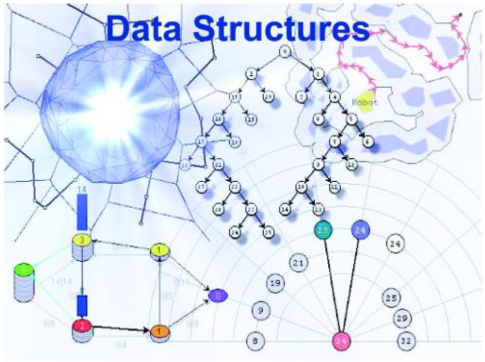
BBM 201
DATA STRUCTURES

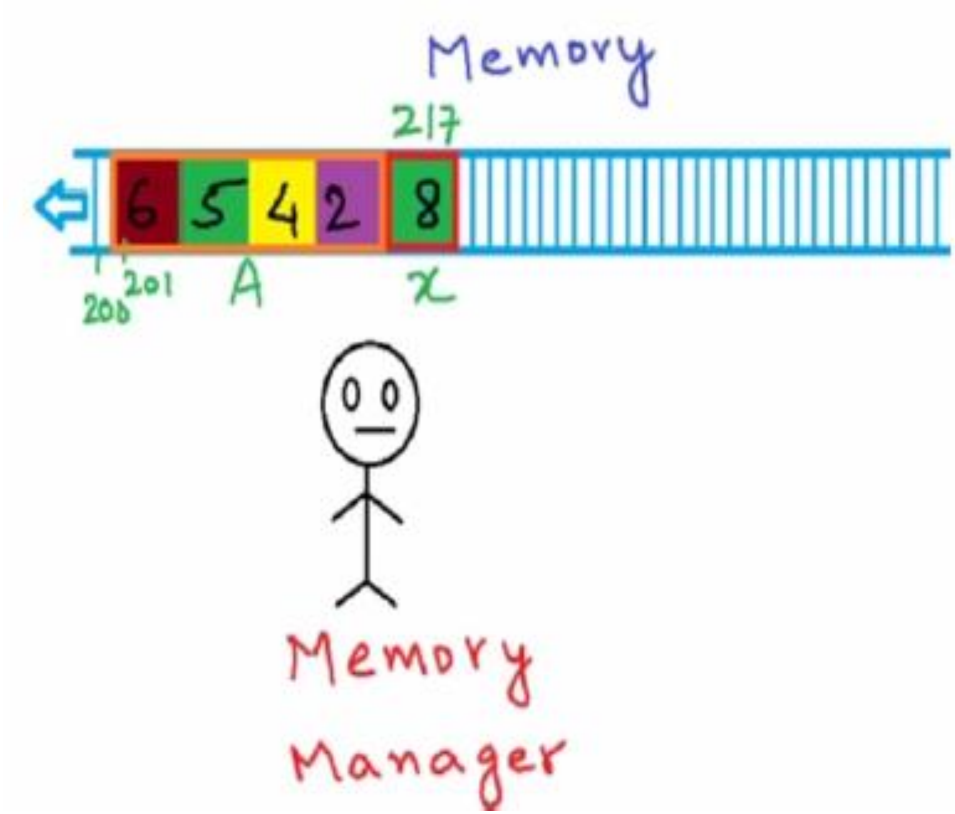
Lecture 8:

Dynamically Allocated Linked Lists



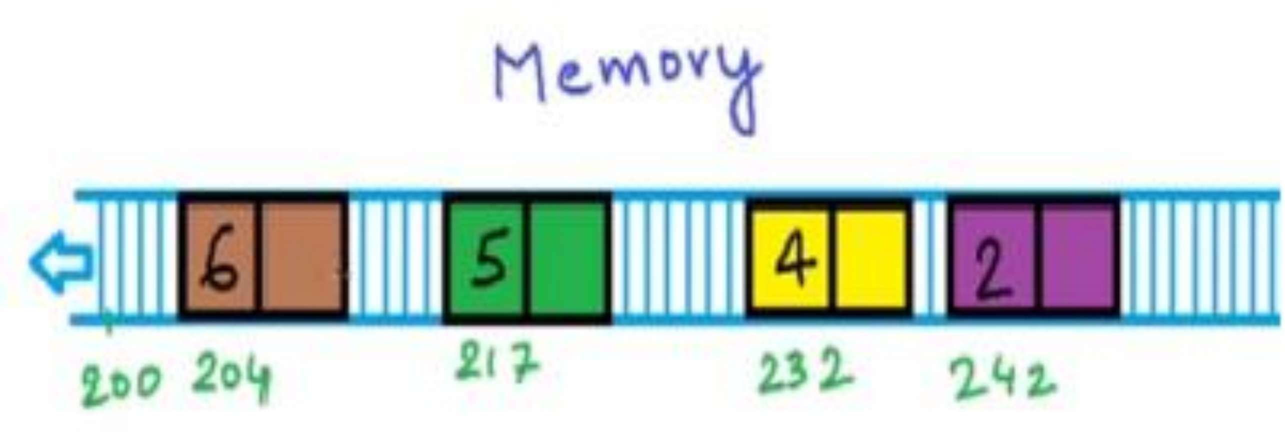
2019-2020 Fall

```
int x;
x = 8;
int A[4];
```



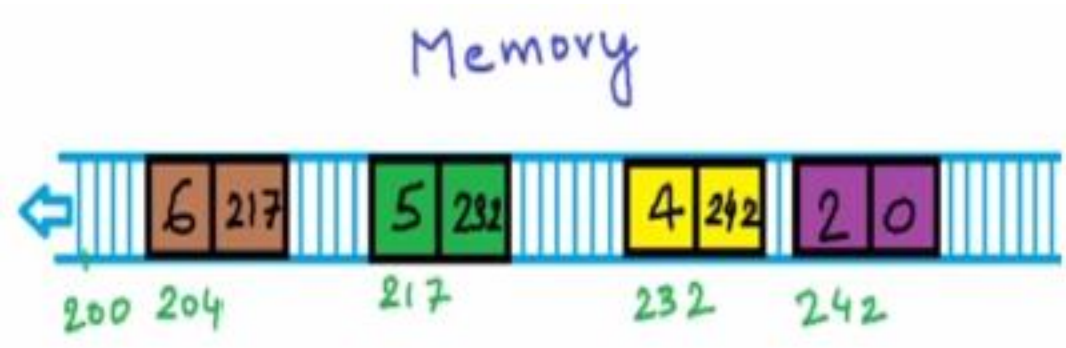
An array is stored as one contiguous block of memory.
How can we add a fifth element to the array A above???

If we used dynamic memory allocation, we need to use realloc.
Otherwise, we can use a linked list.



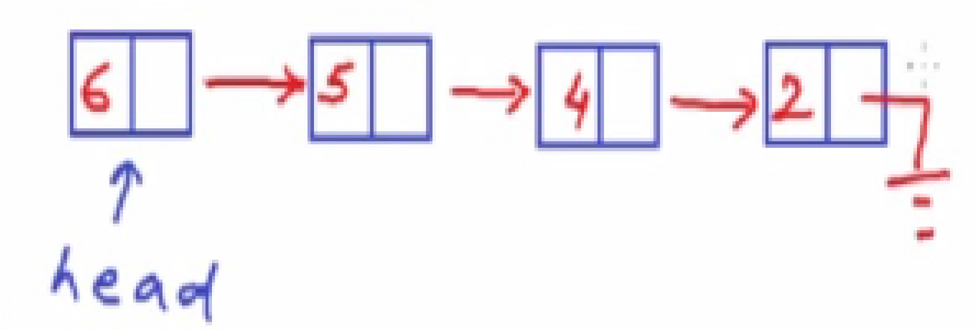
To input elements to the linked list, the memory manager finds an address for each element one by one.

We store not only the value of each element but also the address of the following element for each element.



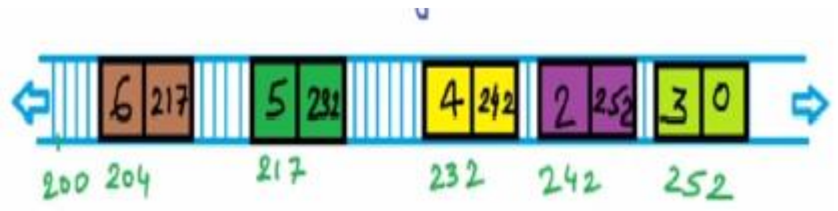

```
Struct Node
{
   int data;   // 4 bytes
   Node* link;    // 4 bytes, points to next node
}
```

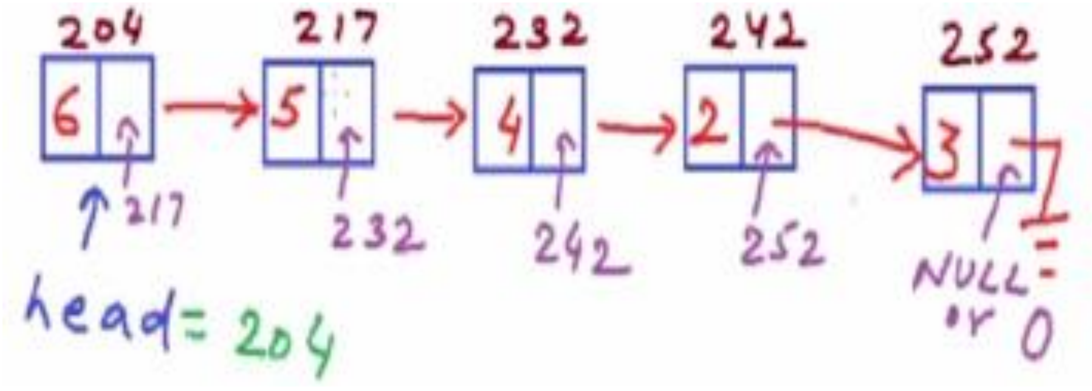
- For each element (or node) two fields are stored, each costing four bytes.



- The first node is called the head node. The link of the last node is NULL or 0.
- The address of the head node gives us access to the complete list.
- To access a node, we need to traverse ALL nodes with smaller index.

Adding a new node



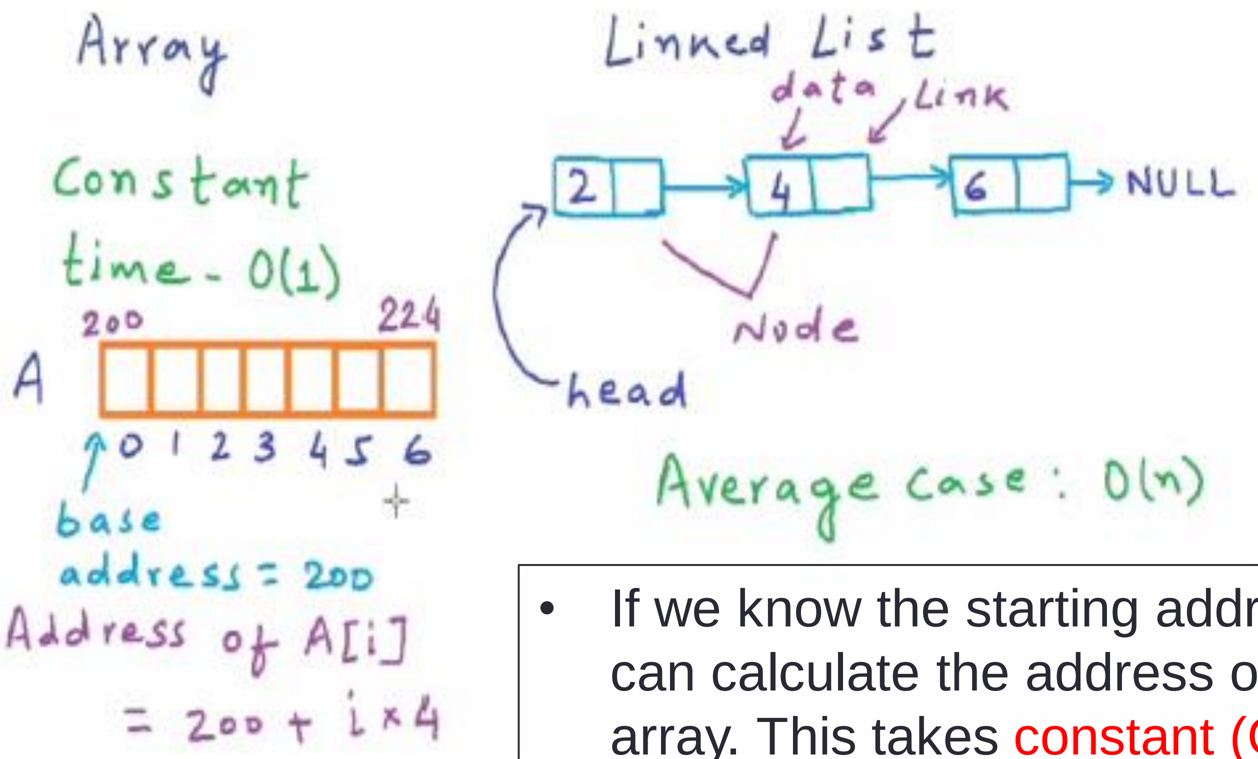


Create a node independently with some memory location and adjust the links properly. Say the node 3 gets address 252.

