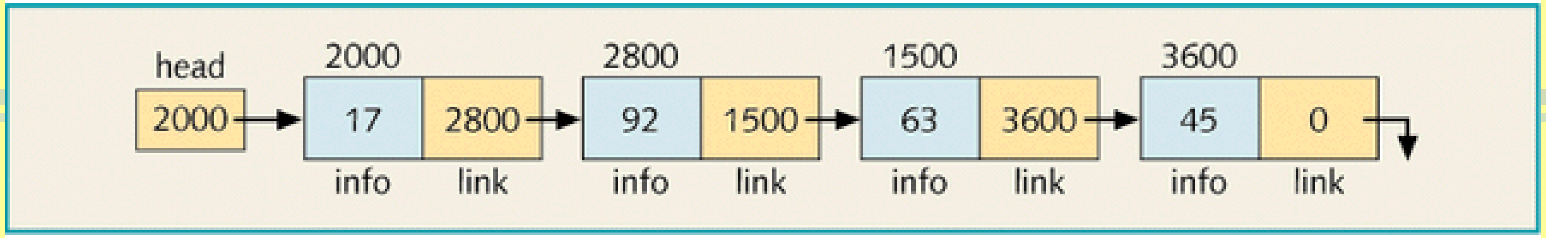
- Circular Header Linked List
  - A list in which last node points back to the header node is called circular linked list.
  - The chains do not indicate first or last nodes. In this case, external pointers provide a frame of reference because last node of a circular linked list does not contain the NULL pointer.
  - The possible operations on this type of linked list are Insertion, Deletion and Traversing.

Linked List Properties

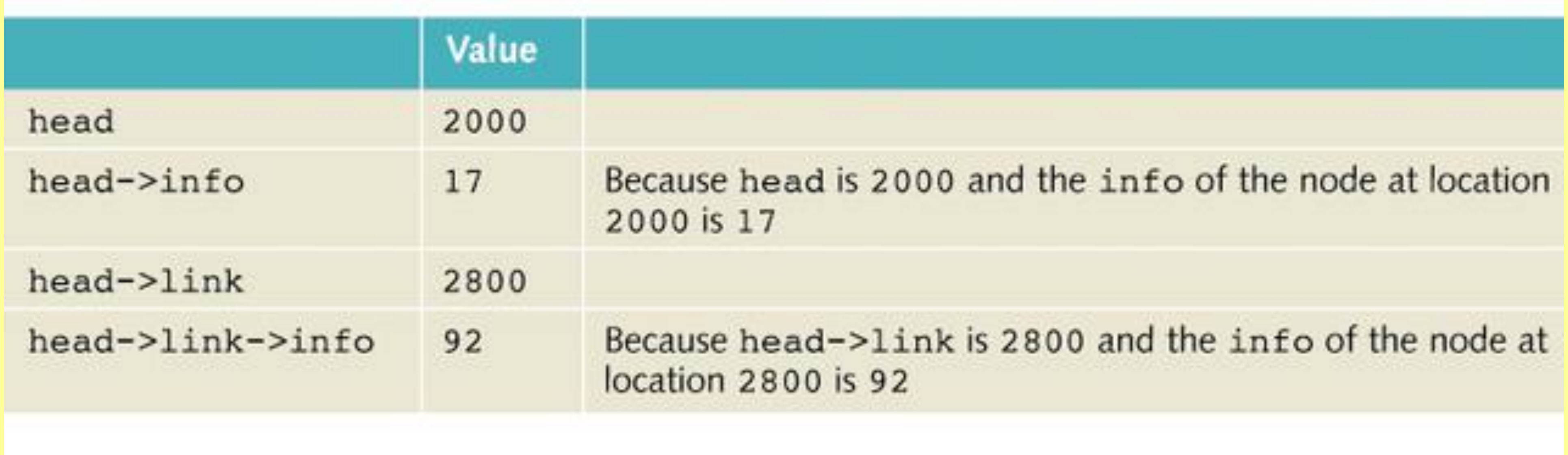


Linked list with four nodes

- This linked list has four nodes
- The address of the first node is stored in the pointer head
- Each node has two components: a component, info, to store the info and another component, link, to store the address of the next node



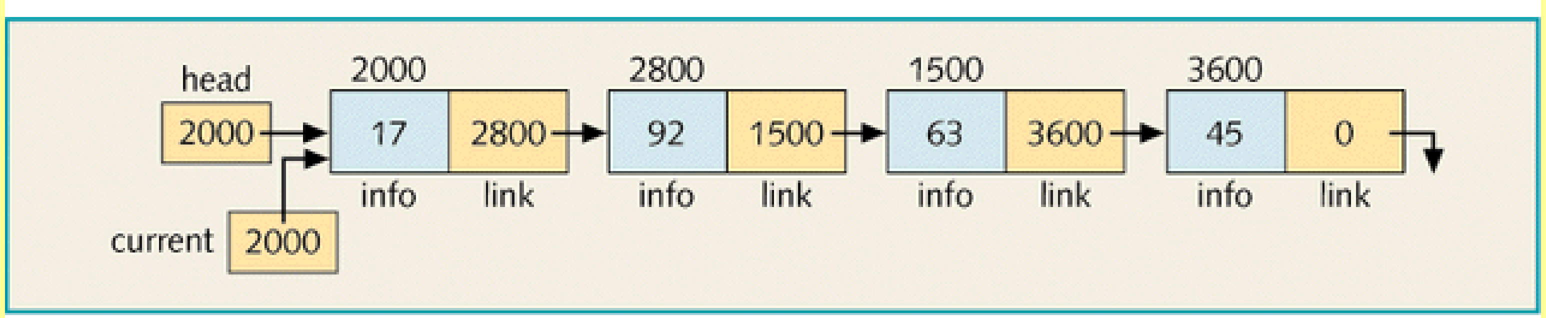
Linked list with four nodes



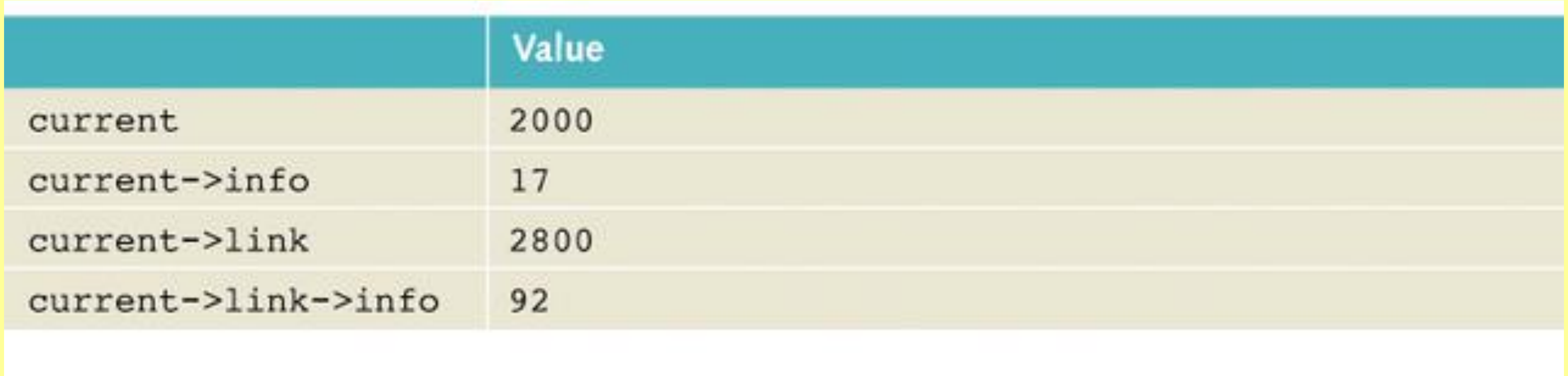
	Value	
`head`	`2000`	
`head->info`	`17`	Because `head` is `2000` and the `info` of the node at location `2000` is `17`
`head->link`	`2800`	
`head->link->info`	`92`	Because `head->link` is `2800` and the `info` of the node at location `2800` is `92`

Linked List Traversing

- current = head;

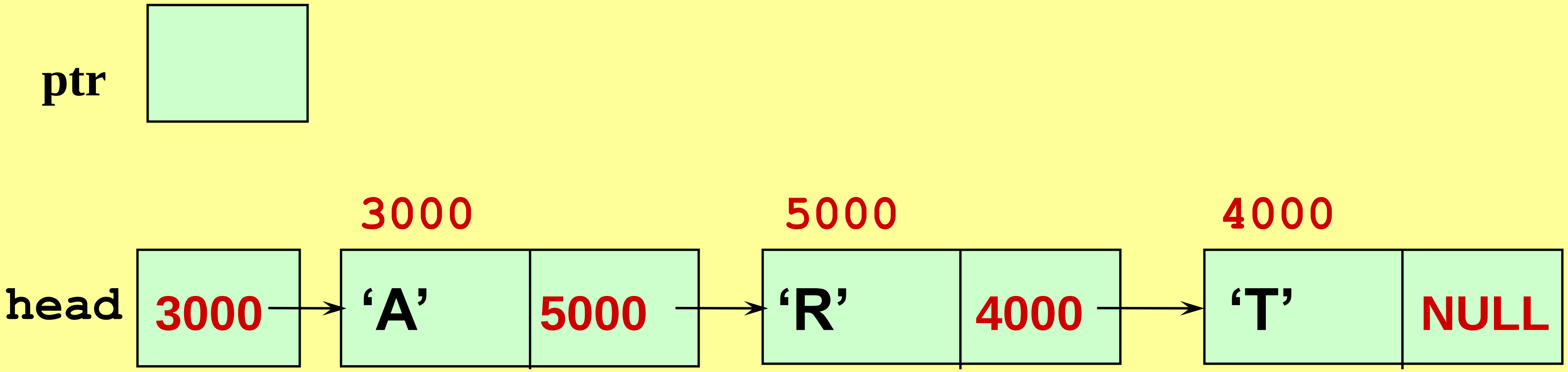


Linked list after current = head; executes



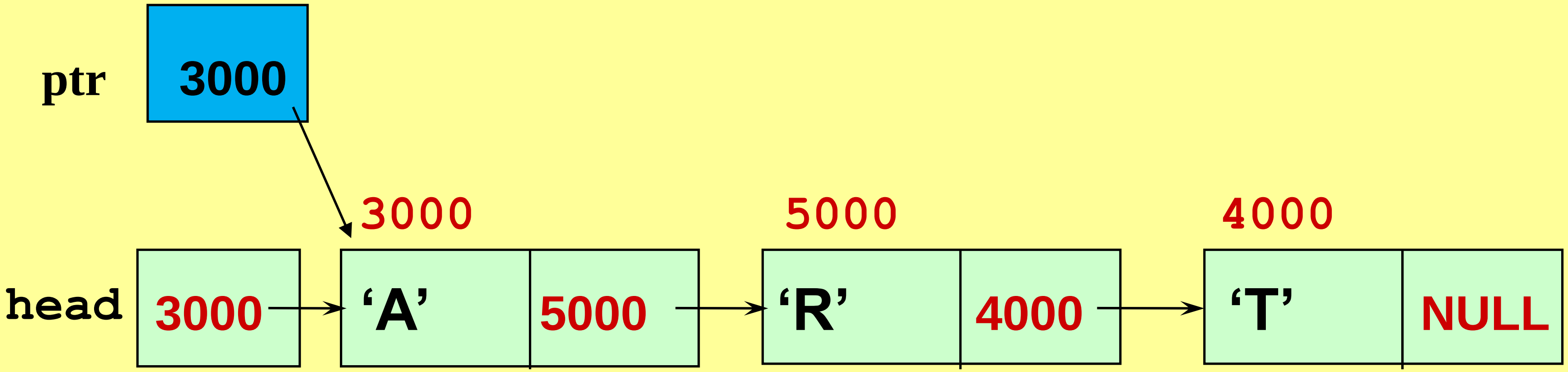
	Value
current	2000
current->info	17
current->link	2800
current->link->info	92

Traversing a Singly Linked List



```
//PRE:  head points to a singly linked list

ptr  =  head ;
while (ptr != NULL)  {
     printf("%c\n",ptr->info) ;
                          // Or, do something else with node *ptr
     ptr  =  ptr->next ;
}
```

Traversing a Singly Linked List



ptr 3000

head 3000

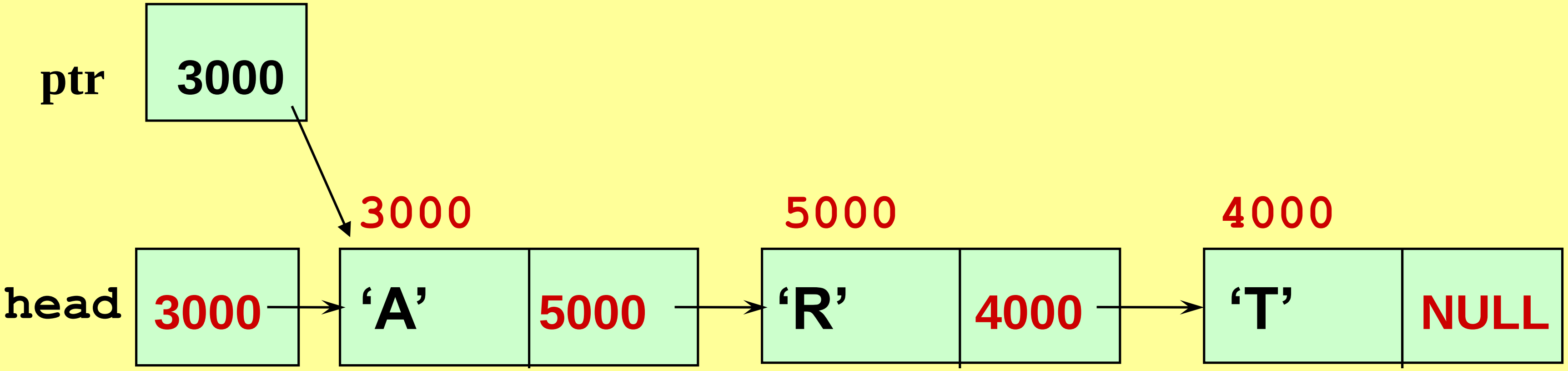
3000 ‘A’ 5000

5000 ‘R’ 4000

4000 ‘T’ NULL

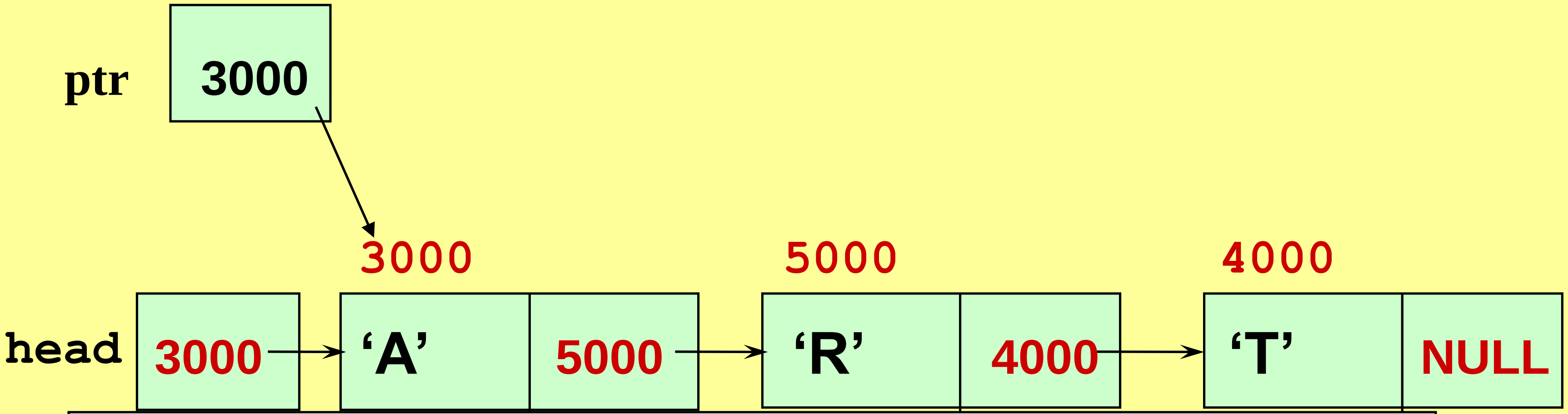
```
//PRE:  head points to a singly linked list

ptr  =  head ;
while (ptr != NULL)  {
      printf(“%c\n”,ptr->info) ;
                                // Or, do something else with node *ptr
      ptr  =  ptr->next ;
}
```

