Week 10 Lecture 1

XMUT-NWEN 241 - 2024 T2

Systems Programming

Felix Yan

School of Engineering and Computer Science

Victoria University of Wellington

Admin

- Assignment 3
  - Due date is 17 November

Content

- Introduction to Linked Lists

Dynamic Data Structures

- Examples of dynamic data structures

Name	*Typical Representation*
List	Nodes (data, *next)
Doubly-linked list	Nodes (data, *next, *prev)
Binary tree	Nodes (data, *left, *right)
Queue	List & *front *back
Stack	List & *top

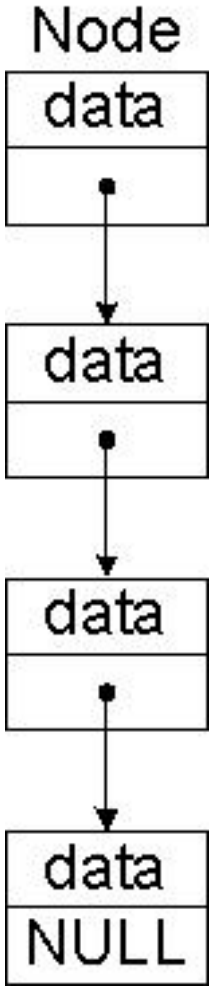
Linked Data Structures

- A structure containing a pointer to another structure of the same type (technically, it does not have to be the same type)

- Example:

```
typedef struct node
{ char data;
  struct node *next;
} Node;
```

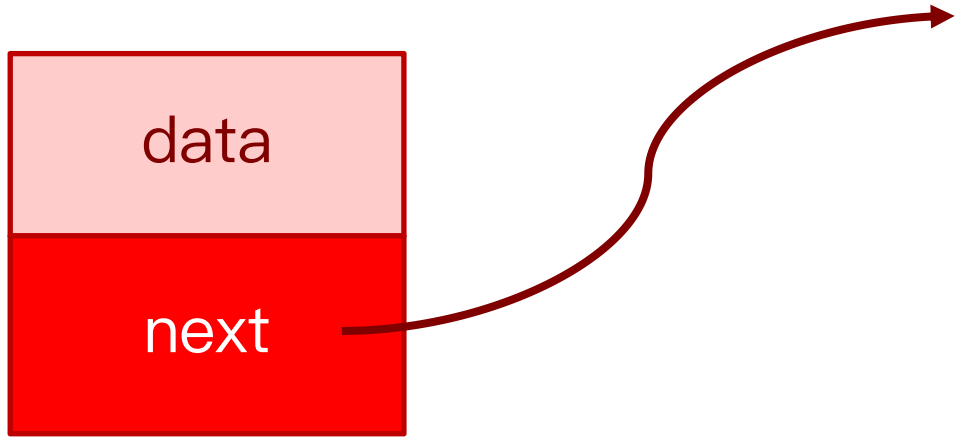
- A singly-linked list is the simplest example



Singly-Linked List (1)

- Node type definition

```
typedef struct node
{
  char data;
  struct node *next;
} Node;
```



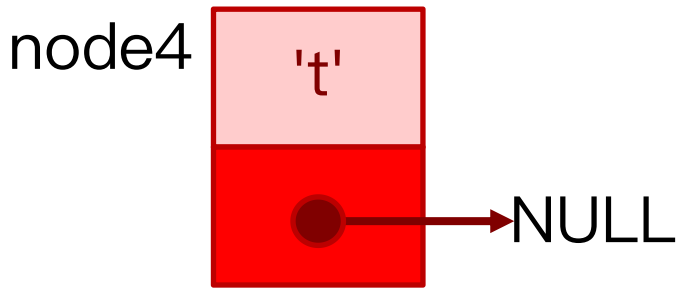
Singly-Linked List (2)

- Node type definition

```
typedef struct node
{ char data;
  struct node *next;
} Node;
```

- Node variables declaration and initialization

```
Node node4 = {'t', NULL};
Node node3 = {'s', &node4};
Node node2 = {'i', &node3};
Node node1 = {'l', &node2};
Node *head = &node1;
```