- The linked list is always identified by the address of the head node.
- Unlike array, it costs O(n) to access an element of a linked list.

Array vs. Linked List:

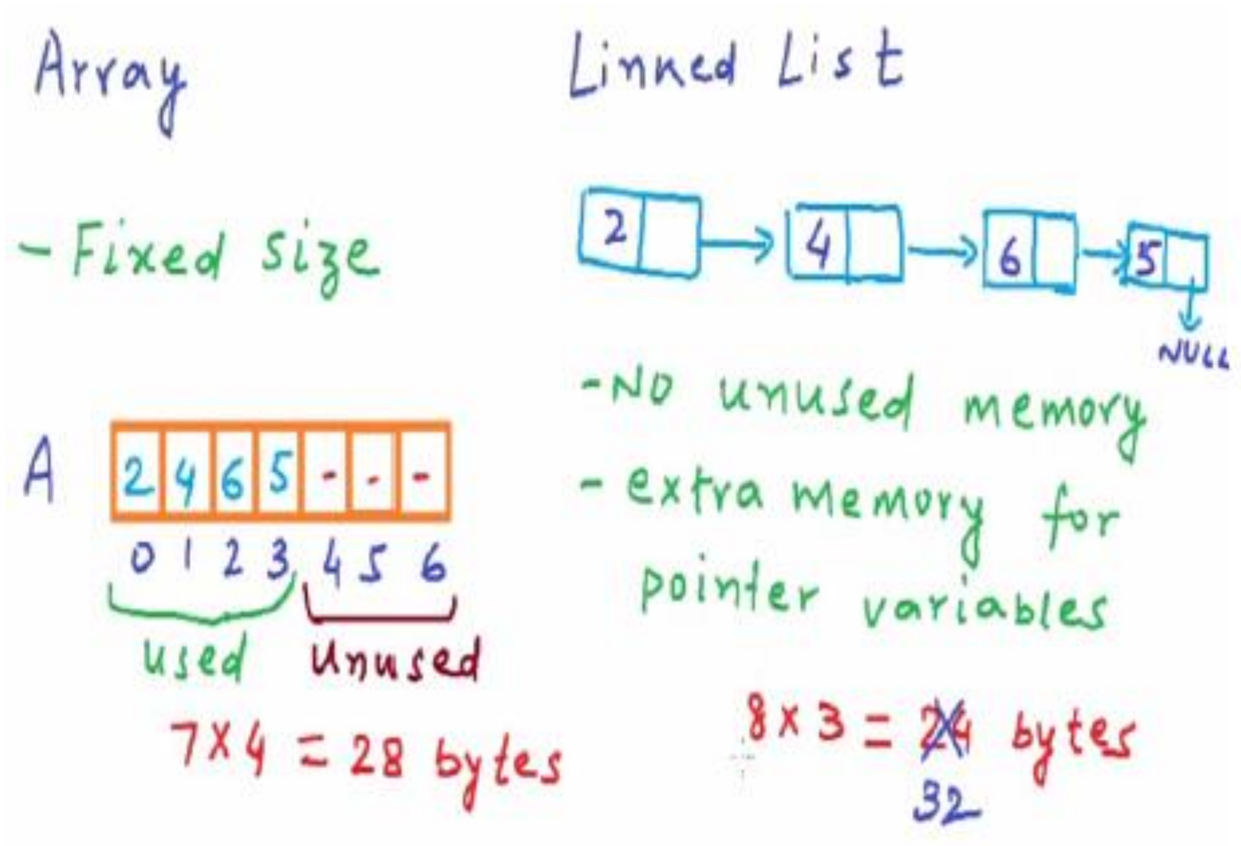
1) Cost of accessing an element



- If we know the starting address of the array, we can calculate the address of the ith element in the array. This takes constant (O(1)) time for any element in the array!
- To find the address of the i'th element in the linked list, we need to traverse all elements until that element (O(n) time).

Array vs. Linked List:

2) Memory requirements



- Before creating an array, we need to know its possible size.
- For linked list, we ask from memory one node at a time, but we use twice the size since also the address of each consecutive node is stored.

If we have a data of complex type, then for each element in the linked list, 20 bytes are used.
The strategy to decide which one to use depends on the case.

Array vs. Linked List:

2) Memory requirements

Array	Linked List
Has fixed size	No unused memory
	Extra memory for pointer variables
Memory may not be available as one large block	Memory may be available as multiple small blocks

Array vs. Linked List:

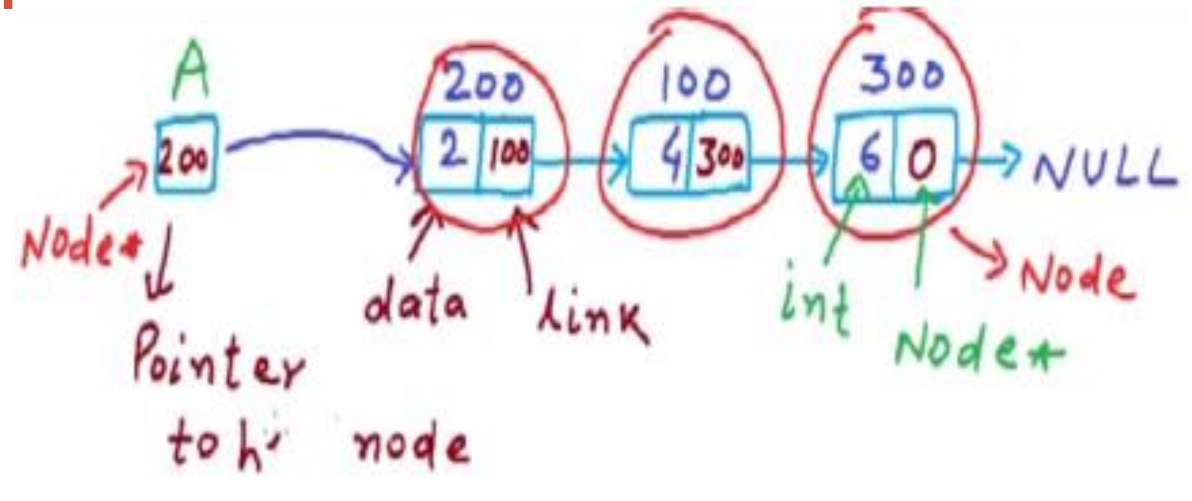
3) Cost of inserting an element

Cost of inserting a new element (worst case):	Array	Linked List
a) At the beginning	O(n)	O(1)
b) At the end	O(1) (if array is not full)	O(n)
c) At i’th position	O(n)	O(n)

- To insert an element to the beginning of an array all elements need to be shifted by one to the next address.

- To add an element to the end of a linked list all elements need to be traversed.

Linked List: Implementation in C and C++




```
Struct Node
{
    int data;
    Node* link;
}
Node* A;
A = NULL; //empty list
Node* temp = (Node*)malloc(sizeof(Node))
```

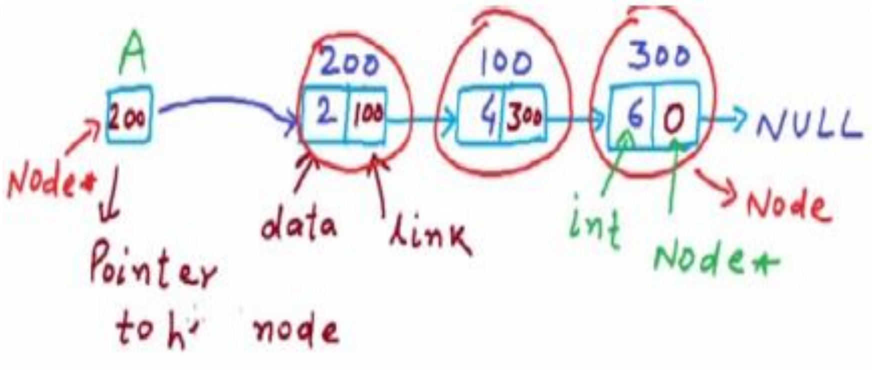
- We define such data type using Structure. Integer is the type of the variable data stored as the element of the linked list.

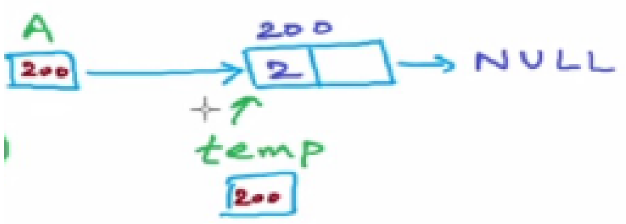
- The second field called link is of type pointer.

- The last line creates one node in the memory.

Dereferencing

```
Struct Node
{
    int data;
    Node* link;
}
Node* A;
A = NULL; //empty list
Node* temp = (Node*)malloc(sizeof(Node))
(*temp).data = 2;
(*temp).link = NULL;
A = temp;
```

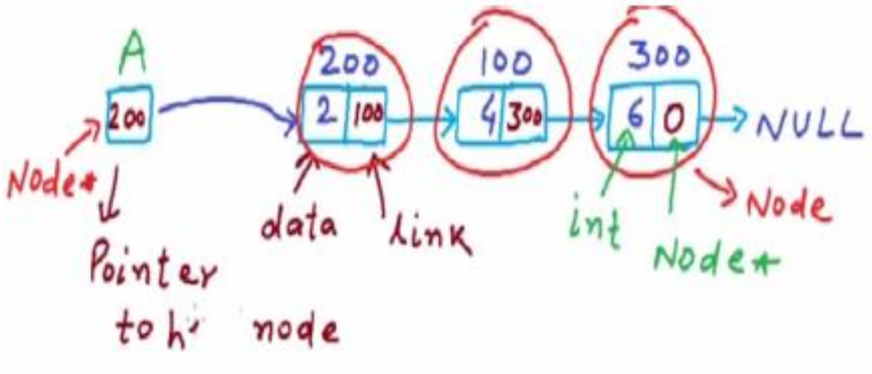


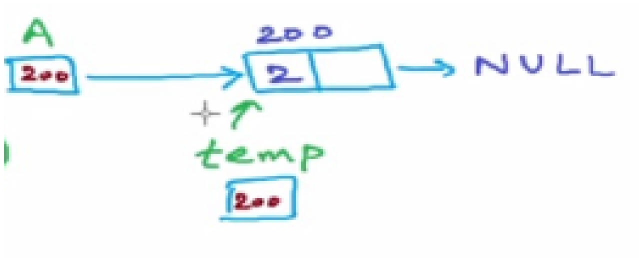


- The data part of this node is 2 and the temp variable is pointing to it.
- The link part is NULL since it is the last node.
- Finally, write the address of the newly created node to A.