Traversing a Singly Linked List



```
//PRE:  head points to a singly linked list

ptr  =  head ;
while (ptr != NULL)  {
     printf(“%c\n”,ptr->info) ;
                          // Or, do something else with node *ptr
     ptr  =  ptr->next ;
}
```

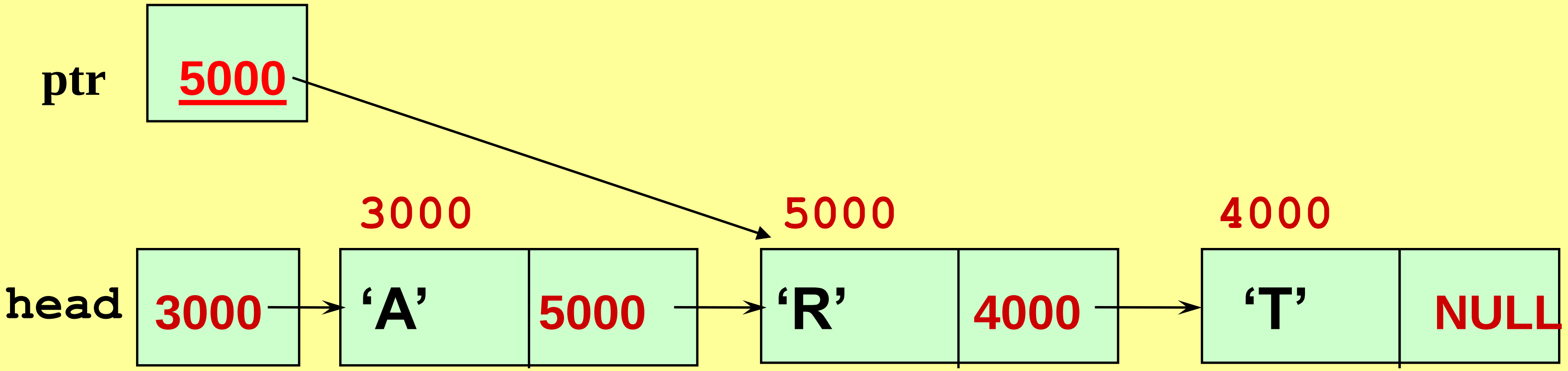
Traversing a Singly Linked List



```
//PRE:  head points to a singly linked list

ptr  =  head ;
while (ptr != NULL)  {
      printf(“%c\n”,ptr->info);
                              // Or, do something else with node *ptr
      ptr  =  ptr->next ;
}
```

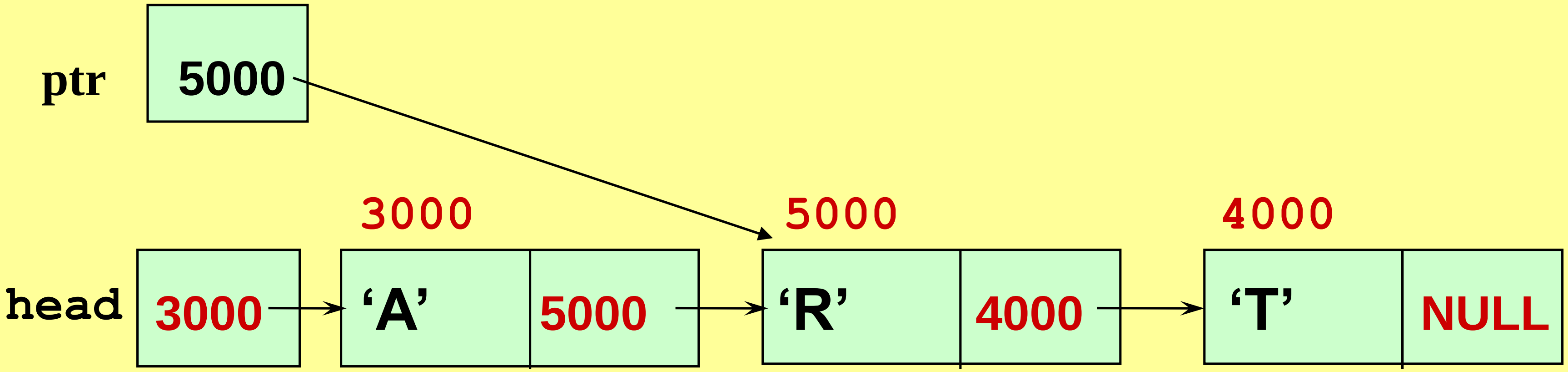
A

Traversing a Singly Linked List



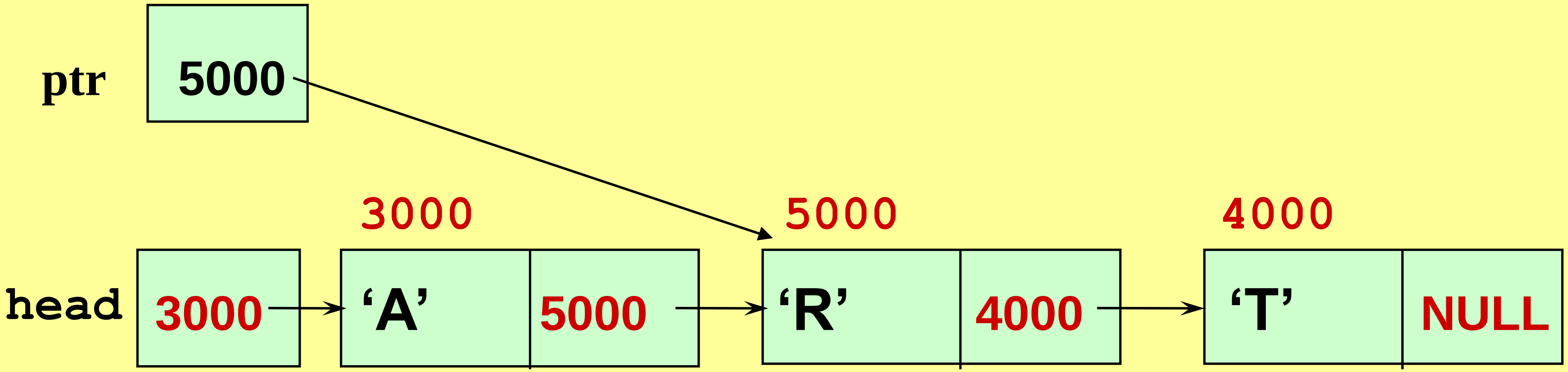
```
//PRE:  head points to a singly linked list

ptr  =  head ;
while (ptr != NULL)  {
      printf("%c\n",ptr->info);
                        // Or, do something else with node *ptr
    ptr  =  ptr->next ;
}
```

Traversing a Singly Linked List



```
//PRE:  head points to a singly linked list

ptr  =  head ;
while (ptr != NULL)  {
      printf("%c\n",ptr->info);
                          // Or, do something else with node *ptr
     ptr  =  ptr->next ;
}
```

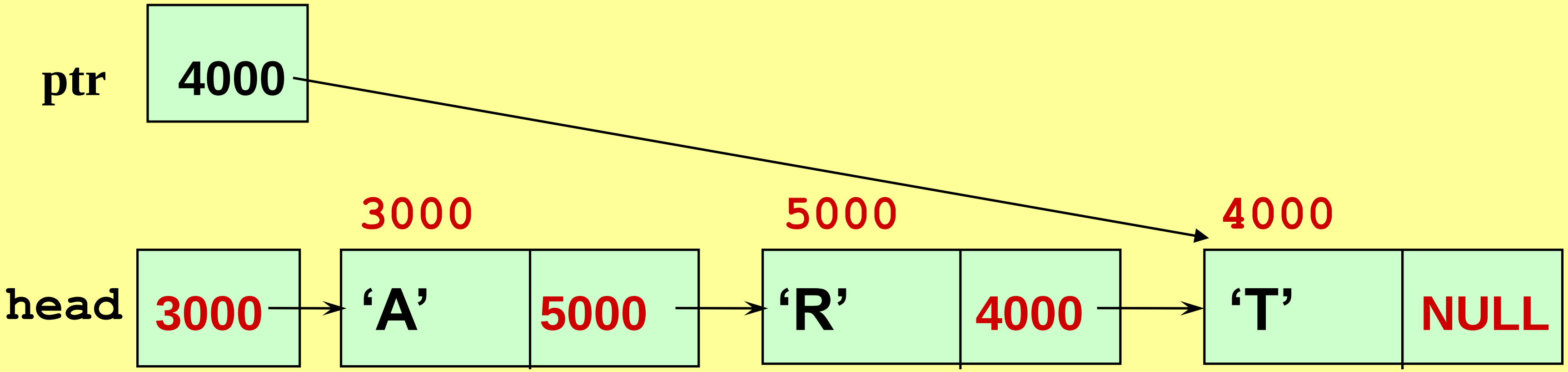
Traversing a Singly Linked List



```
//PRE:  head points to a singly linked list

ptr  =  head ;
while (ptr != NULL)  {
      printf(“%c\n”,ptr->info);
                              // Or, do something else with node *ptr
      ptr  =  ptr->next ;
}
```

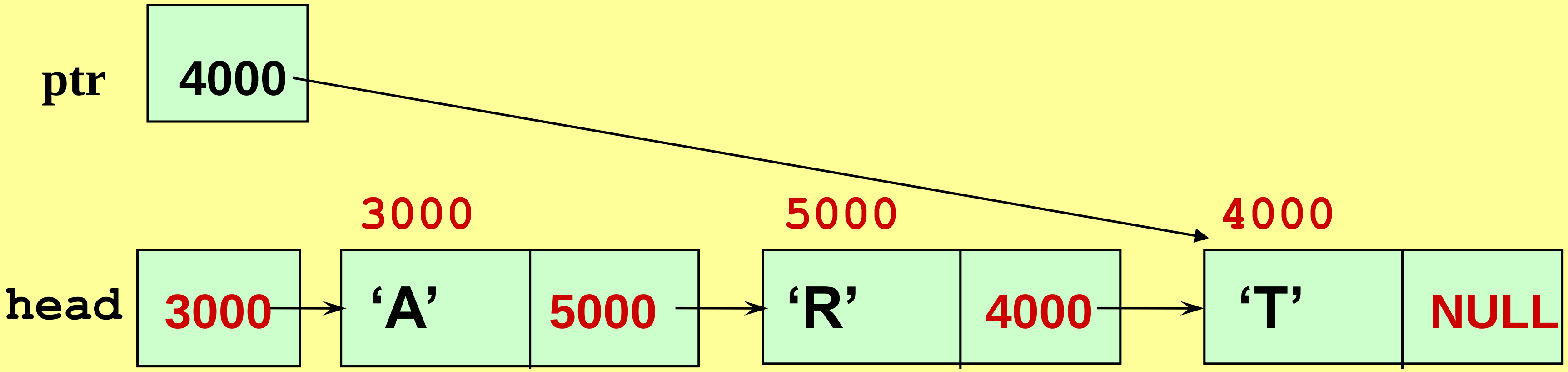
R

Traversing a Singly Linked List



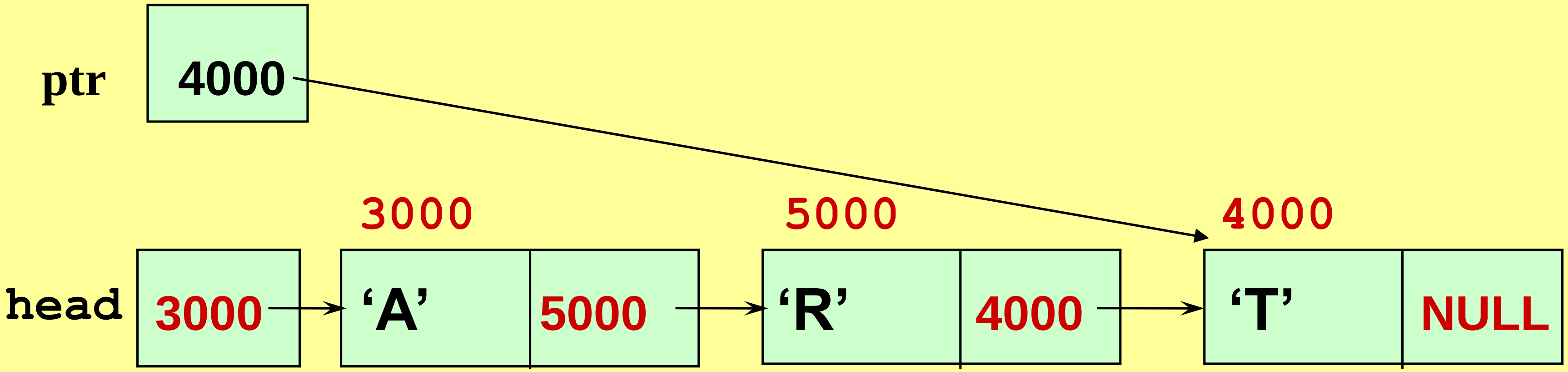
```
//PRE:  head points to a singly linked list

ptr  =  head ;
while (ptr != NULL)  {
      printf("%c\n",ptr->info);
                        // Or, do something else with node *ptr
     ptr  =  ptr->next ;
}
```