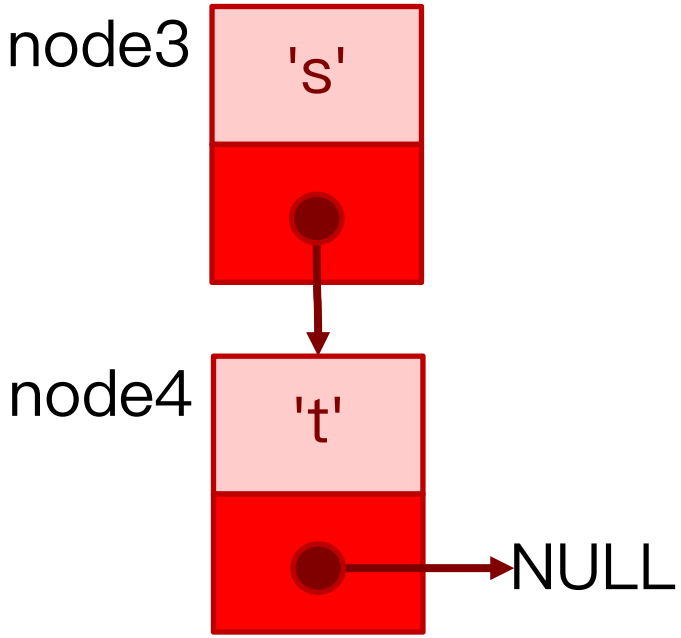
Singly-Linked List (3)

- Node type definition

```
typedef struct node
{ char data;
  struct node *next;
} Node;
```

- Node variables declaration and initialization

```
Node node4 = {'t', NULL};
Node node3 = {'s', &node4};
Node node2 = {'i', &node3};
Node node1 = {'l', &node2};
Node *head = &node1;
```



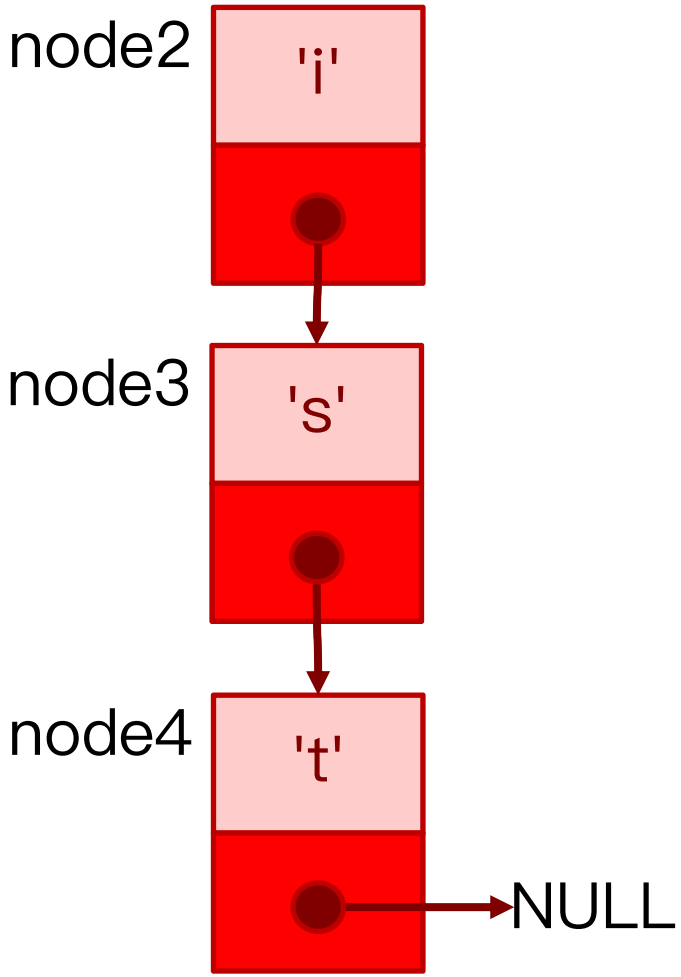
Singly-Linked List (4)