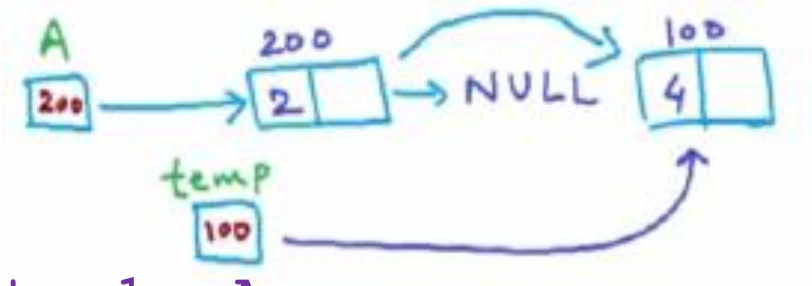
C++ implementation

```
Struct Node
{
    int data;
    Node* link;
}
Node* A;
A = NULL;
Node* temp = new Node();
temp->data = 2;
temp->link = NULL;
A = temp;
```



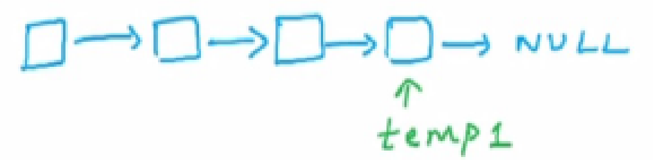


Traversal of a list




```
Node* A;
A = NULL;
Node* temp = new Node();
(*temp).data = 2;
(*temp).link = NULL;
A = temp;
temp = new Node();
temp->data = 4;
temp->link = NULL;
A->link = temp
```

```
Node* temp1 = A;
while(temp1 != NULL){
    print "temp1->data";
    temp1 = temp1->link;
}
```



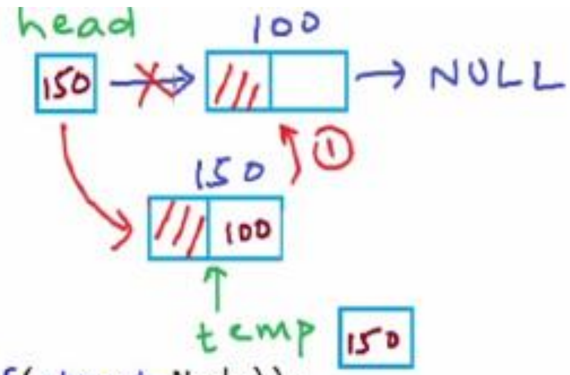
- temp stores the address of the new node.
- We need to record the address of the new node and to do so we need to traverse the whole list to go to the end of the list.
- While traversing, if the link of the node is not NULL, we can move to the next node.
- Finally, the loop prints the elements in this list.

Inserting a node at the beginning

```c
//Linked List: Inserting a node at beginning
#include<stdlib.h>
#include<stdio.h>
struct Node {
    int data;
    struct Node* next;
};
struct Node* head; // global variable, can be accessed anywhere
void Insert(int x);
void Print();
int main() {
    head = NULL; // empty list;
    printf("How many numbers?\n");
    int n,i,x;
    scanf("%d",&n);
    for(i = 0;i<n;i++){
        printf("Enter the number \n");
        scanf("%d",&x);
        Insert(x);
        Print();
    }
}
```

- Pointer storing the address of the next node is called next. (In C++, only Node* next; is written.)
- Insert each number x into the linked list by calling a method insert and print it.

Implementing the insert function




```c
struct Node* head;
void Insert(int x)
{
    Node* temp = (Node*)malloc(sizeof(struct Node));
    temp->data = x;
    temp->next = NULL;
    if(head != NULL) temp->next = head;
    head = temp;
}
```

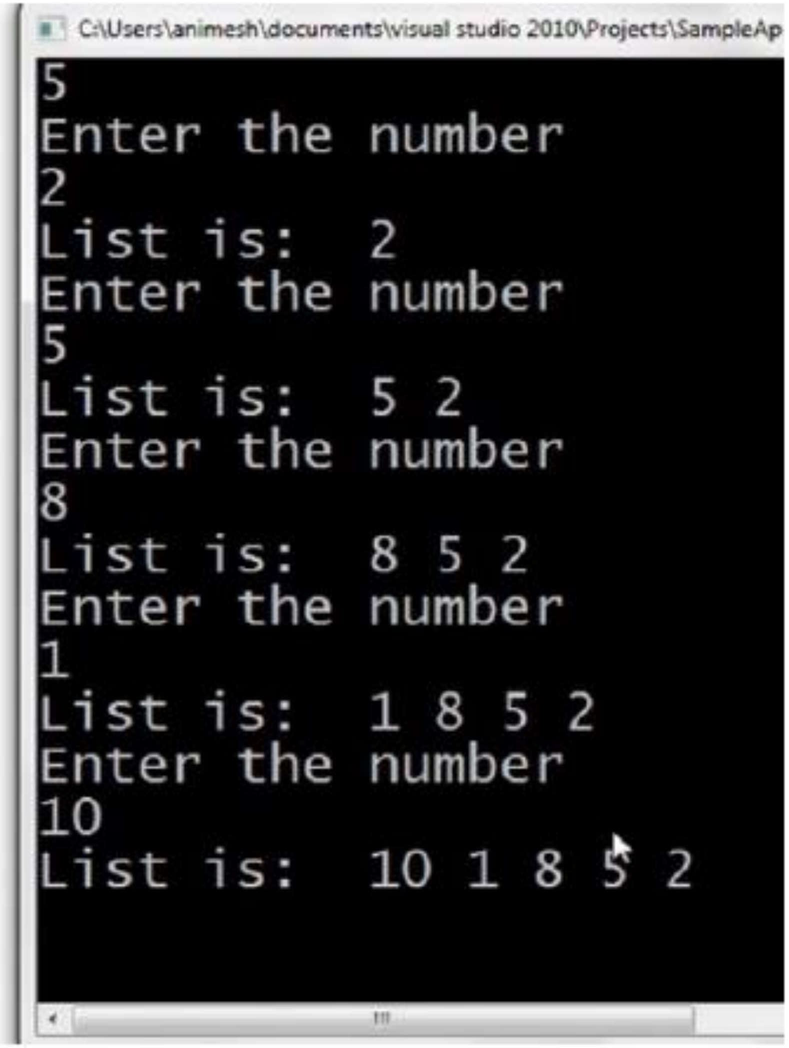
- If the list is empty as above, we want head to point to this new node.
- If the list is not empty, by the line temp->next = head, the new node points to the address 100.
- Next, to cut the link from head to 100, we have the line head = temp.

Implementing the print function

```
struct Node* head; // global variable, can be accessed anywher
void Insert(int x)
{
    struct Node* temp = (Node*)malloc(sizeof(struct Node));
    temp->data = x;
    temp->next = head;
    head = temp;
}
void Print()
{
    struct Node* temp = head;
    printf("List is: ");
    while(temp != NULL)
    {
        printf(" %d",temp->data);
        temp= temp->next;
    }
    printf("\n");
}
```

Implementing the print function

```c
void Print() {
    struct Node* temp = head;
    printf("List is: ");
    while(temp != NULL)
    {
        printf(" %d",temp->data);
        temp= temp->next;
    }
    printf("\n");
}
int main() {
    head = NULL; // empty list;
    printf("How many numbers?\n");
    int n,i,x;
    scanf("%d",&n);
    for(i = 0;i<n;i++){
        printf("Enter the number \n");
        scanf("%d",&x);
        Insert(x);
        Print();
    }
}
```



Note that each newly entered number is stored as the beginning of the list.

Implementing the print function

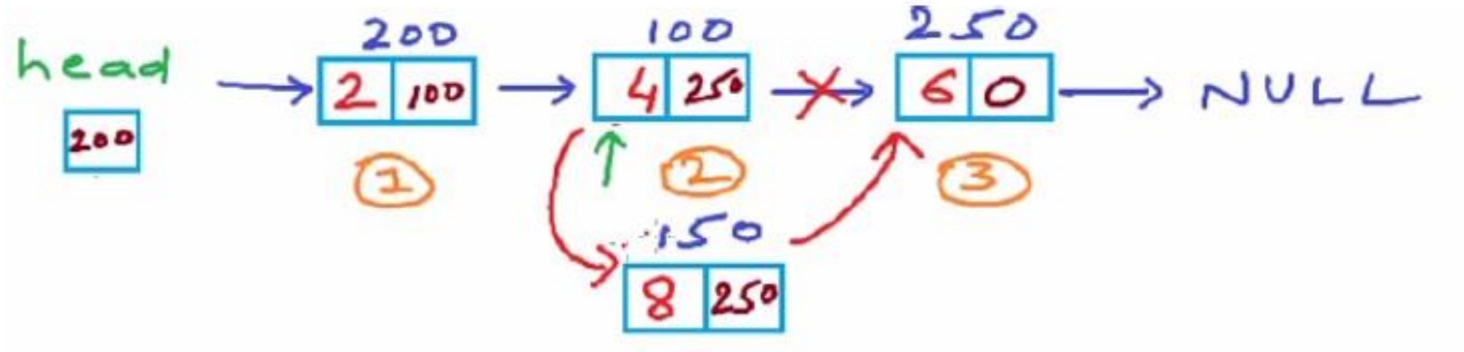
```
void Print(Node* head) {
    printf("List is: ");
    while(head != NULL)
    {
        printf(" %d",head->data);
        head= head->next;
    }
    printf("\n");
}
int main() {
    Node* head = NULL; // empty list;
    printf("How many numbers?\n");
    int n,i,x;
    scanf("%d",&n);
    for(i = 0;i<n;i++){
        printf("Enter the number \n");
        scanf("%d",&x);
        head = Insert(head,x);
        Print(head);
    }
}
```

Head will be passed to print as local variable and therefore an argument of the print function.

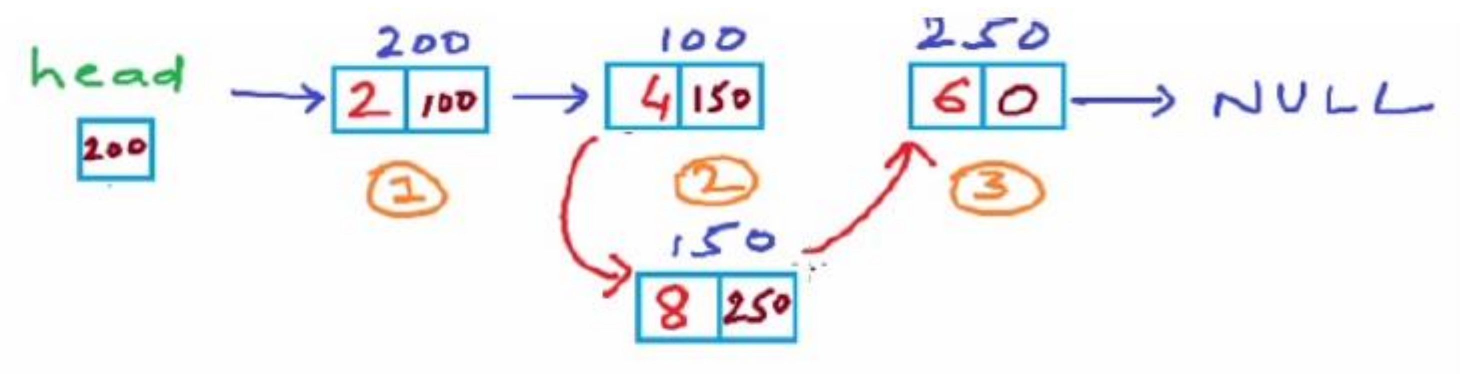
Since head is a local variable in print, we can change the value of head inside the print function instead of defining a temporary variable temp.