Traversing a Singly Linked List



```
//PRE:  head points to a singly linked list

ptr  =  head ;
while (ptr != NULL)  {
        printf(“%c\n”,ptr->info);
                            // Or, do something else with node *ptr
       ptr  =  ptr->next ;
}
```

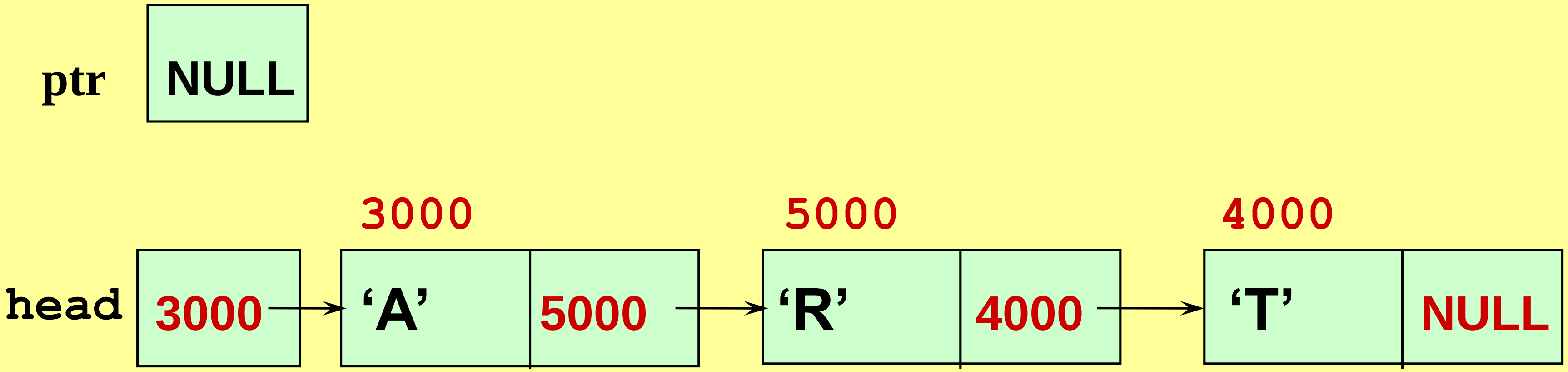
Traversing a Singly Linked List



```
//PRE:  head points to a singly linked list

ptr  =  head ;
while (ptr != NULL)  {
      printf("%c\n",ptr->info);
                              // Or, do something else with node *ptr
      ptr  =  ptr->next ;
}
```

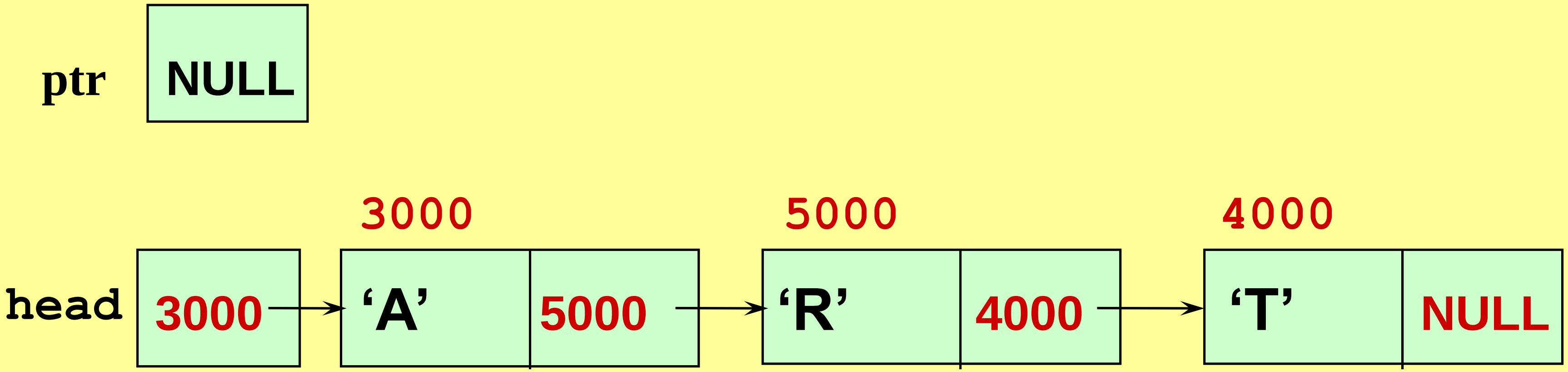
T

Traversing a Singly Linked List



```
//PRE:  head points to a singly linked list

ptr  =  head ;
while (ptr != NULL)  {
      printf("%c\n",ptr->info);
                         // Or, do something else with node *ptr
    ptr  =  ptr->next ;
}
```

Traversing a Singly Linked List

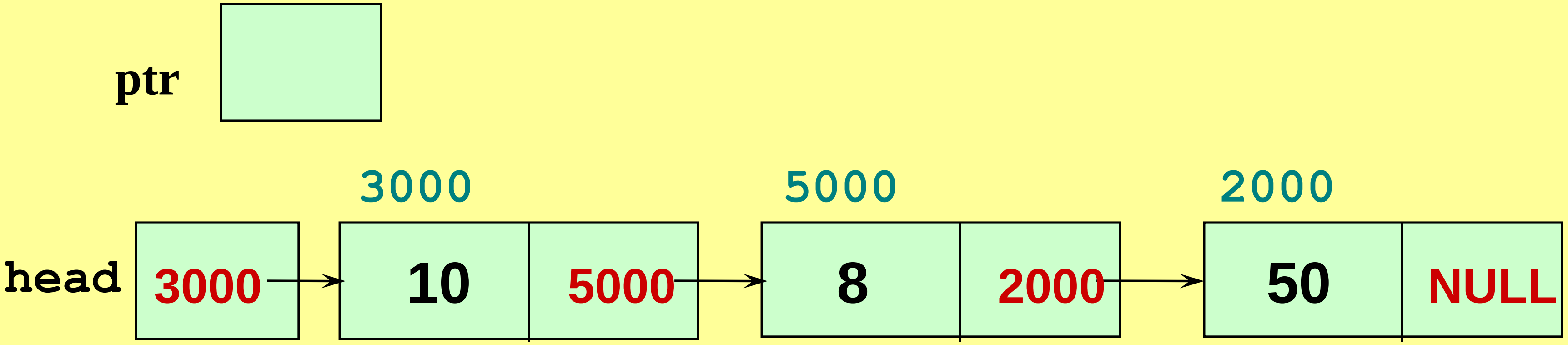


```
//PRE:  head points to a singly linked list

ptr  =  head ;
while (ptr != NULL)  {
      printf("%c\n",ptr->info);
                        // Or, do something else with node *ptr
     ptr  =  ptr->next ;
}
```

Declarations for a Linked List

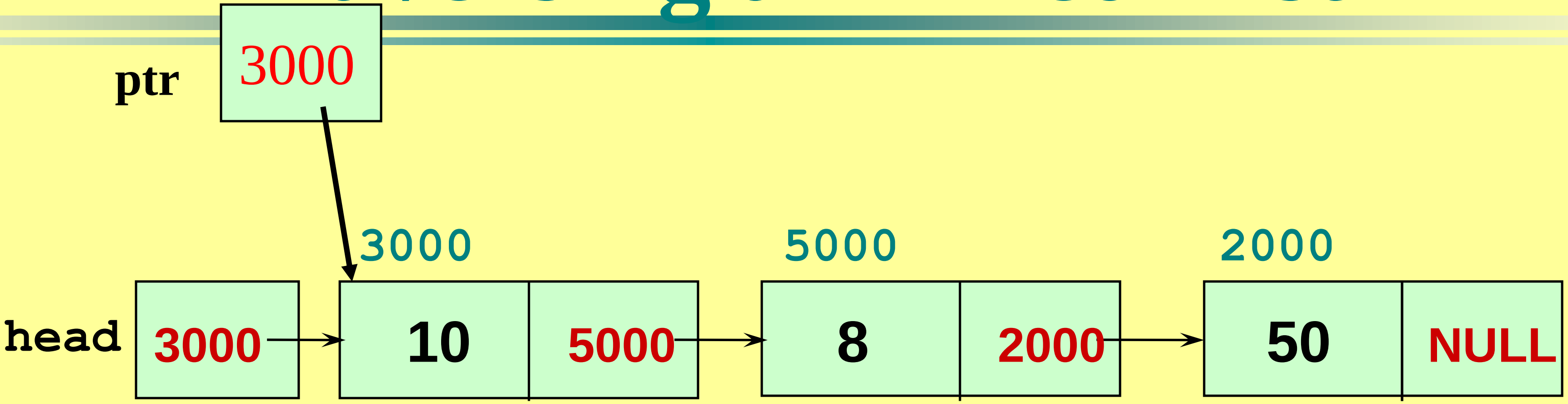
```
// Type DECLARATIONS
struct NodeType  {
  int          info;
  struct NodeType *next;
};

// Variable DECLARATIONS
struct NodeType *head;
struct NodeType *ptr;
```

Traversing a Linked List



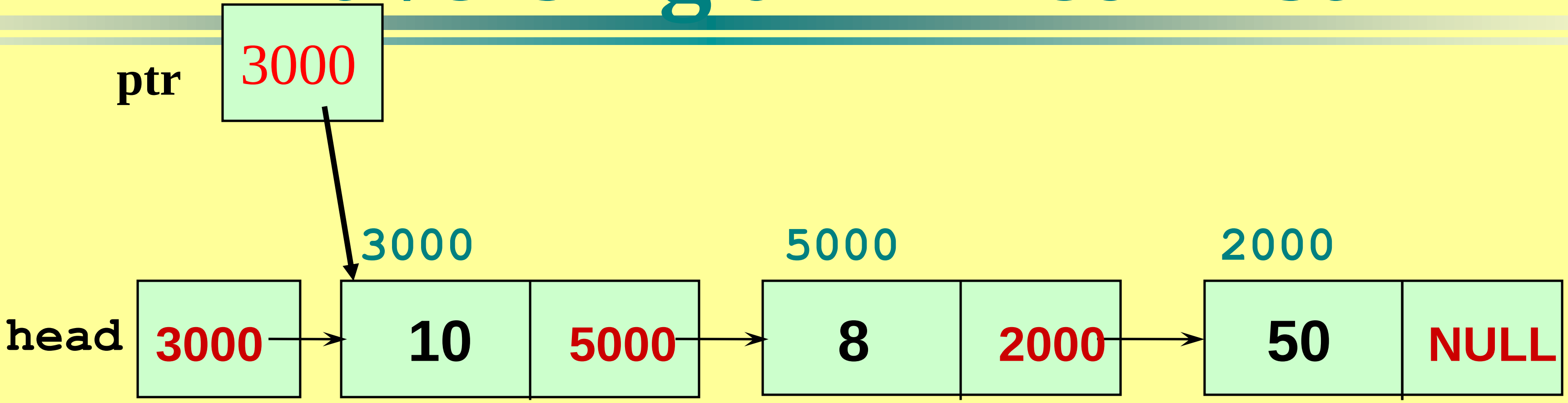
```
//PRE:  head points to a linked list
node *ptr ;

ptr  =  head ;
while (ptr != NULL)  {
       printf("%d\n",ptr->info) ;
      // Or, do something else
ptr  =  ptr->next ;
}
```

Traversing a Linked List



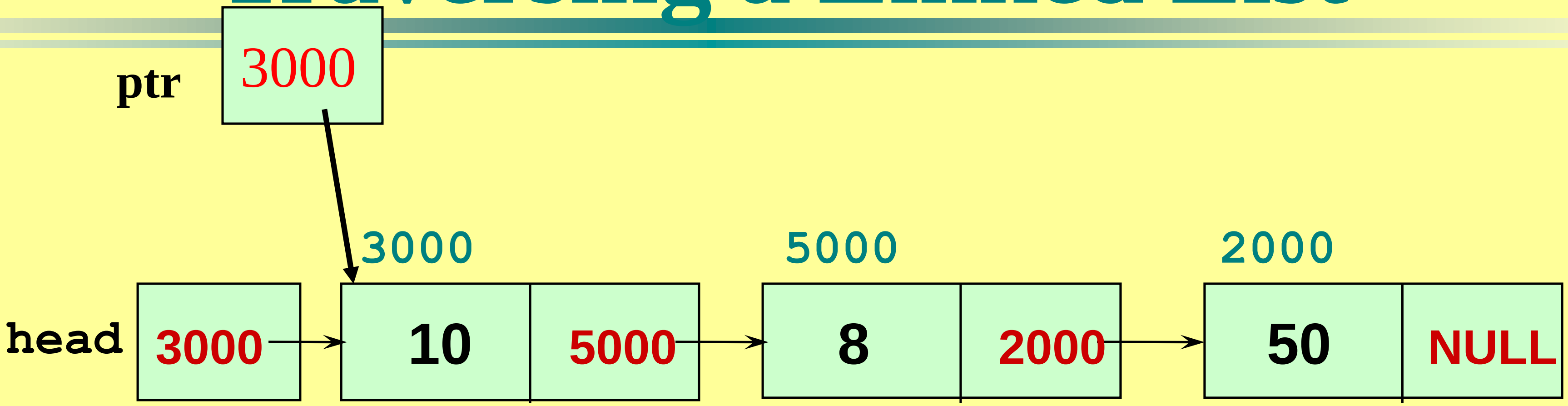
```
//PRE:  head points to a linked list
node *ptr ;

ptr  =  head ;
while (ptr != NULL)  {
       printf("%d\n",ptr->info) ;
      // Or, do something else
ptr  =  ptr->next ;
}
```