- Node type definition

```
typedef struct node
{ char data;
  struct node *next;
} Node;
```

- Node variables declaration and initialization

```
Node node4 = {'t', NULL};
Node node3 = {'s', &node4};
Node node2 = {'i', &node3};
Node node1 = {'l', &node2};
Node *head = &node1;
```



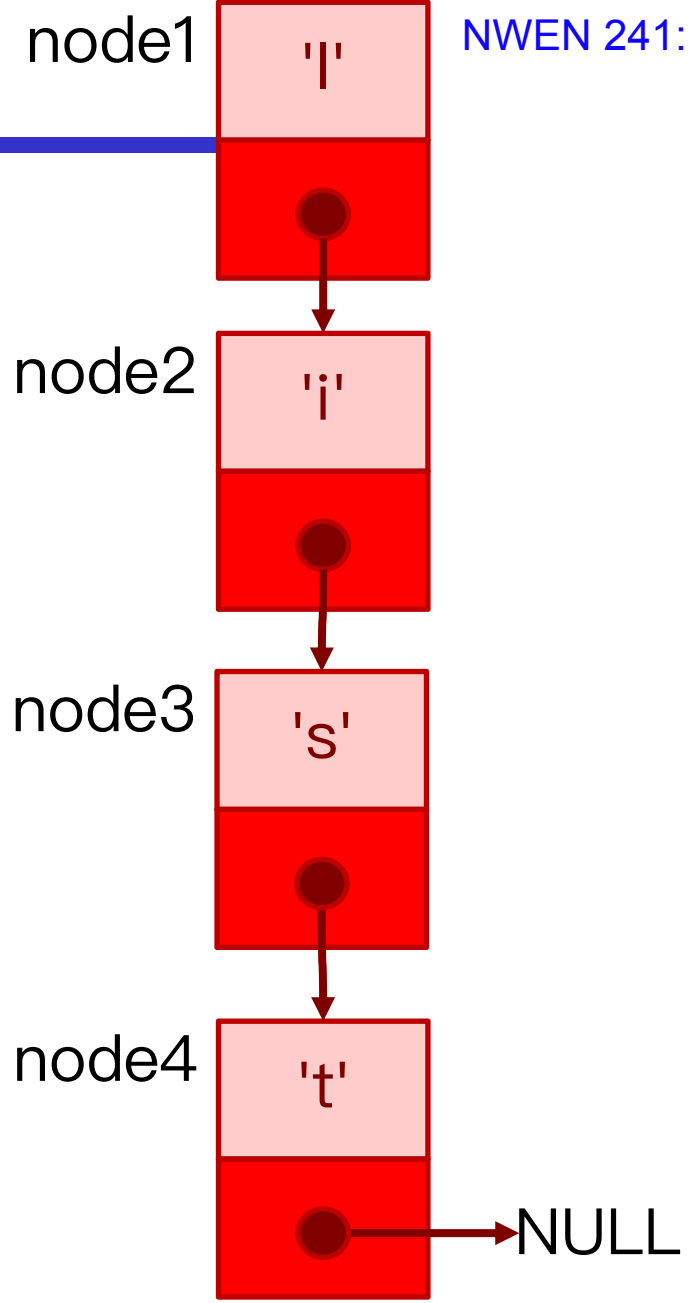
Singly-Linked List (5)

- Node type definition

```
typedef struct node
{ char data;
  struct node *next;
} Node;
```

- Node variables declaration and initialization

```
Node node4 = {'t', NULL};
Node node3 = {'s', &node4};
Node node2 = {'i', &node3};
Node node1 = {'l', &node2};
Node *head = &node1;
```