The insert method also passes the return as head.

Inserting a node at the n'th position



Say, we want to insert a value 8 at 3rd position.
The link field of the new node should be 250 and the link field of the second node should be the address of the new node as shown below.



```
void Insert(int data,int n) {
    Node* temp1 = new Node();
    temp1->data = data;
    temp1->next = NULL;
    if(n == 1) {
        temp1->next = head;
        head = temp1;
        return;
    }
    Node* temp2 = head;
    for(int i =0;i<n-2;i++) {
        temp2 = temp2->next;
    }
    temp1->next = temp2->next;
    temp2->next = temp1;
}
```

In this implementation of Insert function, the new node is defined using C++ syntax (without using malloc).

Finally, we set the link of the new node to the link of the (n-1)st node and then we set the link of the (n-1)st node to the new node.

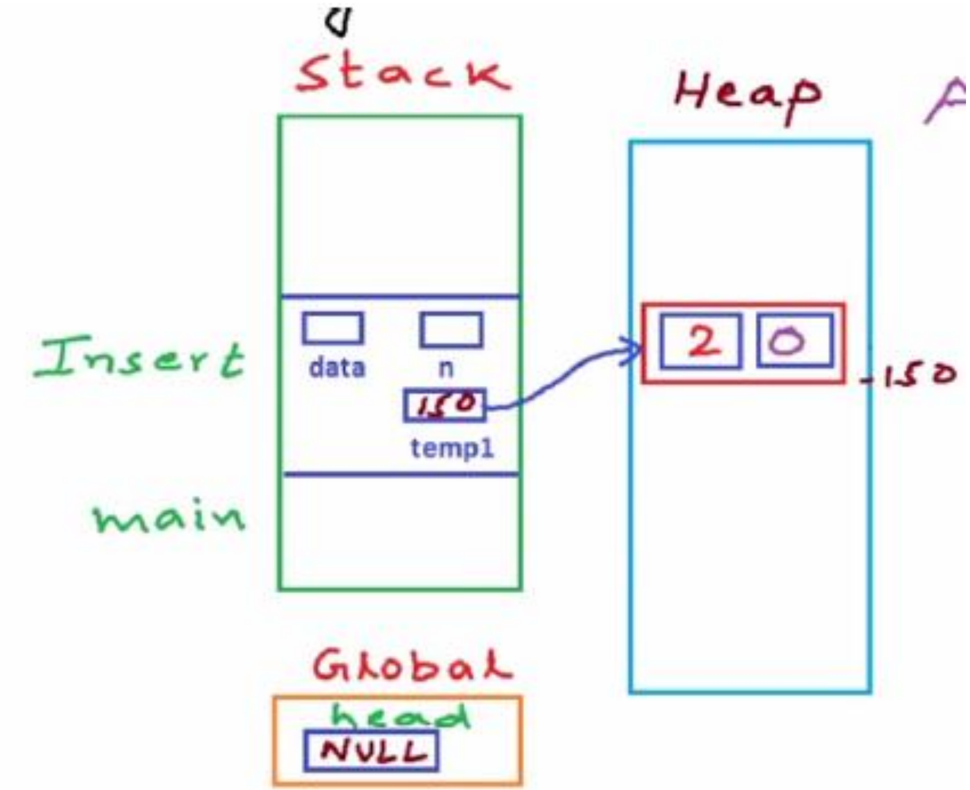
```c
struct Node {
    int data;
    struct Node* next;
};
struct Node* head;
void Insert(int data,int n);
void Print();
int main() {
    head = NULL; //empty list
    Insert(2,1); //List: 2
    Insert(3,2); //List: 2,3
    Insert(4,1); //List: 4,2,3
    Insert(5,2); //List: 4,5,2,3
    Print();
}
```

In the implementation above, again the pointer to the beginning of the linked list, called head, is defined as a global variable.

The insert function takes the value of the new node and the position we want to insert the new node.
Print will print all the numbers in the linked list.

Location of data in the memory

```
void Insert(int data,int n) {
    Node* temp1 = new Node();
    temp1->data = data;
    temp1->next = NULL;
    if(n == 1) {
        temp1->next = head;
        head = temp1;
        return;
    }
    Node* temp2 = head;
    for(int i =0;i<n-2;i++) {
        temp2 = temp2->next;
    }
    temp1->next = temp2->next;
    temp2->next = temp1;
}
int main() {
    head = NULL; //empty list
    Insert(2,1); //List: 2
    Insert(3,2); //List: 2,3
    Insert(4,1); //List: 4,2,3
    Insert(5,2); //List: 4,5,2,3
    Print();
}
```



When we store something in the heap using **new** or **malloc**, we do not have a variable name for this and we can only access it through a pointer variable as seen above.

Deleting a node at n’th position

```c
//Linked List: Delete a node at nth position
#include<stdio.h>
#include<stdlib.h>
struct Node {
    int data;
    struct Node* next;
};
struct Node* head; // global
void Insert(int data); // insert an integer at end of list
void Print();  // print all elements in the list
void Delete(int n) // Delete node at position n
int main()
{
}
```

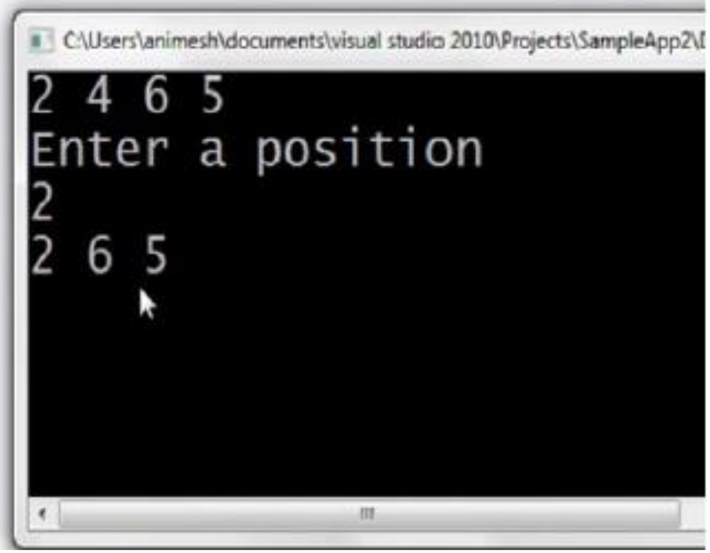
The user is asked to enter a position and the program will delete the node at this particular position.

```c
int main()
{
    head = NULL; // empty list
    Insert(2);
    Insert(4);
    Insert(6);
    Insert(5); //List : 2,4,6,5
    Print();
    int n;
    printf("Enter a position\n");
    scanf("%d",&n);
    Delete(n);
    Print();
}
```

```c
// Deletes node at position n
void Delete(int n)
{
    struct Node* temp1 = head;
    if(n ==1){
        head = temp1->next; //head now points to second node.
        free(temp1);
        return;
    }
    int i;
    for(i = 0;i<n-2;i++)
        temp1 = temp1->next;
    // temp1 points to (n-1)th Node
    struct Node* temp2 = temp1->next; // nth Node
    temp1->next = temp2->next; // (n+1)th Node
    free(temp2); //delete temp2;
}
```

In the first case, we handle the case when there is a node before the node we want to delete.
Create a temporary variable temp1 and point this to head.
Create a temporary variable temp2 that points to the nth node.

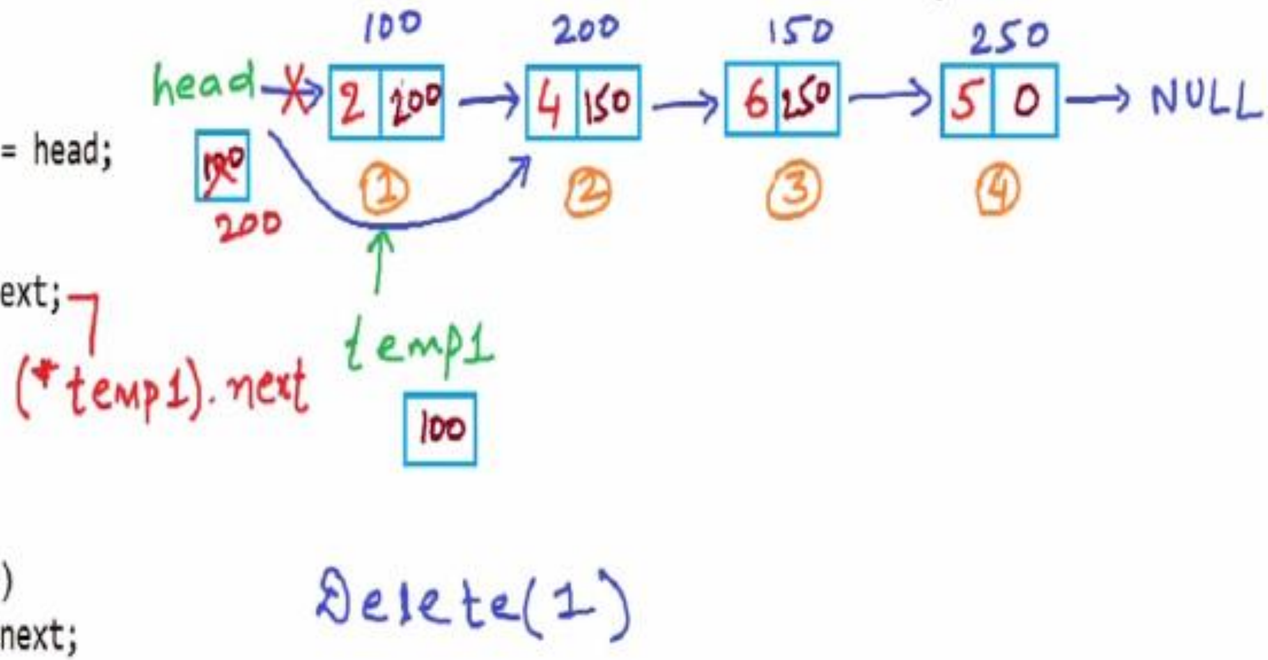
```c
int main()
{
    head = NULL; // empty list
    Insert(2);
    Insert(4);
    Insert(6);
    Insert(5); //List : 2,4,6,5
    Print();
    int n;
    printf("Enter a position\n");
    scanf("%d",&n);
    Delete(n);
    Print();
}
```



Example

Linked List: Delete a node at nth position

```c
void Delete(int n)
{
    struct Node* temp1 = head;

    if(n ==1){
        head = temp1->next;
        free(temp1);
        return;
    }
    int i;
    for(i = 0;i<n-2;i++)
        temp1 = temp1->next;

    struct Node* temp2 = temp1->next;
    temp1->next = temp2->next;
    free(temp2);
}
```

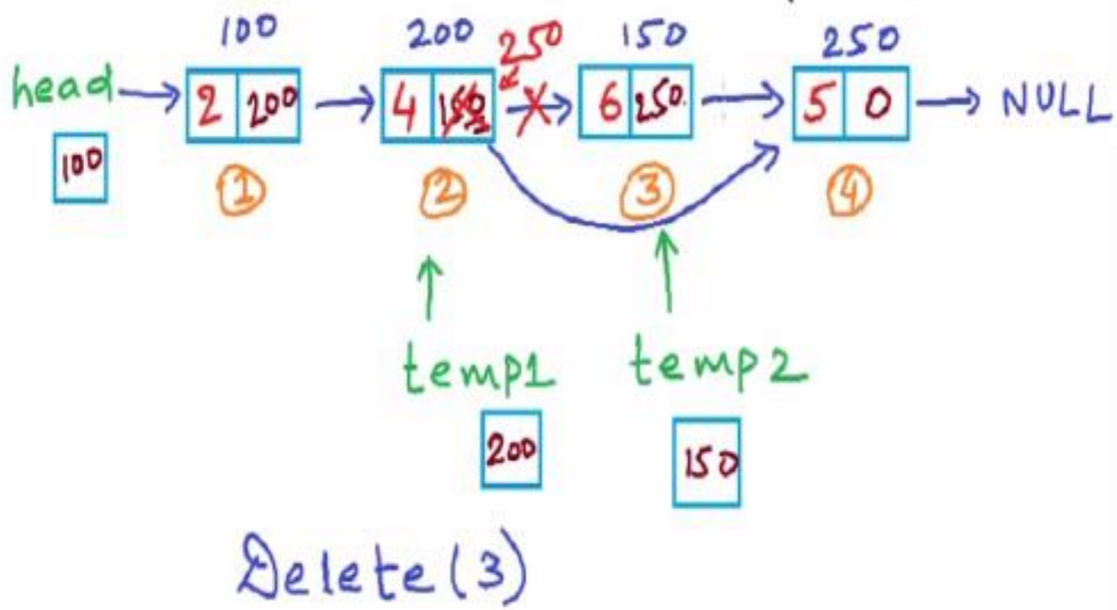


temp1 points to the first node (first position).