Traversing a Linked List



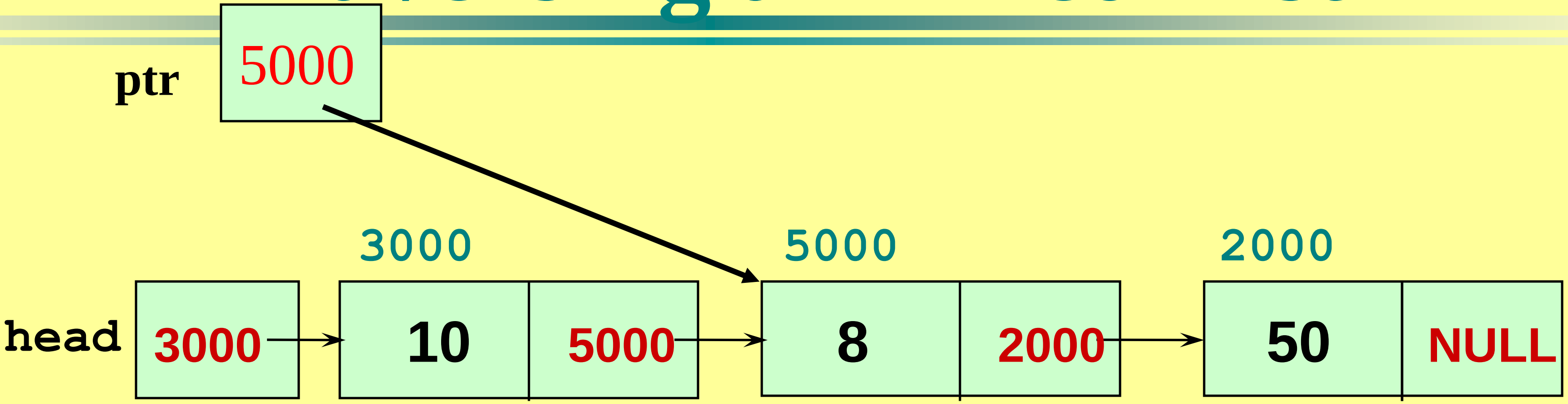
```
//PRE:  head points to a linked list
node *ptr ;

ptr  =  head ;
while (ptr != NULL)  {
        printf("%d\n",ptr->info) ;
        // Or, do something else
ptr  =  ptr->next ;
}
```

Traversing a Linked List



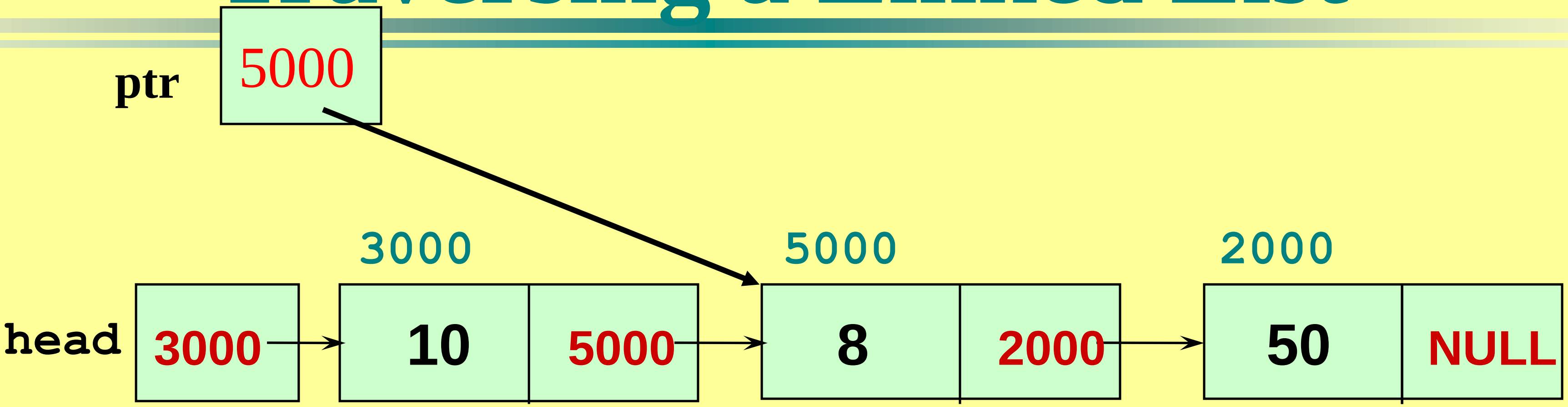
```
//PRE:  head points to a linked list
node *ptr ;

ptr  =  head ;
while (ptr != NULL)  {
      printf("%d\n",ptr->info) ;
      // Or, do something else
ptr  =  ptr->next ;
}
```

Traversing a Linked List



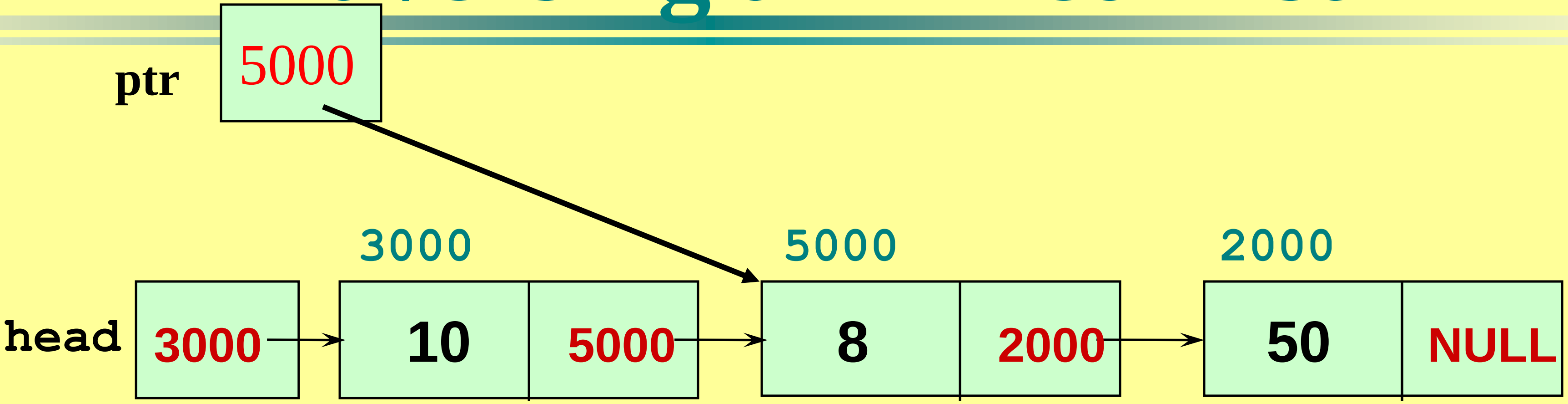
```
//PRE:  head points to a linked list
node *ptr ;

ptr  =  head ;
while (ptr != NULL)  {
       printf(“%d\n”,ptr->info) ;
      // Or, do something else
ptr  =  ptr->next ;
}
```

Traversing a Linked List



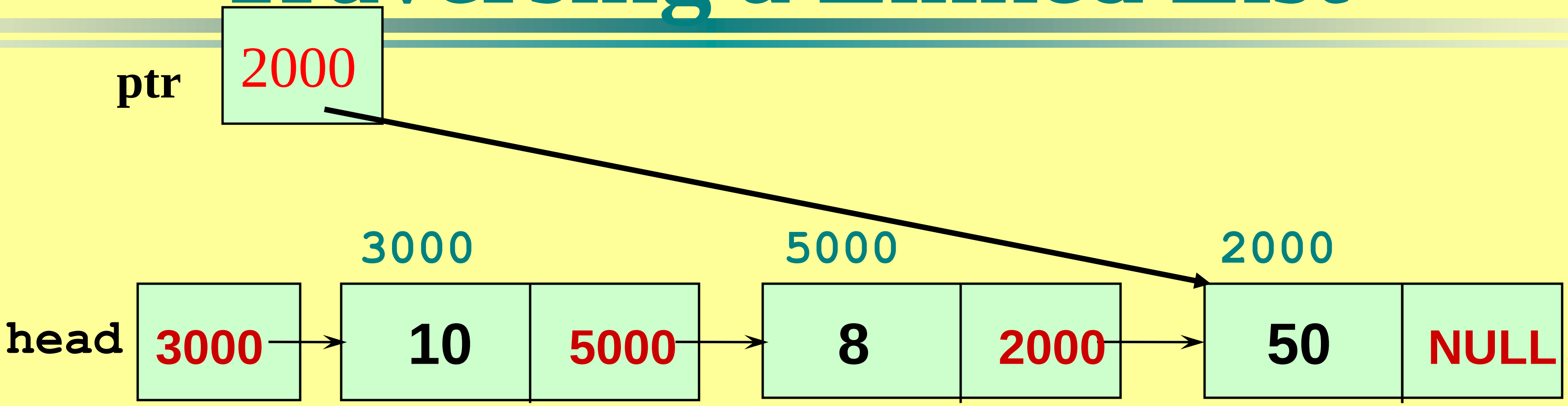
```
//PRE:  head points to a linked list
node *ptr ;

ptr  =  head ;
while (ptr != NULL)  {
        printf("%d\n",ptr->info) ;
       // Or, do something else
ptr  =  ptr->next ;
}
```

Traversing a Linked List



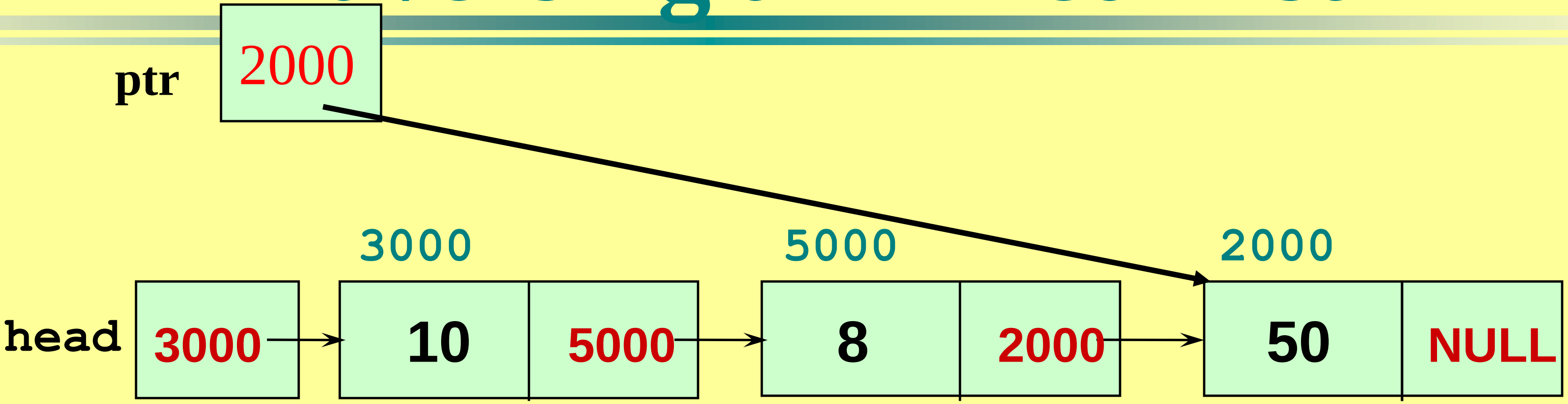
```
//PRE:  head points to a linked list
node *ptr ;

ptr  =  head ;
while (ptr != NULL)  {
        printf("%d\n",ptr->info) ;
       // Or, do something else
ptr  =  ptr->next ;
}
```

Traversing a Linked List



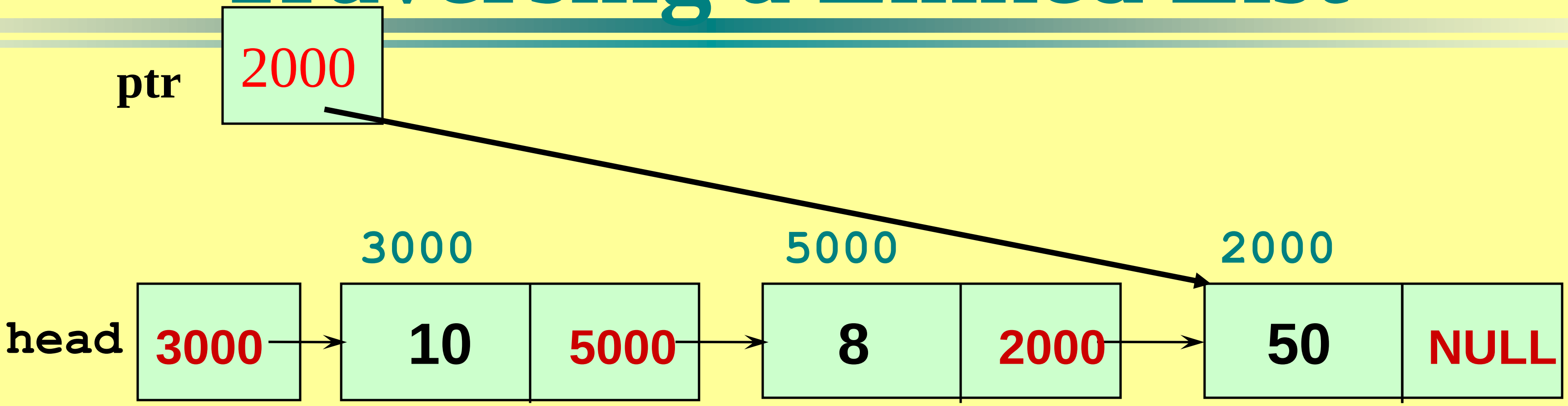
```
//PRE:  head points to a linked list
node *ptr ;

ptr  =  head ;
while (ptr != NULL)  {
      printf("%d\n",ptr->info) ;
     // Or, do something else
ptr  =  ptr->next ;
}
```