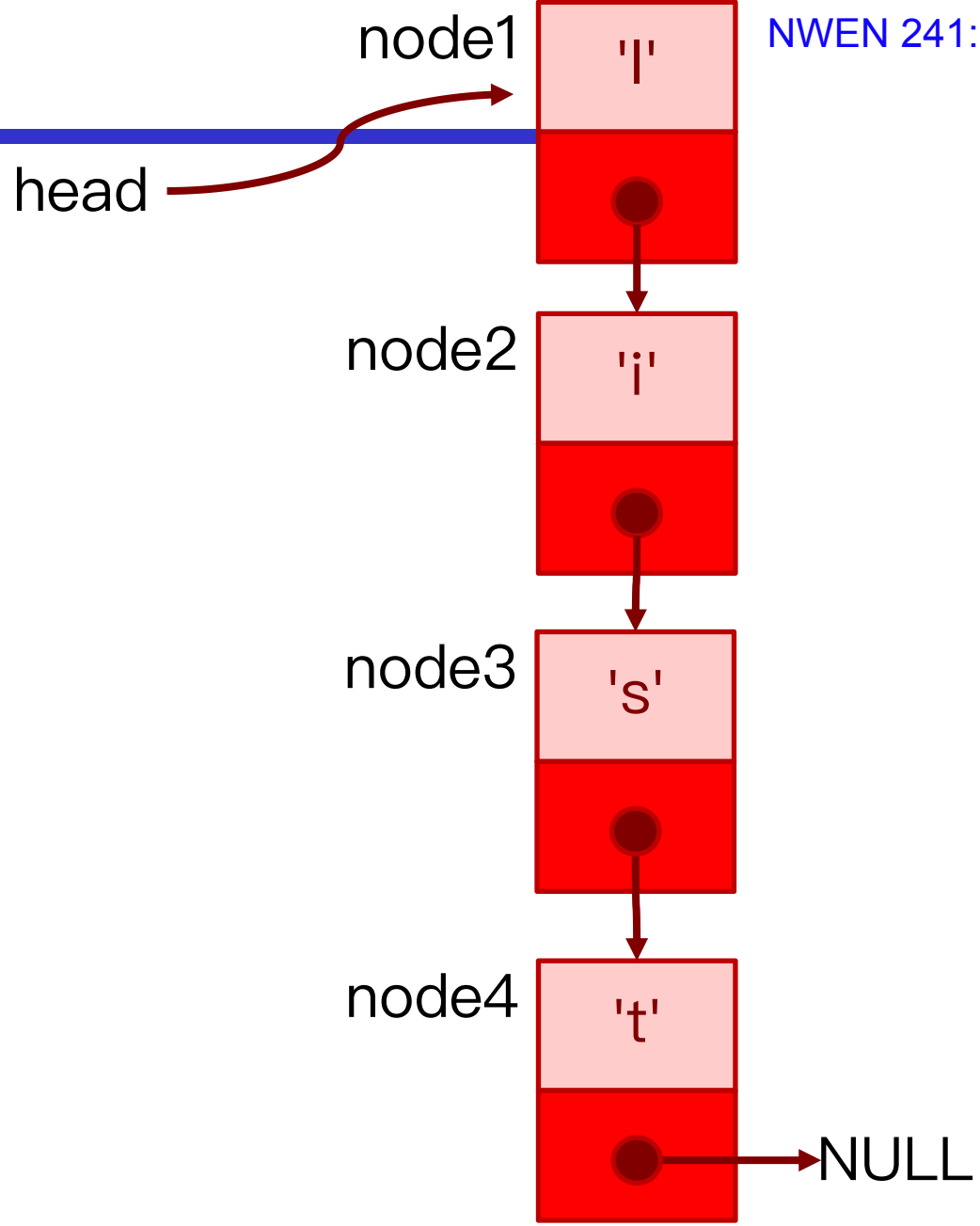
Singly-Linked List (6)

- Node type definition

```
typedef struct node
{ char data;
  struct node *next;
} Node;
```

- Node variables declaration and initialization

```
Node node4 = {'t', NULL};
Node node3 = {'s', &node4};
Node node2 = {'i', &node3};
Node node1 = {'l', &node2};
Node *head = &node1;
```



Singly-Linked List Traversal (1)

- Will always begin from head
- Visit every node until last node

- Example:

```
Node *p = head;
for( ; p != NULL; p = p->next)
  printf("%c", p->data);
```