Example

Linked List: Delete a node at nth position

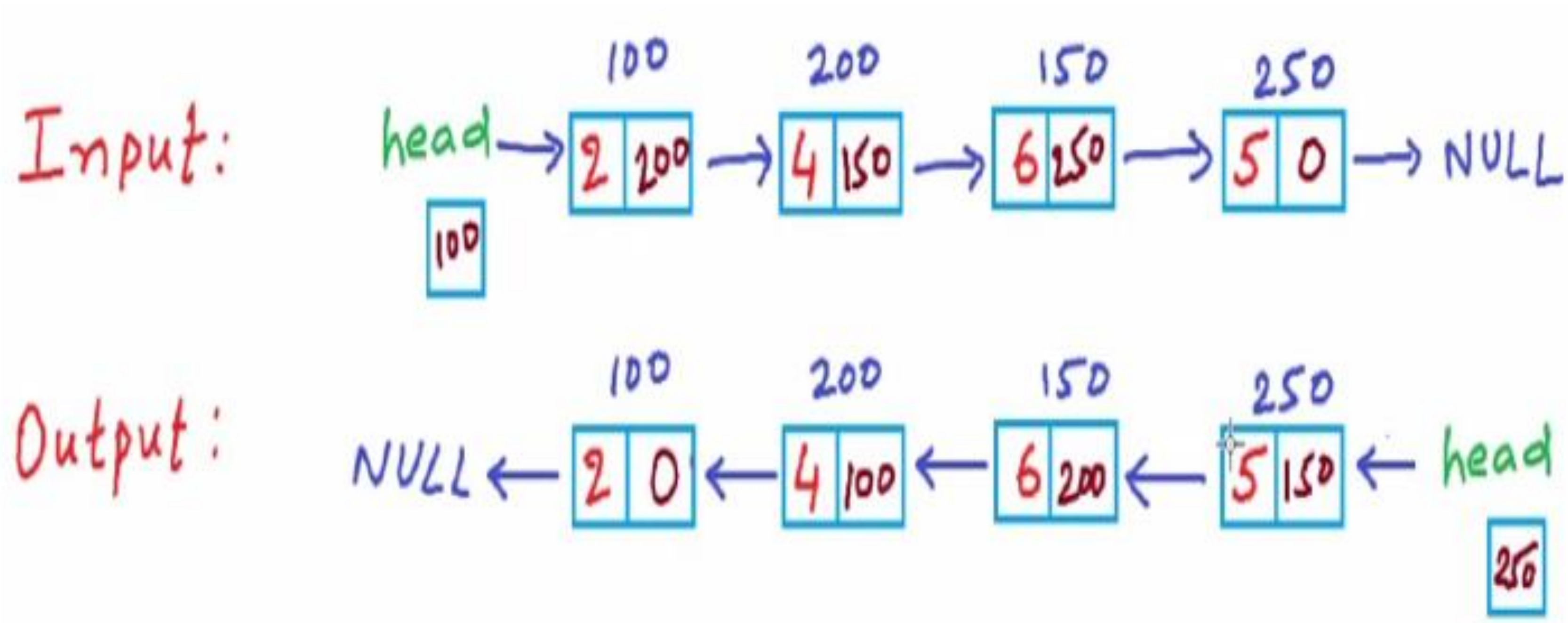
```
void Delete(int n)
{
    struct Node* temp1 = head;

    if(n ==1){
        head = temp1->next;
        free(temp1);
        return;
    }
    int i;
    for(i = 0;i<n-2;i++)
        temp1 = temp1->next;

    struct Node* temp2 = temp1->next;
    temp1->next = temp2->next;
    free(temp2);
}
```



In the final step, temp1 points to the second node and temp2 points to the third node.

At the end, the address stored by temp1, which is 150, changes to temp2.next, which is 250.

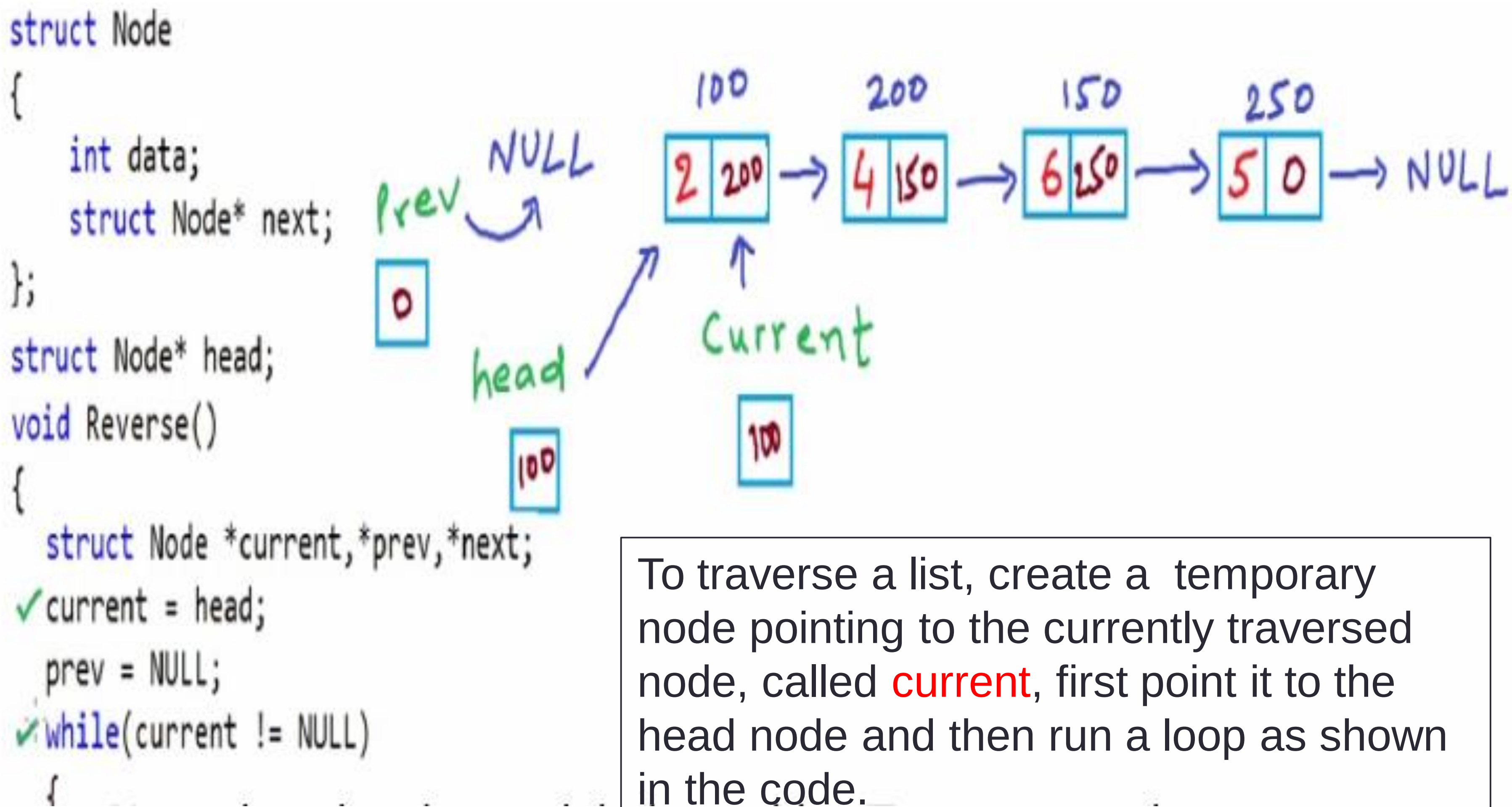
Reverse a linked list using iterative method



Links should be changed. Head node should point to the node at address 250. For the first node, we cut the link from head and build a link to NULL.

Two solutions: iterative approach and recursion.

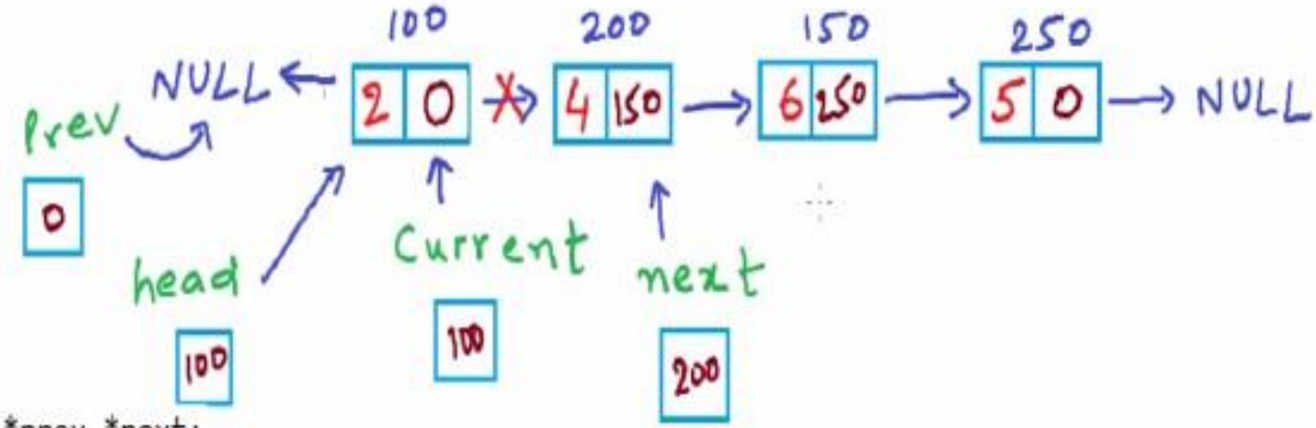
In the iterative solution, we write a loop that traverses each node and make it point to the preceding node instead of the next node.



To traverse a list, create a temporary node pointing to the currently traversed node, called current, first point it to the head node and then run a loop as shown in the code.

We will have to keep track of the previous node of each node while traversing. Call this temporary node prev.