Traversing a Linked List



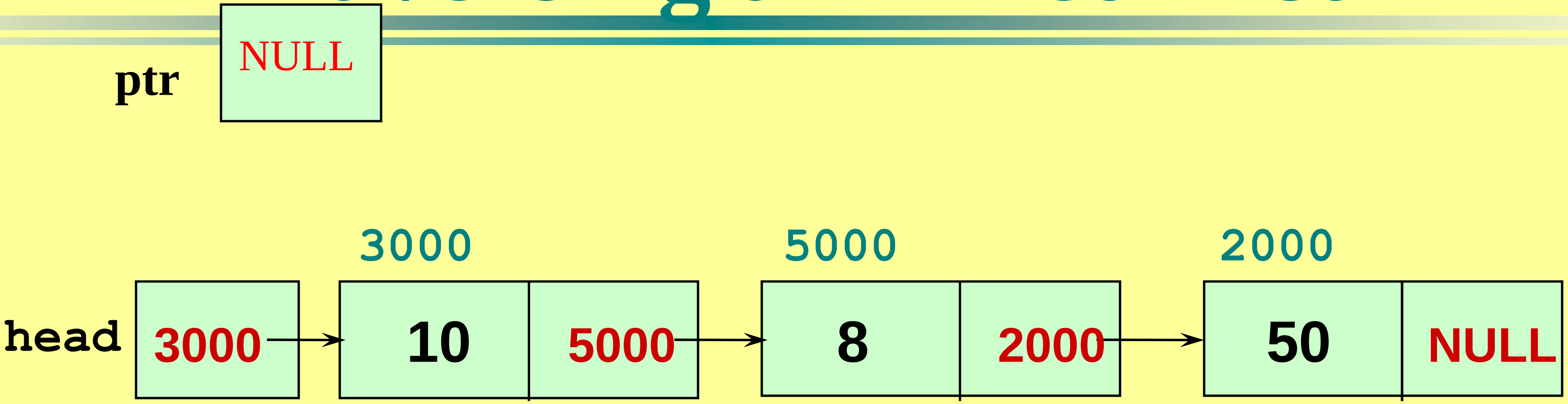
```
//PRE:  head points to a linked list
node *ptr ;

ptr  =  head ;
while (ptr != NULL)  {
       printf(“%d\n”,ptr->info) ;
      // Or, do something else
ptr  =  ptr->next ;
}
```

Traversing a Linked List



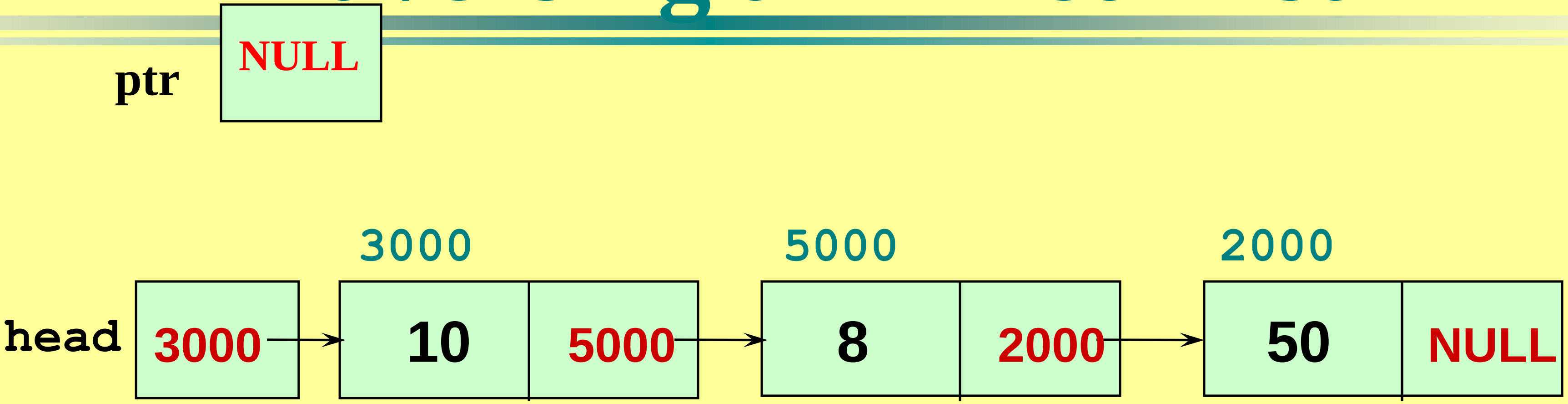
```
//PRE:  head points to a linked list
node *ptr ;

ptr  =  head ;
while (ptr != NULL)  {
       printf("%d\n",ptr->info) ;
      // Or, do something else
ptr  =  ptr->next ;
}
```

Traversing a Linked List



```
//PRE:  head points to a linked list
node *ptr ;

ptr  =  head ;
while (ptr != NULL)  {
      printf("%d\n",ptr->info) ;
     // Or, do something else
ptr  =  ptr->next ;
}
```

Traversing a Linked List



```
//PRE:  head points to a linked list
node *ptr ;

ptr  =  head ;
while (ptr != NULL)   {
       printf("%d\n",ptr->info) ;
      // Or, do something else
ptr  =  ptr->next ;
}
```

Creating `new node / list`

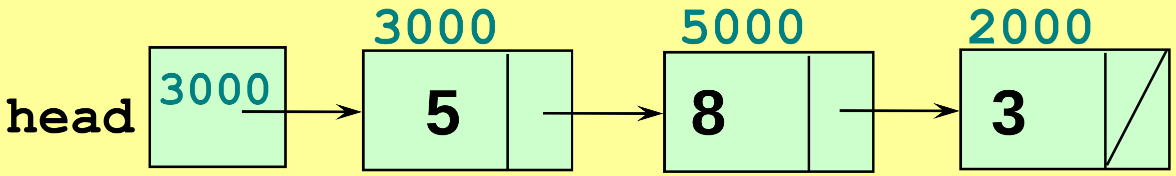
If memory is available new node will be allocated using malloc(), and it returns an address of the memory allocated to pointer.

The dynamically allocated object exists until the free operation destroys it.

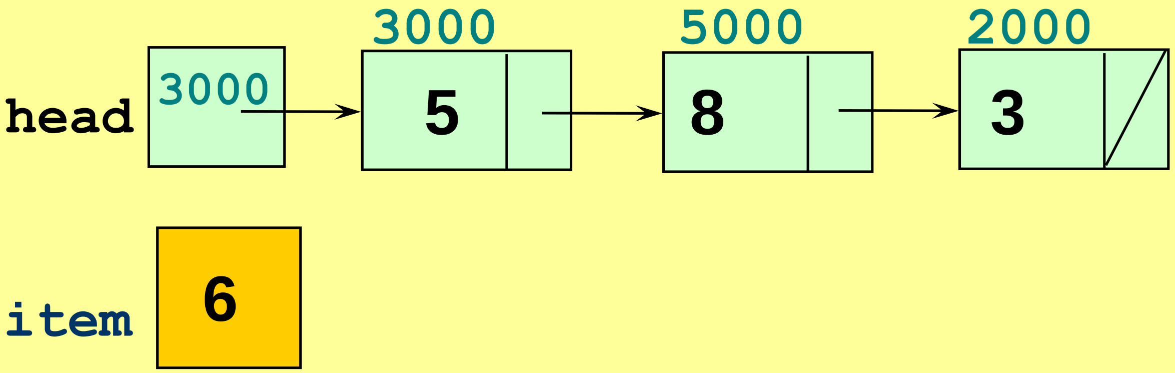
Inserting a Node at the Front of a List

```
// Type DECLARATIONS
struct NodeType  {
   int           info;
   struct NodeType *next;
};

// Variable DECLARATIONS
struct NodeType *head;
```

Inserting a Node at the Front of a List

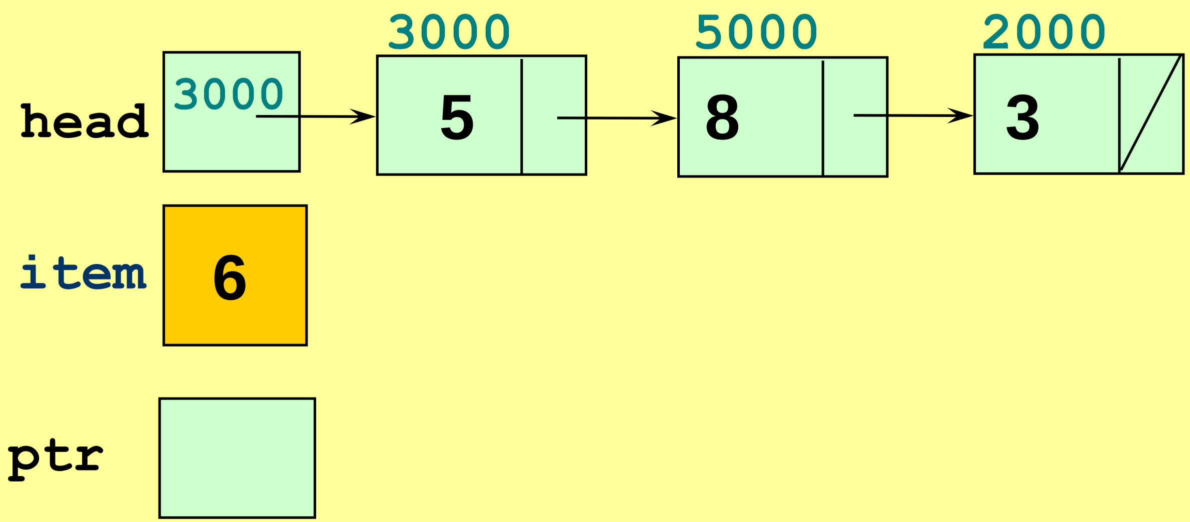


Inserting a Node at the Front of a List



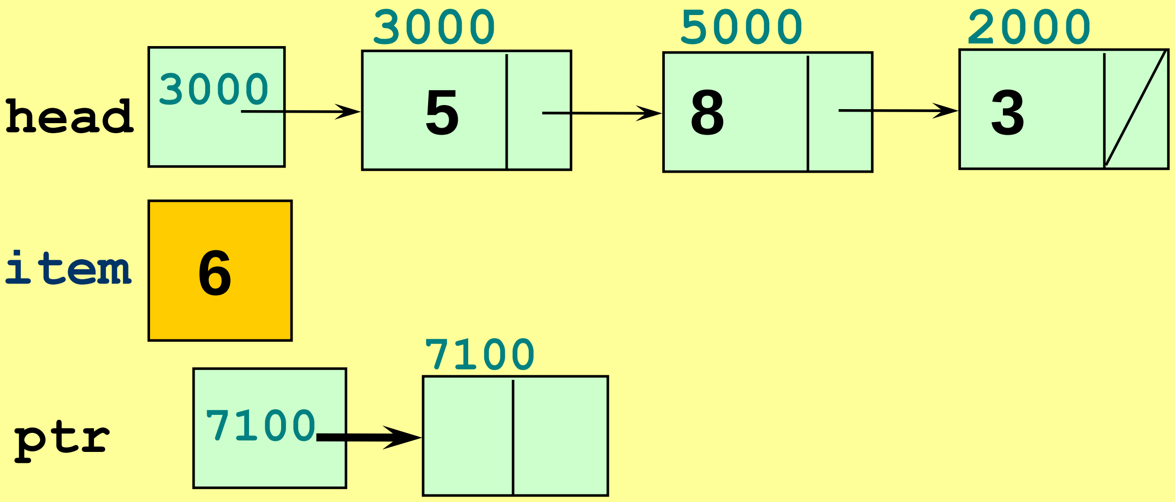
```
int item = 6;
struct NodeType * ptr;
 ptr = (struct NodeType *)malloc(sizeof(struct NodeType));
 ptr ->info = item;


ptr ->next = head;
head = ptr;
```

Inserting a Node at the Front of a List



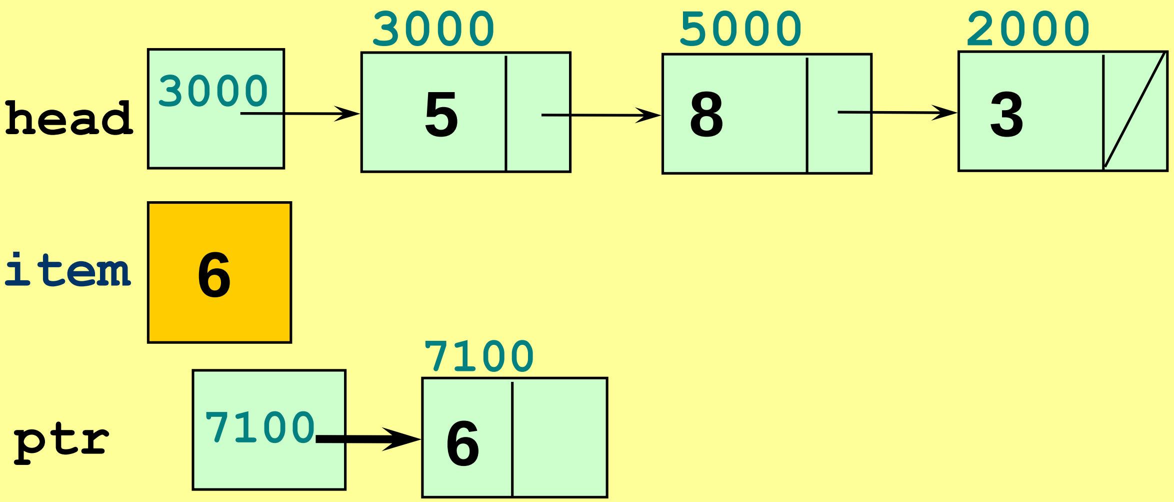
```
int     item = 6;
struct NodeType * ptr;
 ptr = (struct NodeType *)malloc(sizeof(struct NodeType));
 ptr ->info = item;


ptr ->next = head;
head = ptr;
```

Inserting a Node at the Front of a List



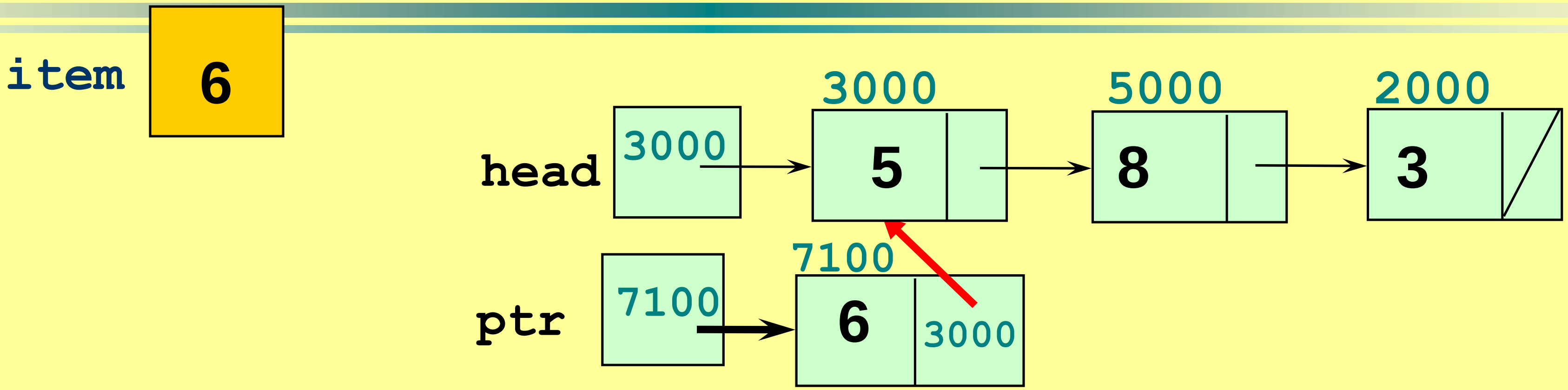
```
int     item = 6;
struct NodeType * ptr;
 ptr = (struct NodeType *)malloc(sizeof(struct NodeType));
 ptr ->info = item;


ptr ->next = head;
head = ptr;
```