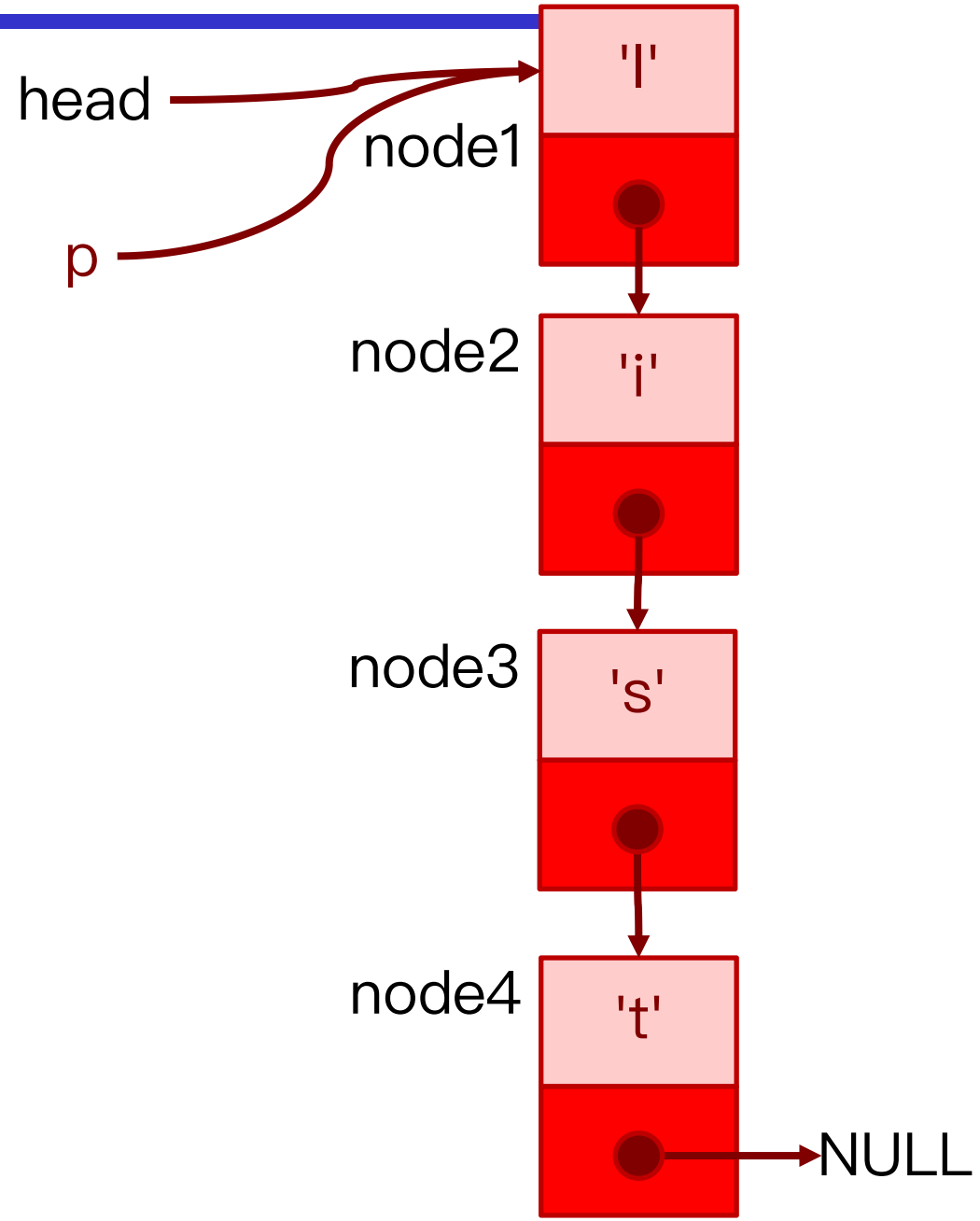
Singly-Linked List Traversal (2)

- Will always begin from head
- Visit every node until last node

- Example:

```
Node *p = head;
for( ; p != NULL; p = p->next)
  printf("%c", p->data);
```

- Output:



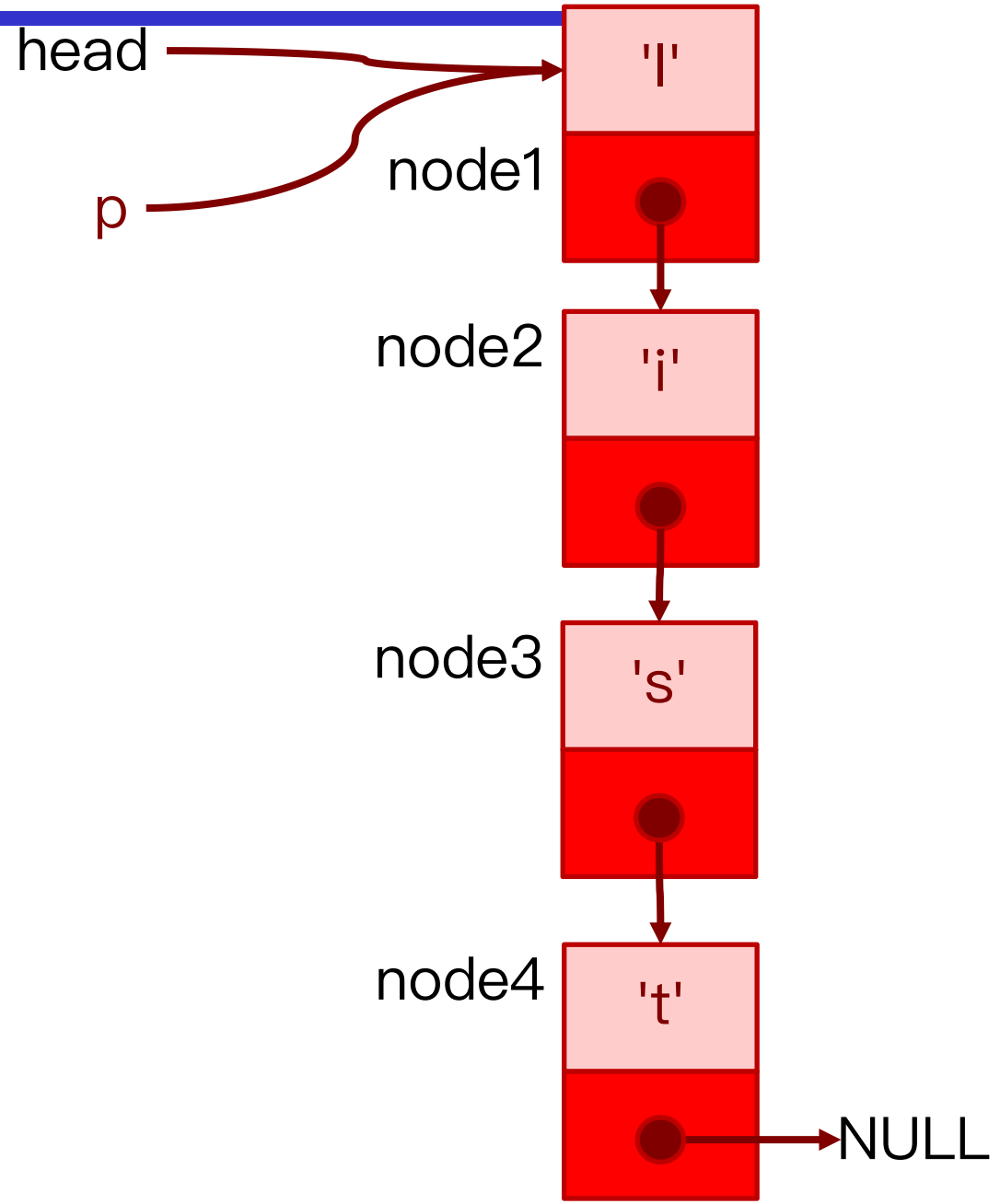
Singly-Linked List Traversal (3)

- Will always begin from head
- Visit every node until last node

- Example:

```
Node *p = head;
for( ; p != NULL; p = p->next)
  printf("%c", p->data);
```

- Output:



Singly-Linked List Traversal (4)