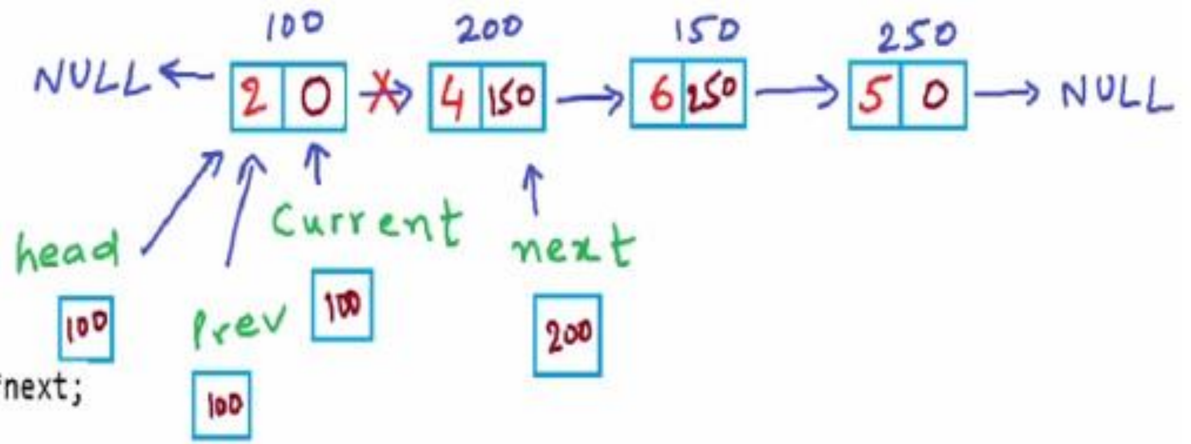
```
struct Node
{
    int data;
    struct Node* next;
};
struct Node* head;
void Reverse()
{
  struct Node *current,*prev,*next;
  current = head;
  prev = NULL;
  while(current != NULL)
  {
      next = current->next;
      current->next = prev;
```



And at each step of the traversal, we need to store the address of the old next node of the current node using a temporary variable, otherwise we lose this link. So, call this temporary variable next.

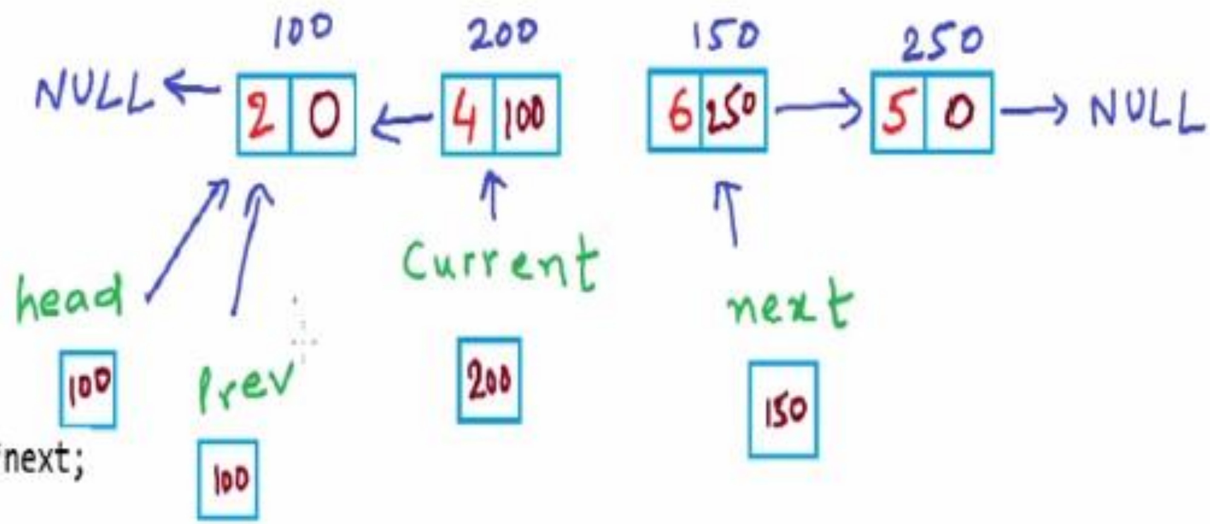
Initially, prev points to NULL, current points to the first node. After the first iteration of the while loop, next points to the second node and the address field of first node stores NULL after using the dereferencing current->next=prev.

```
struct Node
{
    int data;
    struct Node* next;
};
struct Node* head;
void Reverse()
{
  struct Node *current,*prev,*next;
 ✓current = head;
  prev = NULL;
 ✓while(current != NULL)
  {
      next = current->next;
      current->next = prev;
      prev = current;
      current = next;
```

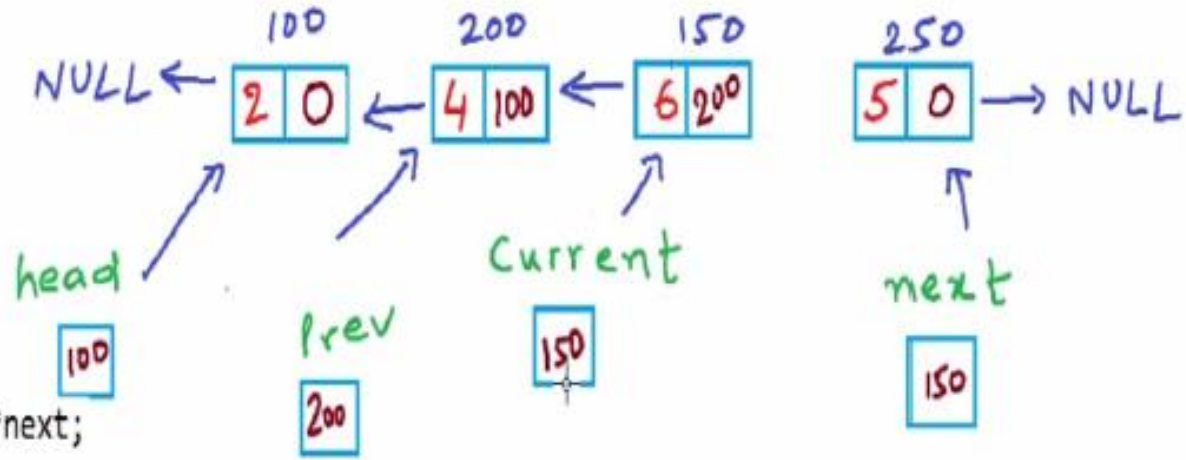


In the last two lines of the while loop, we update where prev and current are pointing to complete the traversal of one node.
Note that the next in current->next and the local variable next are different variables!!!

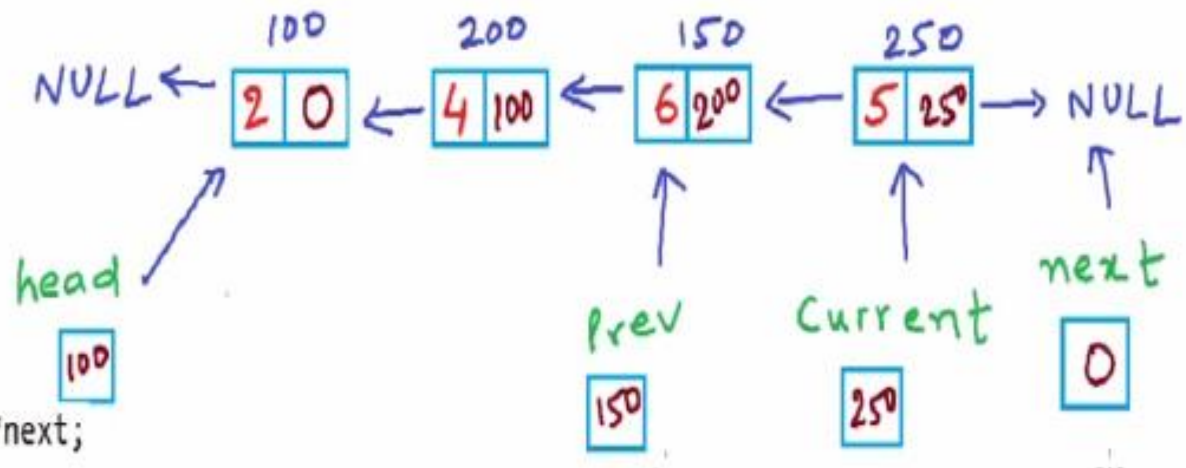
```
struct Node
{
    int data;
    struct Node* next;
};
struct Node* head;
void Reverse()
{
   struct Node *current,*prev,*next;
   current = head;
   prev = NULL;
   while(current != NULL)
   {
      next = current->next;
      current->next = prev;
      prev = current; ✓
      current = next; ✓
   }
```



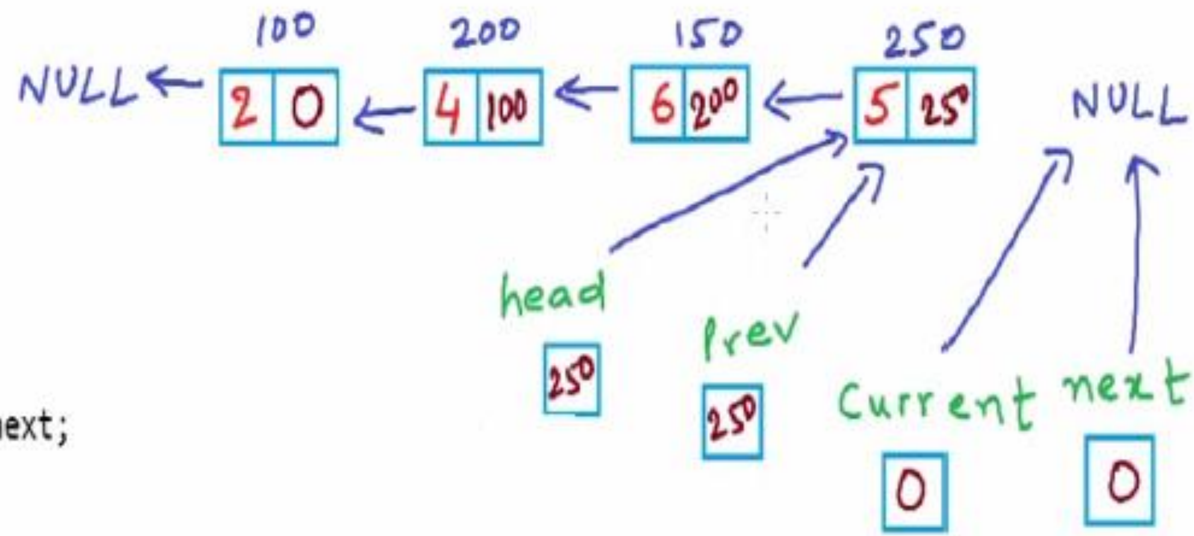
```
struct Node
{
    int data;
    struct Node* next;
};
struct Node* head;
void Reverse()
{
  struct Node *current,*prev,*next;
  current = head;
  prev = NULL;
  while(current != NULL)
  {
     next = current->next;
     current->next = prev;
     prev = current; ✓
     current = next; ✓
  }
```



```
struct Node
{
    int data;
    struct Node* next;
};
struct Node* head;
void Reverse()
{
  struct Node *current,*prev,*next;
  current = head;
  prev = NULL;
  while(current != NULL)
  {
      next = current->next;
      current->next = prev;
      prev = current;
      current = next;
  }
```



```
struct Node
{
    int data;
    struct Node* next;
};
struct Node* head;
void Reverse()
{
  struct Node *current,*prev,*next;
  current = head;
  prev = NULL;
  while(current != NULL)
  {  next = current->next;
     current->next = prev;
     prev = current; ✓
     current = next; ✓
  }
  head = prev;
}
```



Now in this version, head is defined as a local variable in the main method. Therefore, reverse function takes the address of the head node as argument and returns the address of the head node.

```c
// Reverse a nexted list
#include<stdio.h>
#include<stdlib.h>
struct Node {
    int data;
    struct Node* next;
};
struct Node* Reverse(struct Node* head) {
    struct Node *current,*prev,*next;
    current = head;
    prev = NULL;
    while(current != NULL)
    {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    head = prev;
    return head;
```

```
int main()
{
    struct Node* head =NULL;  // local variable
    head = Insert(head,2);   // Insert: struct Node* Insert
    head = Insert(head,4);
    head = Insert(head,6);
    head = Insert(head,8);
    Print(head);
    head = Reverse(head);
    Print(head);
}
```

The insert function takes two arguments: the address of the head node and the data to be inserted.
The insert function returns the address of the head node. Print function prints the elements in the list.