Inserting a Node at the Front of a List



```
int     item = 6;
struct NodeType * ptr;
 ptr = (struct NodeType *)malloc(sizeof(struct NodeType));
 ptr ->info = item;




ptr ->next = head;
head = ptr;
```

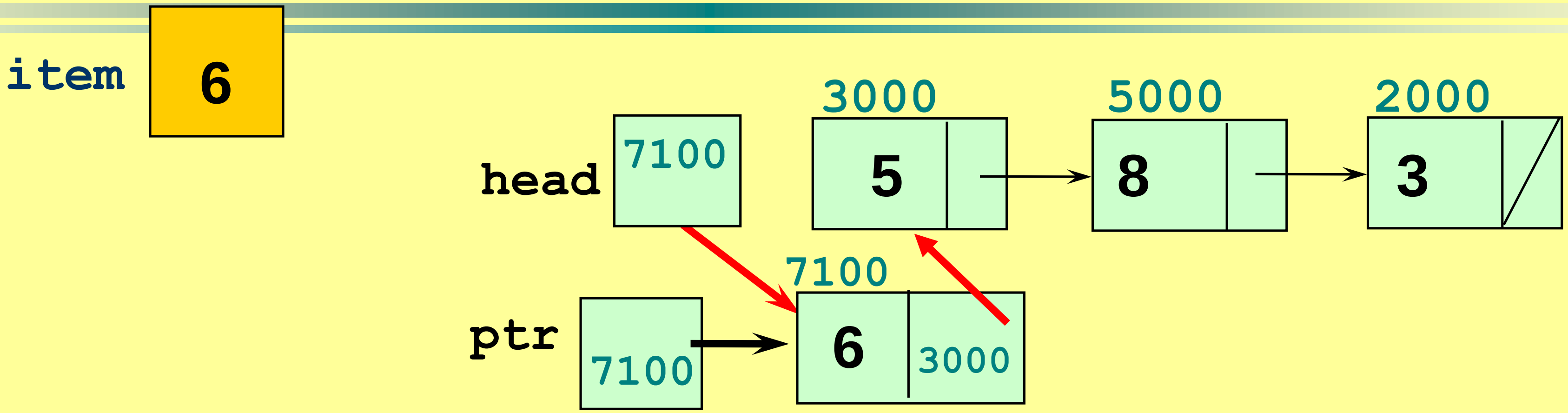
Inserting a Node at the Front of a List



```
int      item = 6;
struct NodeType * ptr;
 ptr = (struct NodeType *)malloc(sizeof(struct NodeType));
 ptr ->info = item;
```

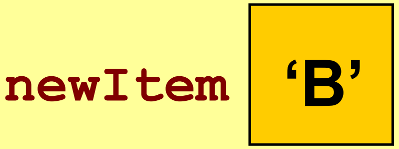
```
ptr ->next = head;
head = ptr;
```

Inserting a Node at the Front of a List

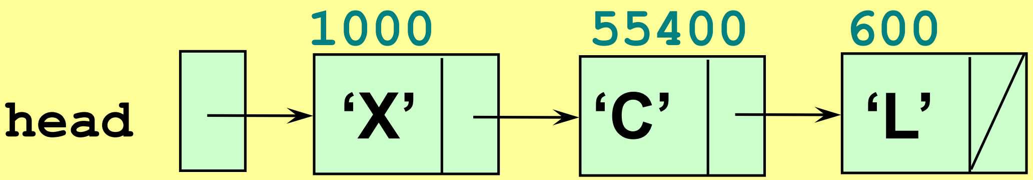


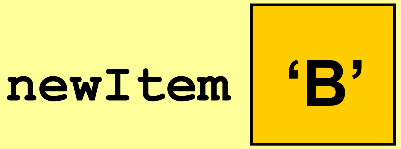
```
int      item = 6;
struct NodeType *ptr;
 ptr = (struct NodeType *)malloc(sizeof(struct NodeType));
 ptr ->info = item;




ptr ->next = head;
head = ptr;
```

Inserting a node at the front of a list



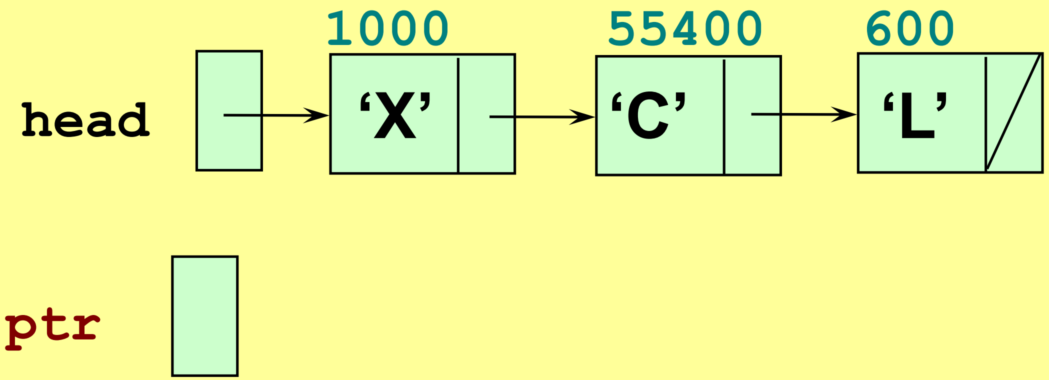
```
char newItem = ‘B’;
struct NodeType *  ptr;
ptr= (struct NodeType *)malloc(sizeof(struct NodeType);
ptr->info = newItem;
ptr->next = head;
head = ptr;
```

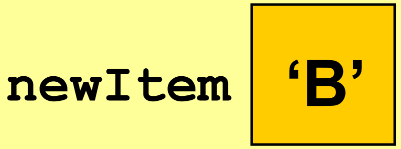




Inserting a node at the front of a list

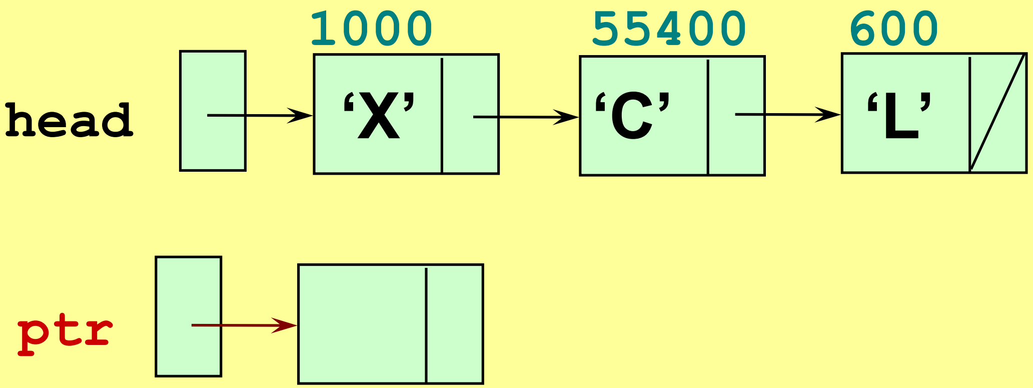
```
char newItem = ‘B’;
struct NodeType *  ptr;
ptr= (struct NodeType *)malloc(sizeof(struct NodeType);
ptr->info = newItem;
ptr->next = head;
head = ptr;
```

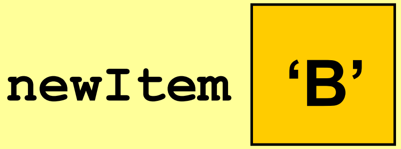




Inserting a node at the front of a list

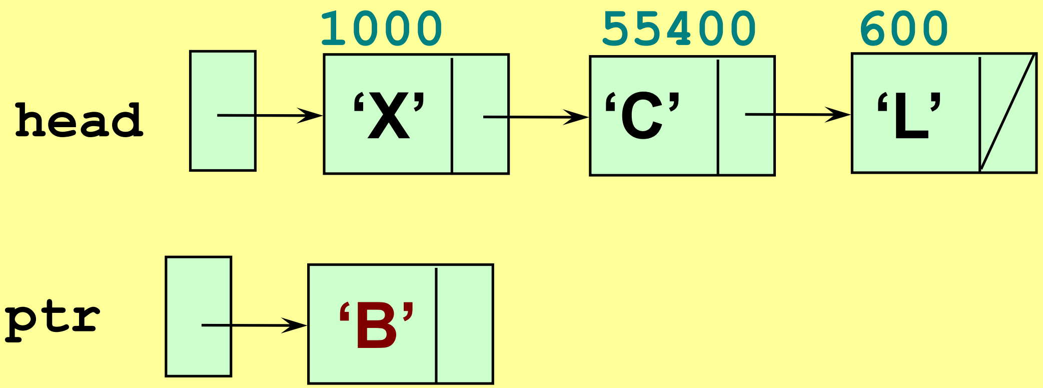
```
char newItem = ‘B’;
struct NodeType *  ptr;
ptr= (struct NodeType *)malloc(sizeof(struct NodeType);
ptr->info = newItem;
ptr->next = head;
head = ptr;
```

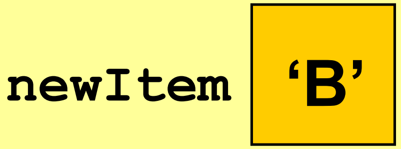




Inserting a node at the front of a list

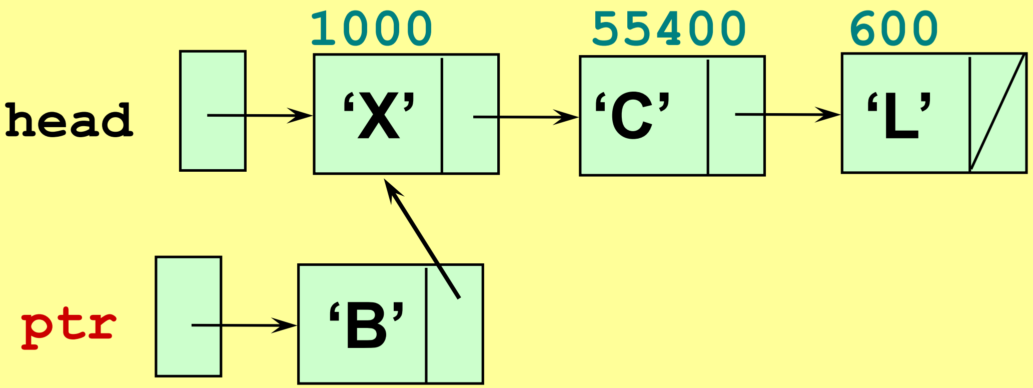
```
char newItem = ‘B’;
struct NodeType *  ptr;
ptr= (struct NodeType *)malloc(sizeof(struct NodeType);
ptr->info = newItem;
ptr->next = head;
head = ptr;
```

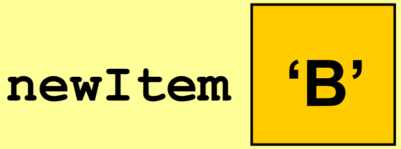




Inserting a node at the front of a list

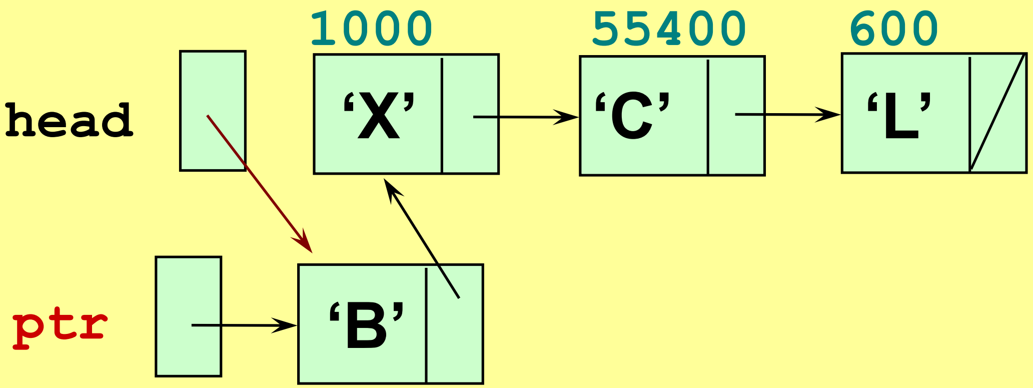
```
char newItem = ‘B’;
struct NodeType *  ptr;
ptr= (struct NodeType *)malloc(sizeof(struct NodeType);
ptr->info = newItem;
ptr->next = head;
head = ptr;
```





Inserting a node at the front of a list

```
char newItem = ‘B’;
struct NodeType *  ptr;
ptr= (struct NodeType *)malloc(sizeof(struct NodeType);
ptr->info = newItem;
ptr->next = head;
head = ptr;
```



Program to create a Linked List by inserting each node at the front of the list

Step 1: declare the structure

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

Step 2: declare the pointer variables of type structure

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

```
/* Declare and Initialize head pointer */
struct node *head = NULL; // To indicate the list is empty
```

Step 3: Write the function to create list by inserting new node at beginning

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

/* Declare and Initialize node pointers */
struct node *head = NULL; // To indicate the list is empty
```

```
//creates a list be adding node at beginning
void create_insertbeg()
{
        struct node* temp;
        //creating new node
        temp = (struct node*)malloc(sizeof(struct node));
        //Store values in new node
        printf("Enter node data: ");
        scanf("%d", &temp->data);
        temp->next = NULL;
        if(head==NULL)   { //check if the list is empty
                head = temp;
        }
        else{
                temp->next = head;
                head = temp;
            }
}
```

```
//creates a list be adding node at beginning
void create_insertbeg()
{
        struct node* temp;
        //creating new node
        temp = (struct node*)malloc(sizeof(struct node));
        //Store values in new node
        printf("Enter node data: ");
        scanf("%d", &temp->data);
        temp->next = NULL;
        if(head==NULL)   { //check if the list is empty
                head = temp;
        }
        else{
                temp->next = head;
                head = temp;
            }
}
```

```
//creates a list be adding node at beginning
void create_insertbeg()
{
        struct node* temp;
       //creating new node
        temp = (struct node*)malloc(sizeof(struct node));
        //Store value in new node’s data part
        printf("Enter node data: ");
        scanf("%d", &temp->data);
        temp->next = NULL;
        if(head==NULL)   { //check if the list is empty
                head = temp;
        }
        else{
                temp->next = head;
                head = temp;
            }
}
```

```
//creates a list be adding node at beginning
void create_insertbeg()
{
        struct node* temp;
        //creating new node
        temp = (struct node*)malloc(sizeof(struct node));
        //Store value in new node’s data part
        printf("Enter node data: ");
        scanf("%d", &temp->data);
        //Initialize new node’s link/next part to NULL
        temp->next = NULL;
        if(head==NULL)   { //check if the list is empty
                head = temp;
        }
        else{
                temp->next = head;
                head = temp;
            }
}
```

```
//creates a list be adding node at beginning
void create_insertbeg()
{
        struct node* temp;
        //creating new node
        temp = (struct node*)malloc(sizeof(struct node));
        //Store value in new node’s data part
        printf("Enter node data: ");
        scanf("%d", &temp->data);
        //Initialize new node’s link/next part to NULL
        temp->next = NULL;
        //Connect the new node with list
        if(head==NULL)   { //check if the list is empty
                head = temp;
        }
        else{
                temp->next = head;
                head = temp;
            }
}
```

```
//creates a list be adding node at beginning
void create_insertbeg()
{
        struct node* temp;
        //creating new node
        temp = (struct node*)malloc(sizeof(struct node));
        //Store value in new node’s data part
        printf("Enter node data: ");
        scanf("%d", &temp->data);
        //Initialize new node’s link/next part to NULL
        temp->next = NULL;
        //Connect the new node with list
        if(head==NULL)   { //check if the list is empty
                head = temp;
        }
        else{
                temp->next = head;
                head = temp;
            }
}
```

Complexity = ?

Step 4: Write the function to display list

```
// print the linked list value
void printLinkedlist() {
  struct node *p=head;
  while (p != NULL) {
          printf("%d ", p->value);
    p = p->next;
  }
}
```

```
if(p == NULL)
    {
        printf("\nEmpty List\n");
    }
```

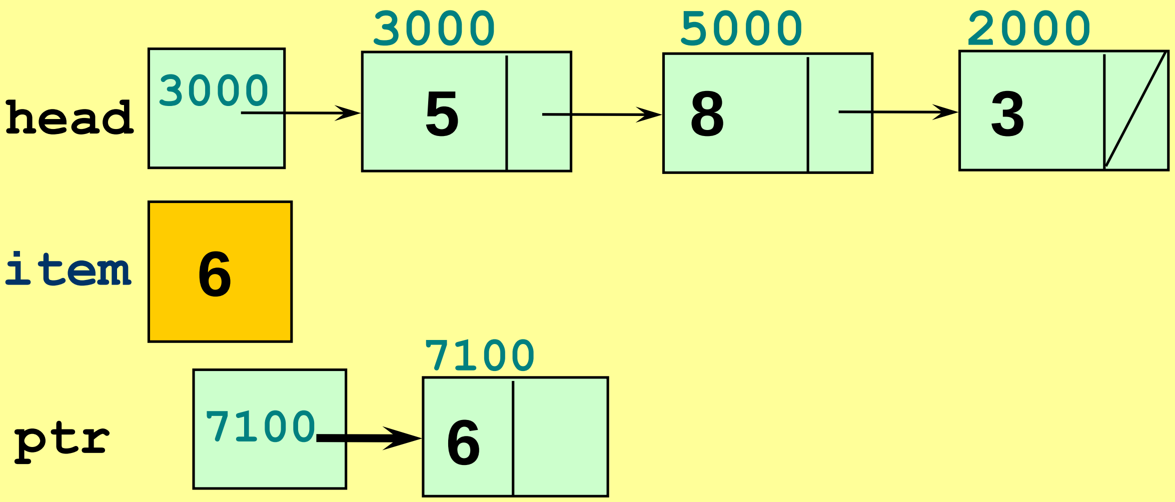
Step 4: Write the function to display list

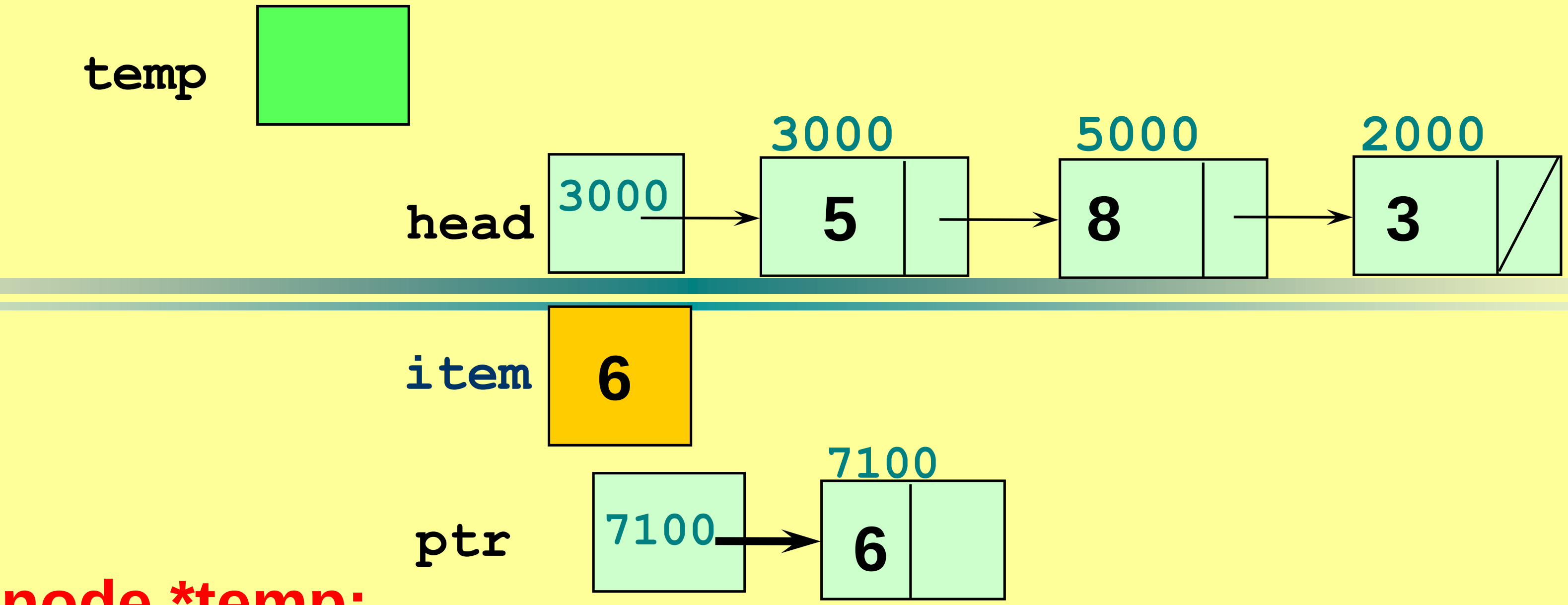
```
// print the linked list value
void printLinkedlist() {
 struct node *p=head;
if(ptr == NULL)
   {
       printf("\nEmpty List\n");
   }
else{
 while (p != NULL) {
         printf("%d ", p->value);
   p = p->next;
 }
}
}
```

Step 5: Write the main function and call the create and display functions

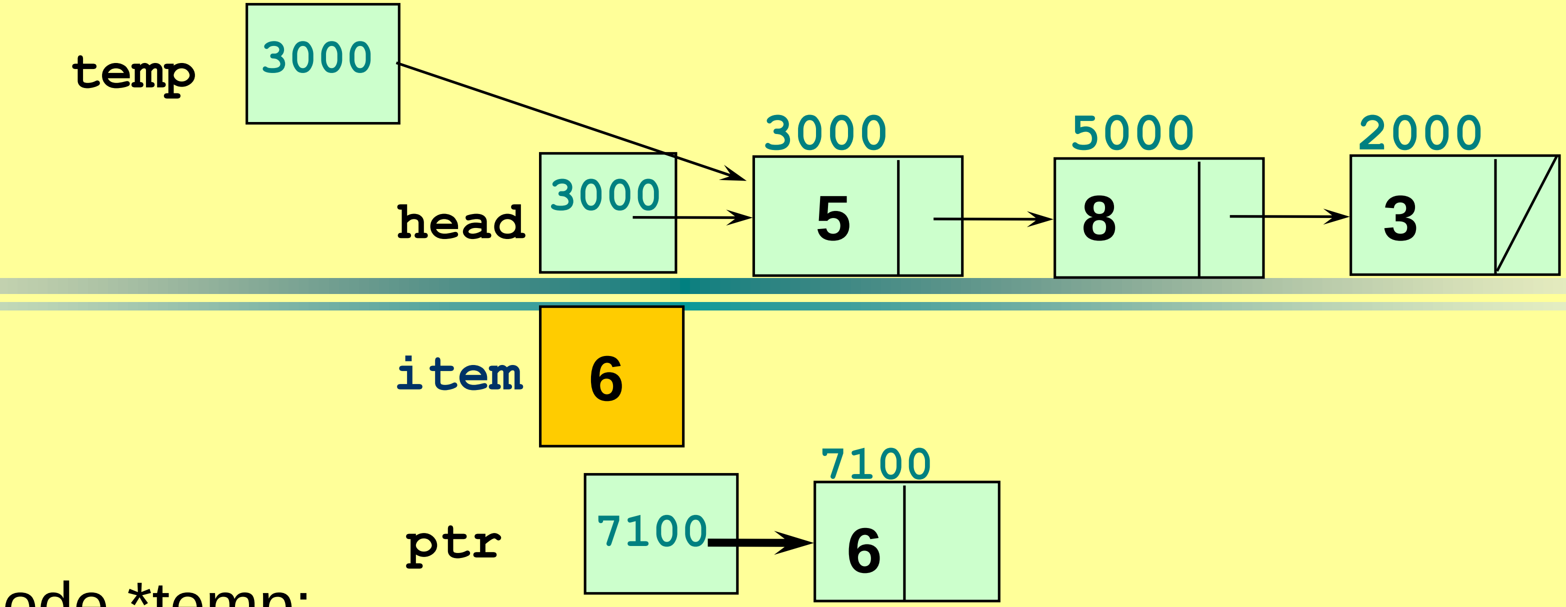
```
main()
{
int ch;
 do {
        create_insertbeg();
        printf(“Enter 1 to continue, 0 to Stop”);
        scanf(“%d”,&ch);
      }while(ch==1);
printLinkedlist() ;
}
```

Write the function to create list by inserting new node at END

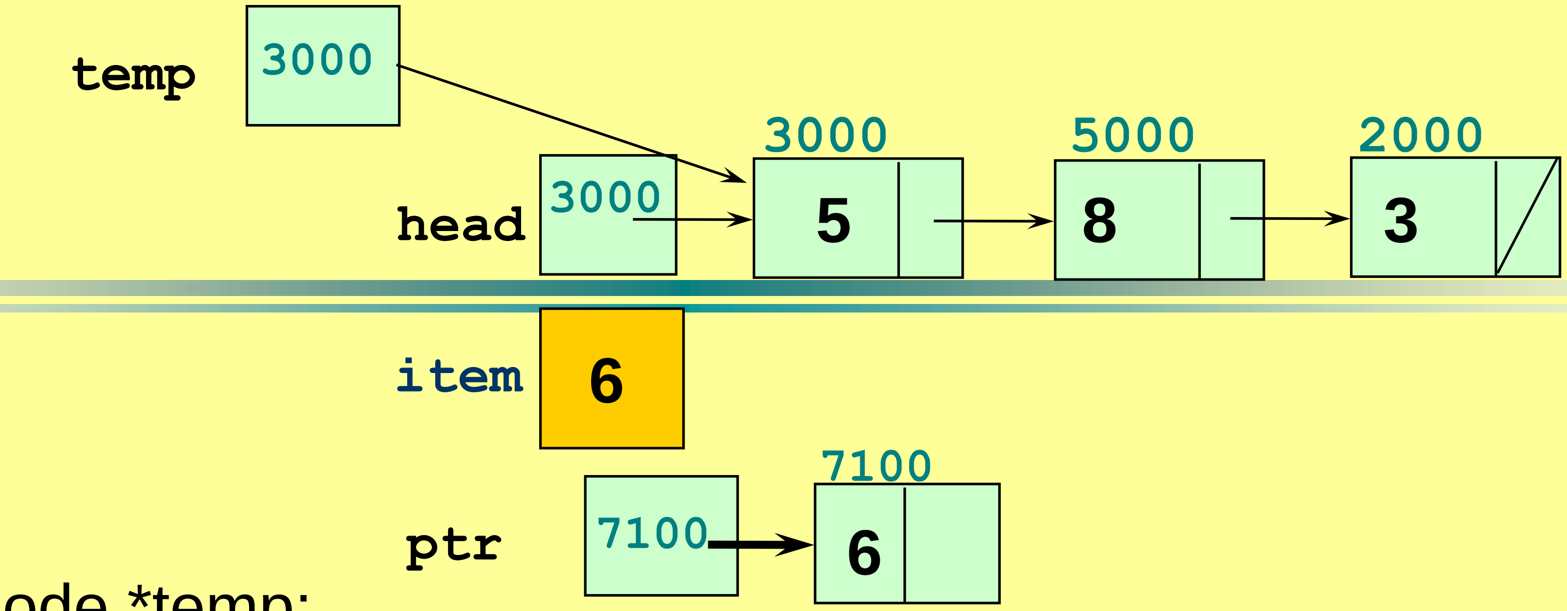




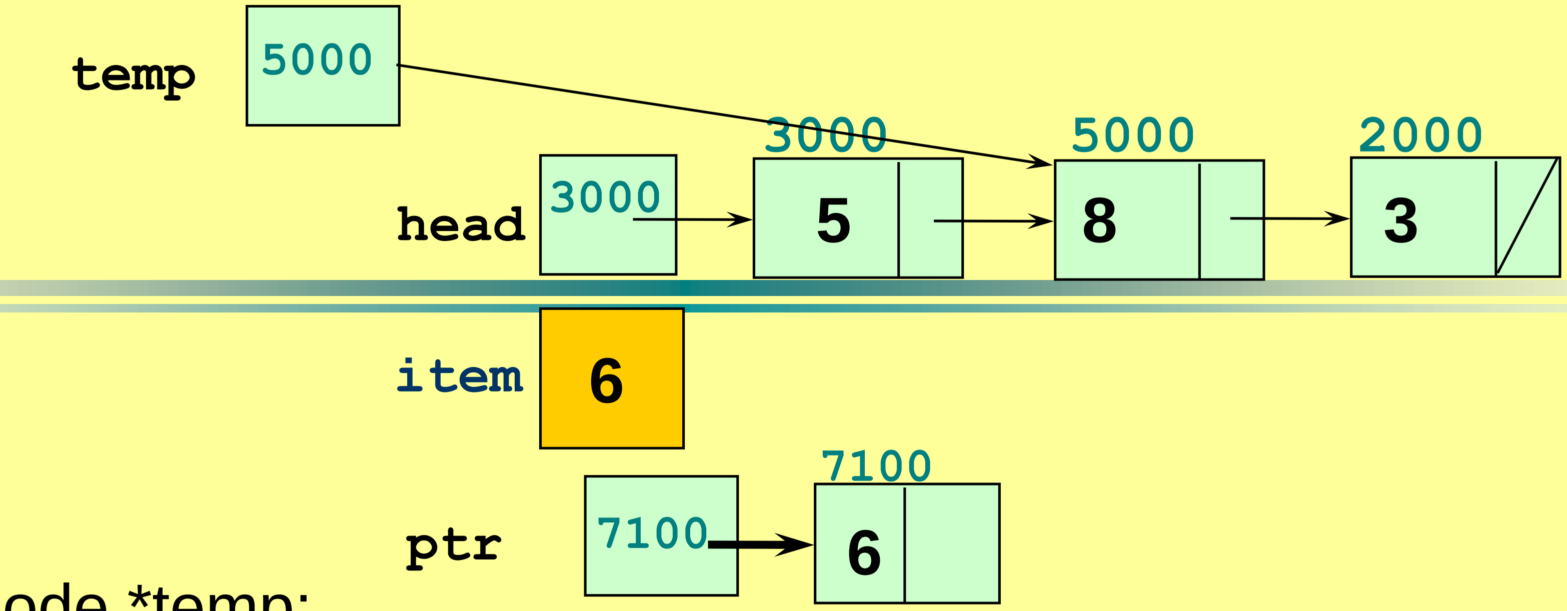
```
struct node *temp;
temp = head;
        while (temp -> next != NULL)
        {
           temp = temp -> next;
        }