Print elements of a linked list in forward and reverse order using recursion

The print function takes the address of a node, so the argument is of type pointer.

For now forget about how we input the linked list, assume it is already entered. Printf will print the value at the node p.

Then we make a recursive call to the Print function passing the address of the next node without forgetting the exit condition for the recursion.

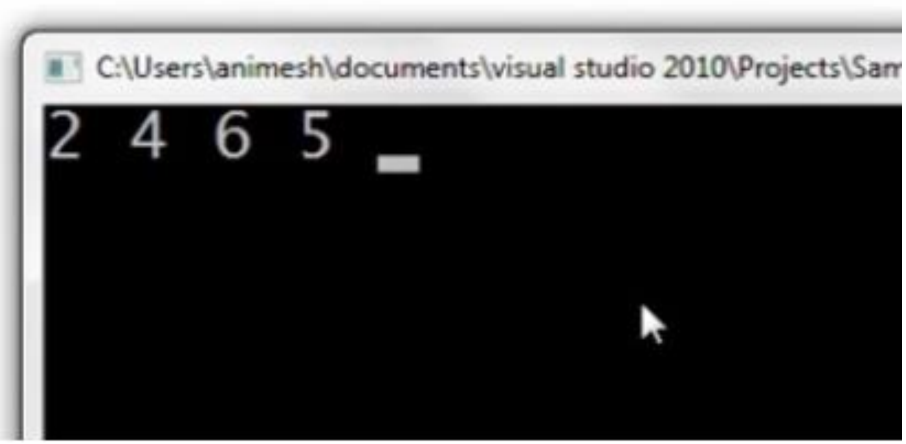
```
// Print Linked List using Recursion
#include<stdio.h>
#include<stdlib.h>
struct Node {
    int data;
    struct Node* next;
};
void Print(struct Node* p)
{
    if(p == NULL) return; // Exit condition
    printf("%d ",p->data); // First print the value in the node
    Print(p->next); // Recursive call
```

In the insert function, the insert function returns the current address of the head node after insertion. (Note that, head is a local variable in the main method.)
Here, the insert function inserts a node at the end of the list.

```
struct Node* Insert(Node* head,int data) {
    Node *temp = (struct Node*)malloc(sizeof(struct Node));
    temp->data = data;
    temp->next = NULL;
    if(head == NULL) head = temp;
    else {
        Node* temp1 = head;
        while(temp1->next != NULL) temp1 = temp1->next;
        temp1->next = temp;
    }
    return head;
}

int main()
{
    struct Node* head = NULL;  // local variable
    head = Insert(head,2);
```

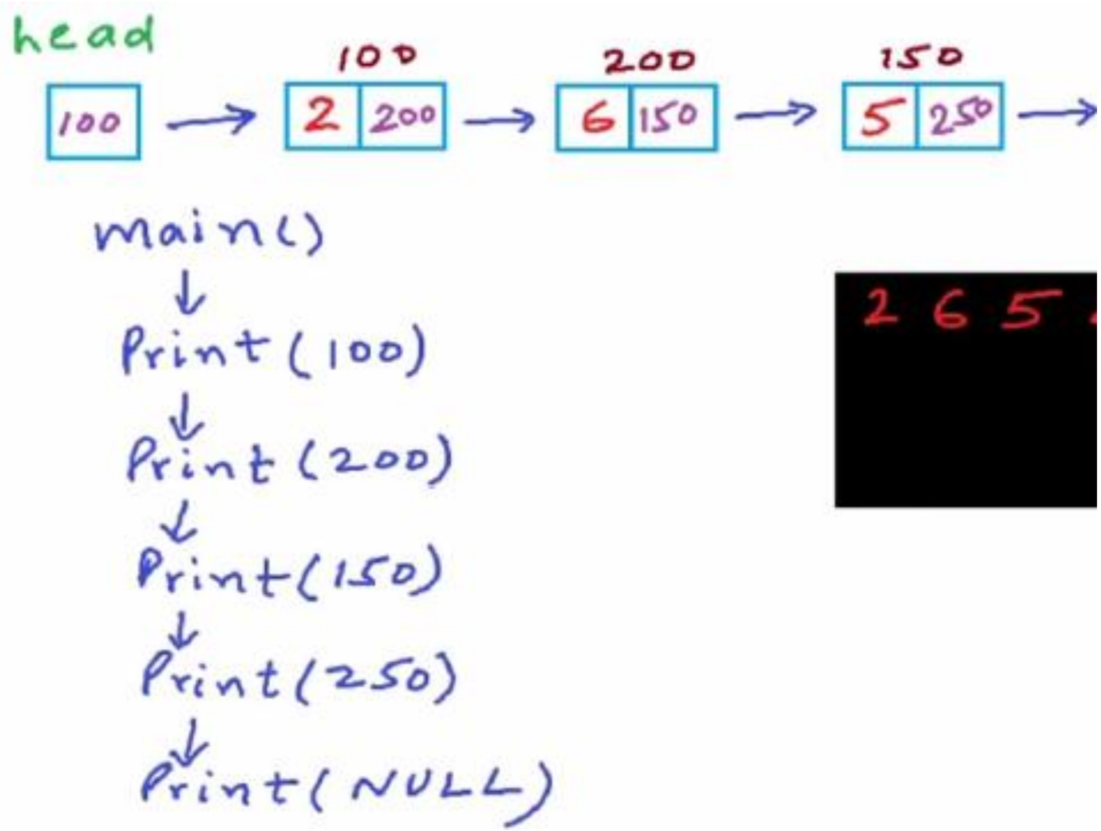
By using the recursive print function, we were able to print the linked list in forward order. See next slide for steps.

```
int main()
{
    struct Node* head = NULL;  // local variable
    head = Insert(head,2);
    head = Insert(head,4);
    head = Insert(head,6);
    head = Insert(head,5);
    Print(head);
}
```



Each time Print is visited, it prints the data stored in the address input as an argument.
The arrows showing the steps of the recursion is called a recursion tree.

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

void Print(struct Node* p)
{
    if(p == NULL)
    {
        printf("\n");
        return;
    }
    printf("%d ",p->data);
    Print(p->next);
}
```



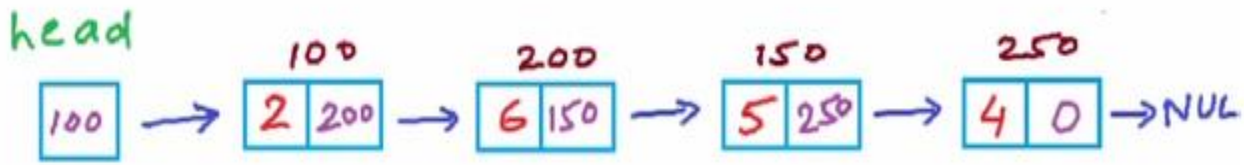
```c
// Print Linked List using Recursion
#include<stdio.h>
#include<stdlib.h>
struct Node {
    int data;
    struct Node* next;
};
void Print(struct Node* p)
{
    if(p == NULL) return; // Exit condition
    Print(p->next); // Recursive call
    printf("%d ",p->data); // First print the value in the node

}
```

```c
int main()
{
    struct Node* head = NULL;  // local variable
    head = Insert(head,2);
    head = Insert(head,4);
    head = Insert(head,6);
    head = Insert(head,5);
    Print(head);
}
```

```
C:\Users\animesh\documents\visual stu
5 6 4 2 _
```

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

void ReversePrint(struct Node* p)
{
    if(p == NULL)
    {
        return;
    }
    ReversePrint(p->next);
    printf("%d ",p->data);
}
```

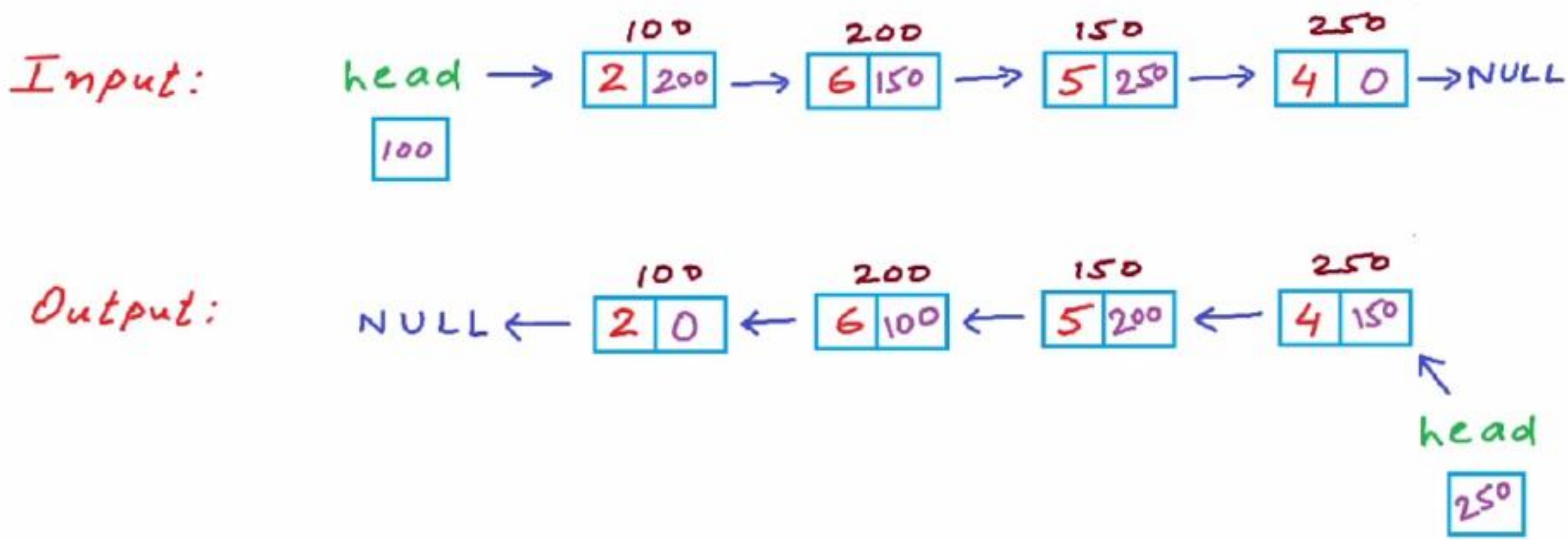
Exit condition



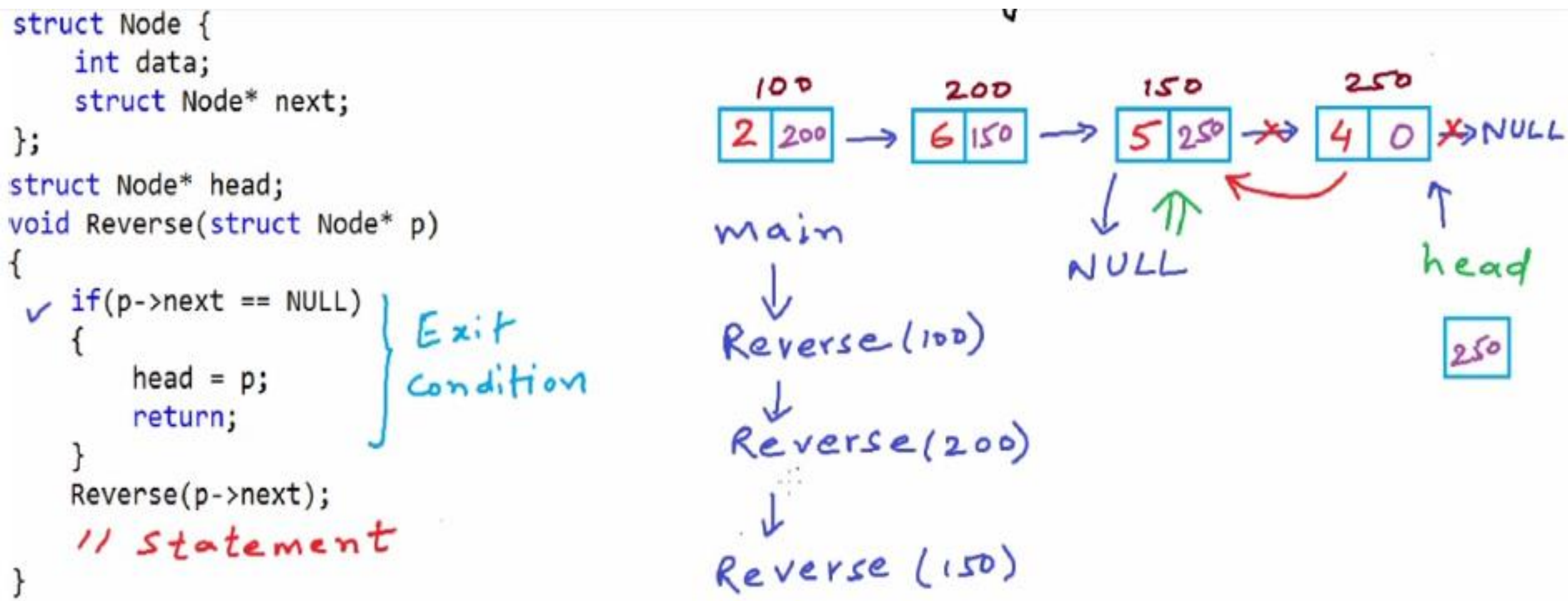
main()