         temp->next = ptr;
```



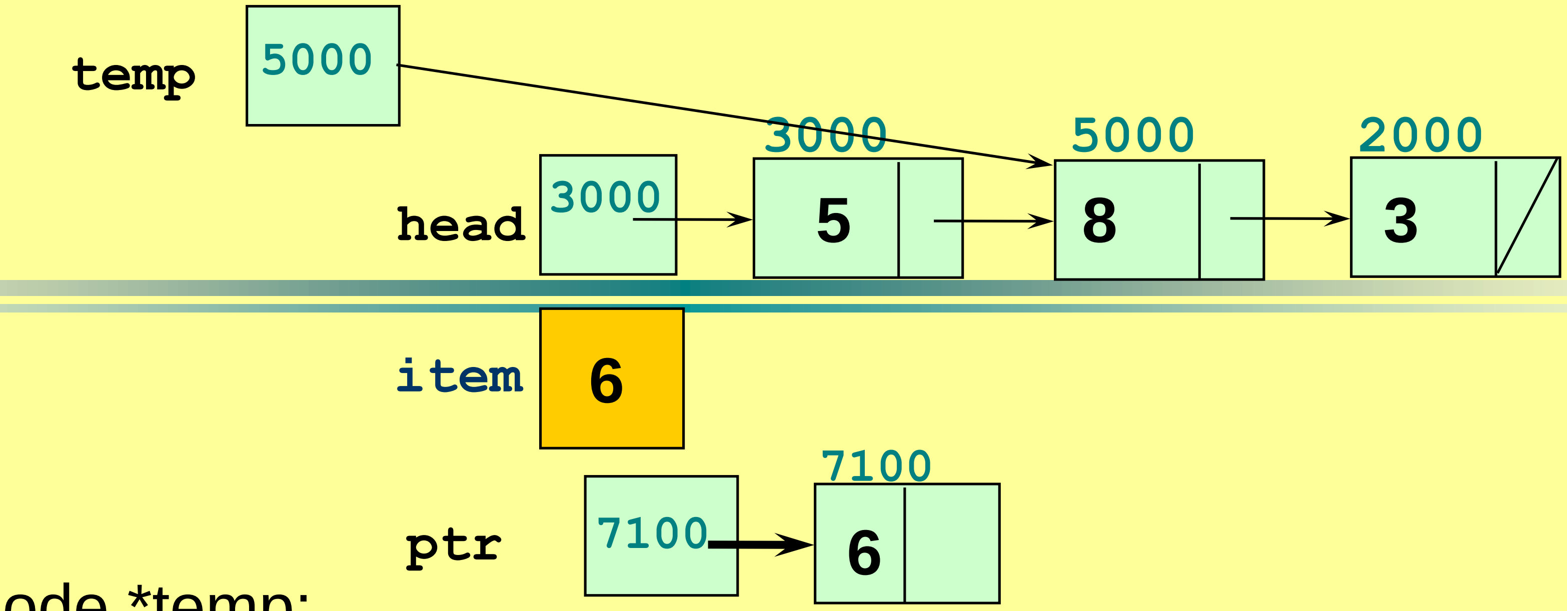
```
struct node *temp;
temp = head;
          while (temp -> next != NULL)
          {
              temp = temp -> next;
          }


           temp->next = ptr;
```



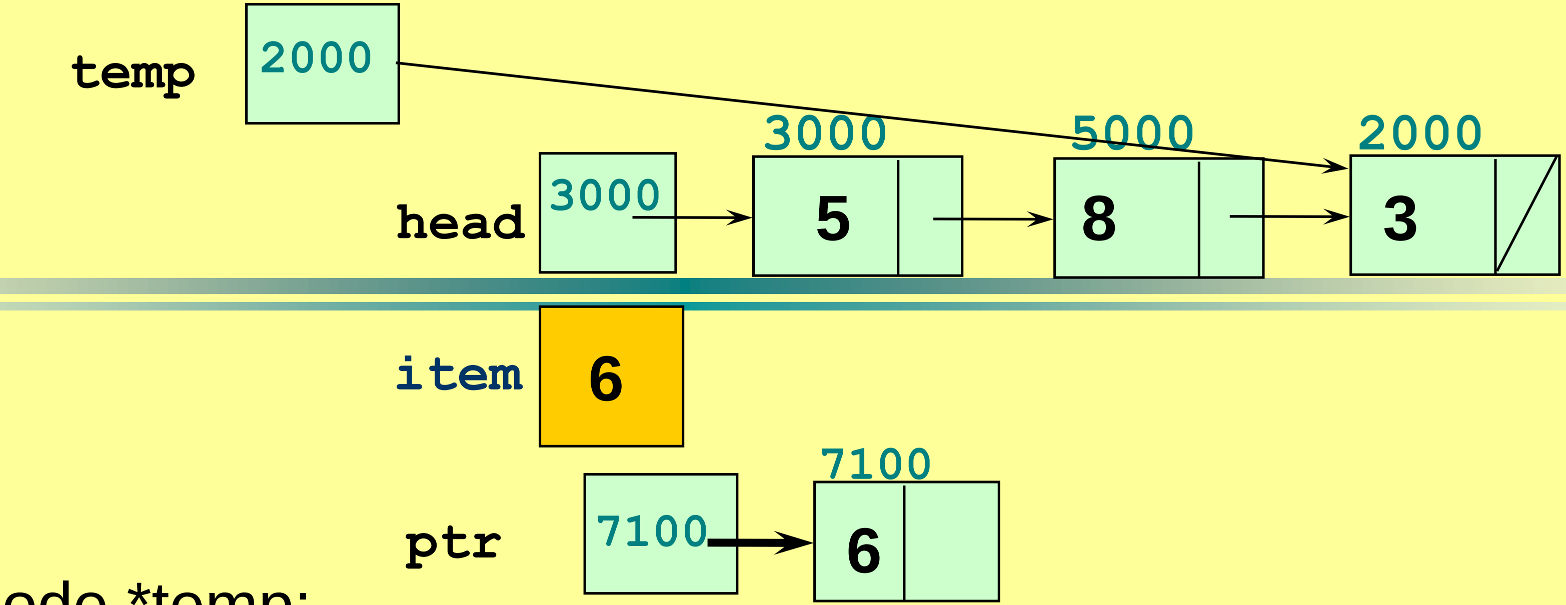
```
struct node *temp;
temp = head;
          while (temp -> next != NULL)
          {
              temp = temp -> next;
          }


          temp->next = ptr;
```



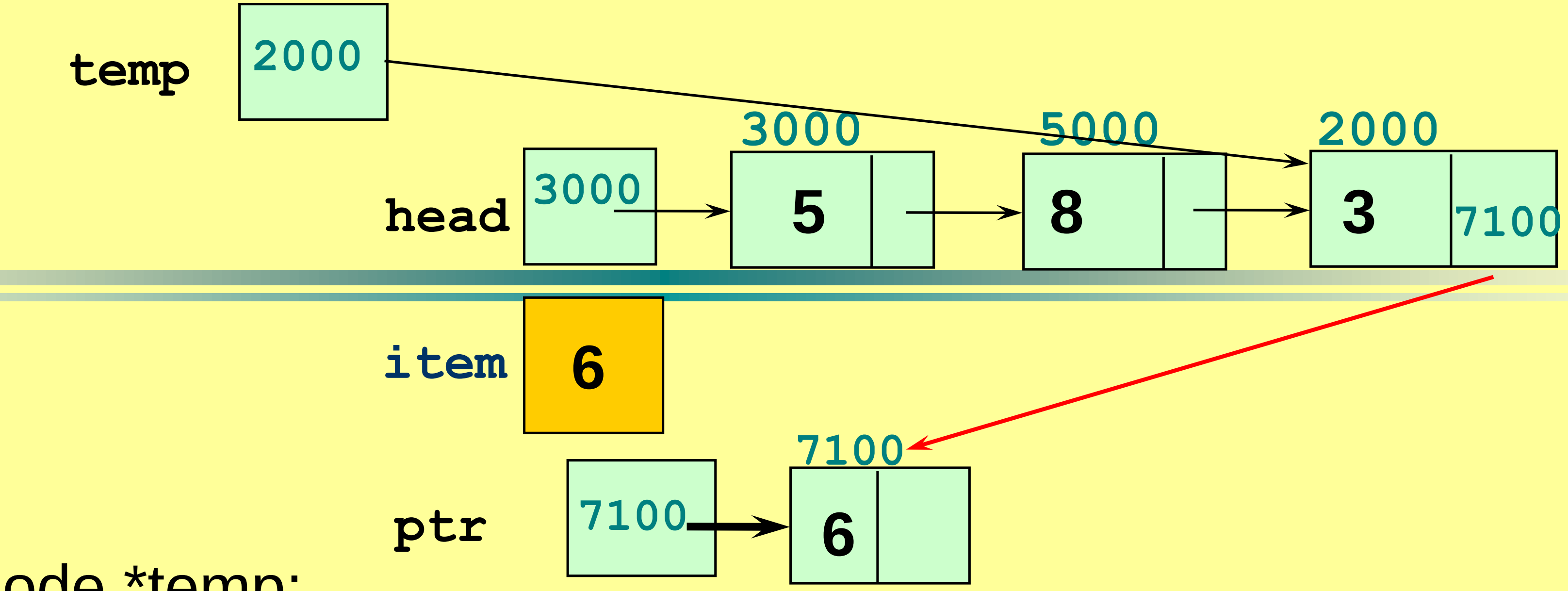
```
struct node *temp;
temp = head;
          while (temp -> next != NULL)
          {
             temp = temp -> next;
          }


           temp->next = ptr;
```



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


         temp->next = ptr;
```



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


          temp->next = ptr;
```



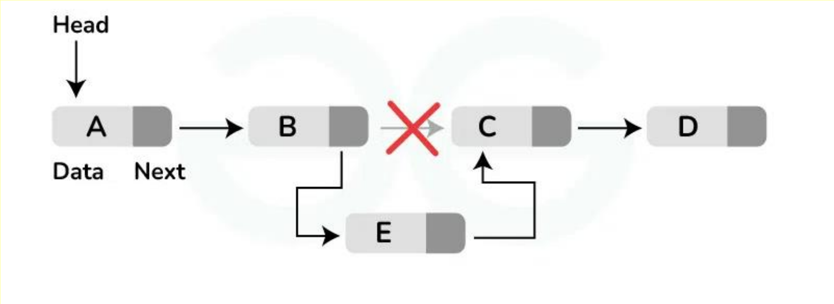
```
struct node *temp;
temp = head;
        while (temp -> next != NULL)
        {
            temp = temp -> next;
        }


        temp->next = ptr;
```

```
void lastinsert()
{
    struct node *ptr,*temp;

    ptr = (struct node*)malloc(sizeof(struct node));
    printf("\nEnter value");
    scanf("%d",&ptr->data);
    ptr -> next = NULL;


if(head == NULL)  //List is empty
        {
            head = ptr;
        }
        else
        {
            temp = head;
            while (temp -> next != NULL)
            {
                temp = temp -> next;
            }


             temp->next = ptr;
         }

}
```

Insert a New Node after a Given Node in Linked List

How to Insert a New Node after a Given Node in Linked List

- **Approach:**
- To insert a node after a **given node** in a Linked List, we need to:
- Check if the given node exists or not (Traverse the list).
- If it do not exists,
  - terminate the process.
- If the **given node** exists,
  - Create a new node
  - Store the original next pointer of **given node** to the next pointer of new node
  - Change the next pointer of **given node** to the new node


Head
A
Data
Next
B
C
D
E