Reverse a linked list using recursion

Reverse a linked list using recursion

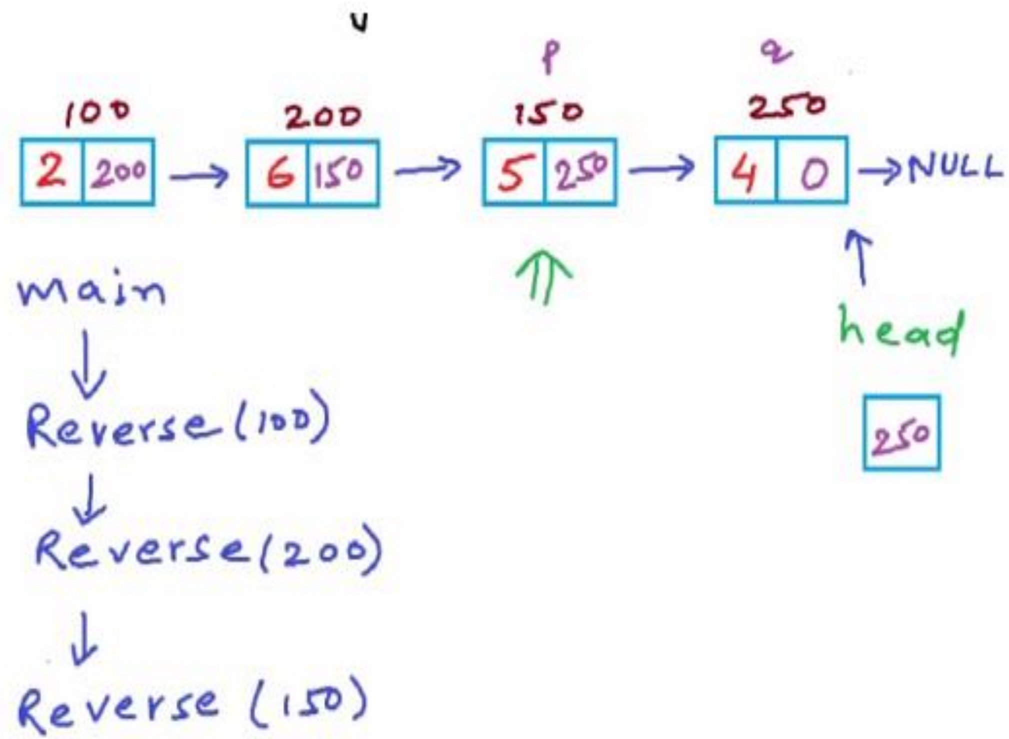


The address passed to the reverse function is the address of the first node. As soon as we reach the last node, the program modifies the head pointer to point to the fourth node.
Below, we see the links after Reverse(250) and Reverse(150) are finished.



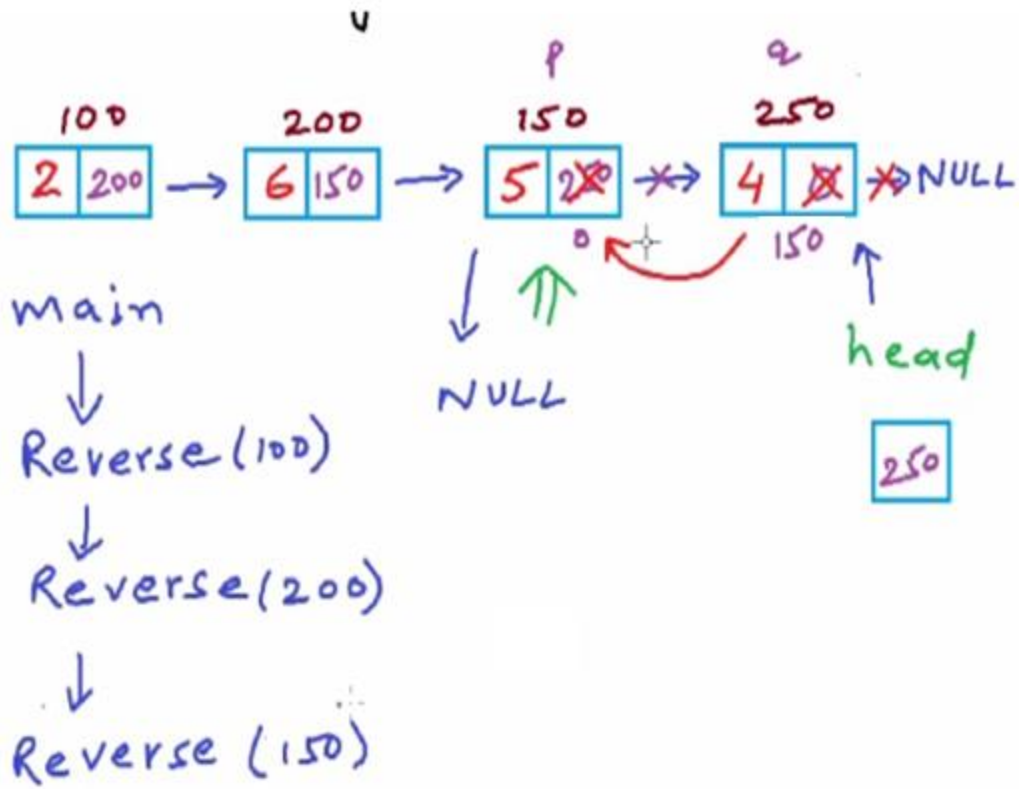
The last three lines will execute after the recursive calls are finished and we are traversing the list in backwards direction.
When Reverse(150) is executed, p would be 150 and q would be p.next. See next slide.

```
struct Node {
    int data;
    struct Node* next;
};
struct Node* head;
void Reverse(struct Node* p)
{
    if(p->next == NULL)
    {
        head = p;
        return;
    }
    Reverse(p->next);
    struct Node* q = p->next;
    q->next = p;
    p->next = NULL;
}
```



```
struct Node {
    int data;
    struct Node* next;
};
struct Node* head;
void Reverse(struct Node* p)
{
    if(p->next == NULL)
    {
        head = p;
        return;
    }
    Reverse(p->next);
    struct Node* q = p->next;
    q->next = p;
    p->next = NULL;
}
```

Exit condition



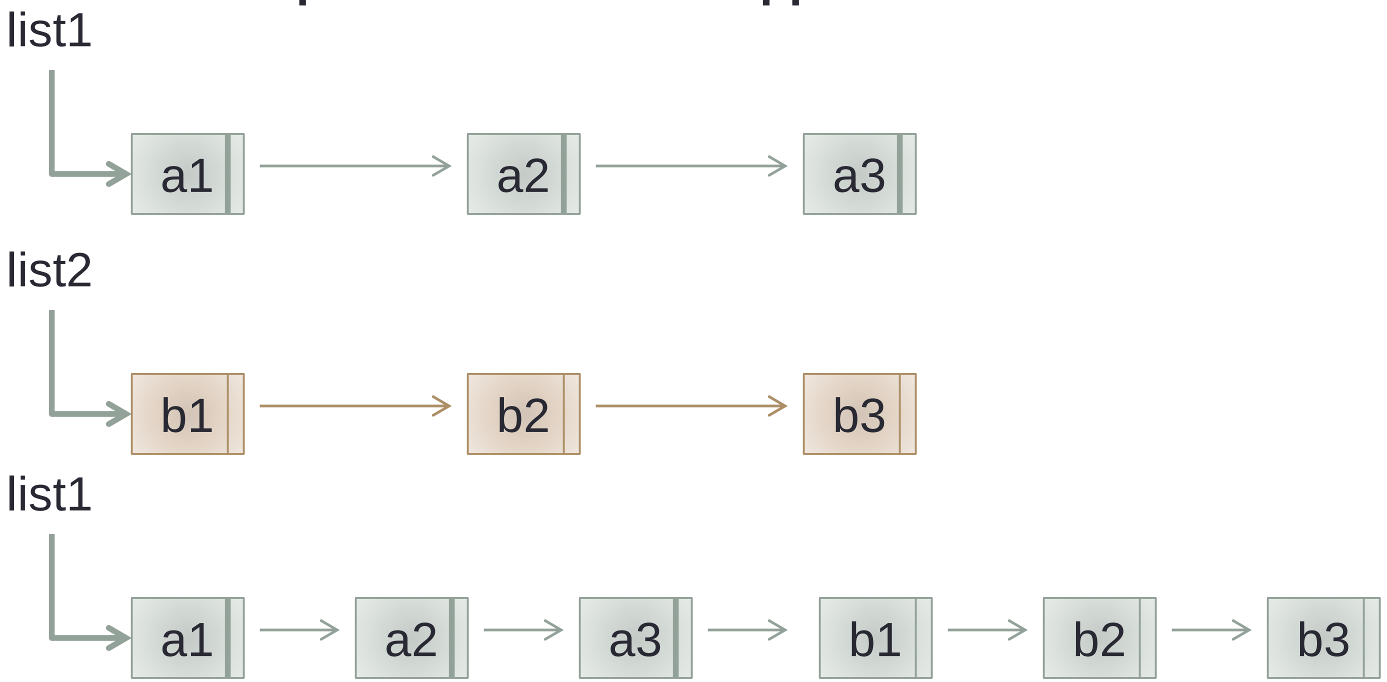
Question 1: Print the contents of a given linked list pointed with a list pointer (list *)?

```
void print_items ( listnode *  list )
{
    printf (" \n list contents: "  );
    for(;list; list= list-> link)
          printf (" %d ", list->data  );
     printf (" \n ");
}
```

Question 2: Count the number of items in a given linked list pointed with a list pointer (list *)?

```
int count_items ( listnode *  list )
{
    int n=0;
    for(;list; list= list-> link, n++);
    return n;
}
```

Question 3: list1 and list2 are two pointers pointing to two separate linked lists. Append list2 to the end of list1.




```
void appendlists ( listnode *  list1, listnode *  list2 )
{
   list * p;
   if( list1) {
       for(p=list1; p->link; p = p -> link);
       p->link = list2;
   }
   else
       list1=list2;
   list2=NULL;
}
```