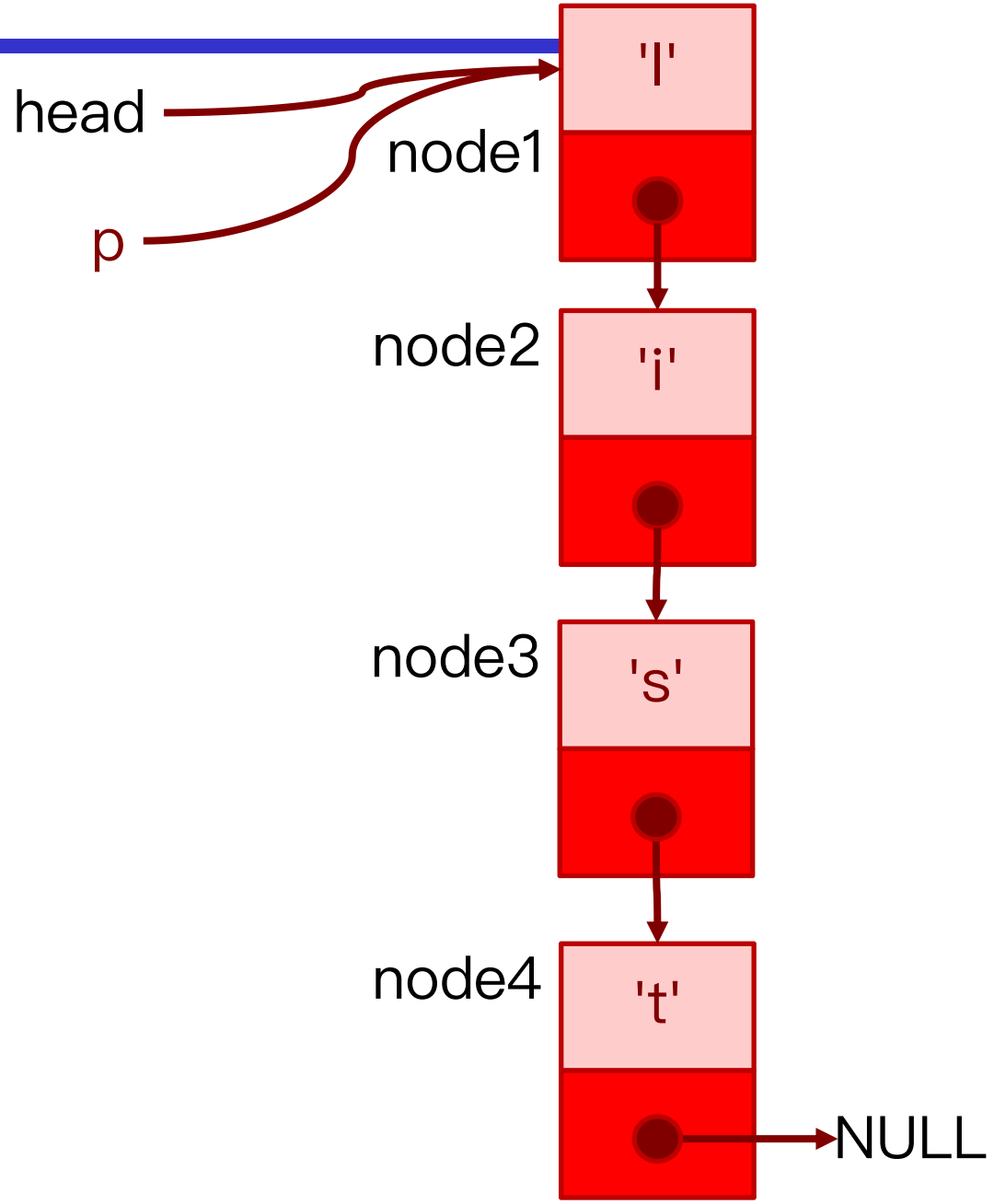
- Will always begin from head
- Visit every node until last node

- Example:

```
Node *p = head;
for( ; p != NULL; p = p->next)
  printf("%c", p->data);
```

- Output:

```
l
```



Singly-Linked List Traversal (5)

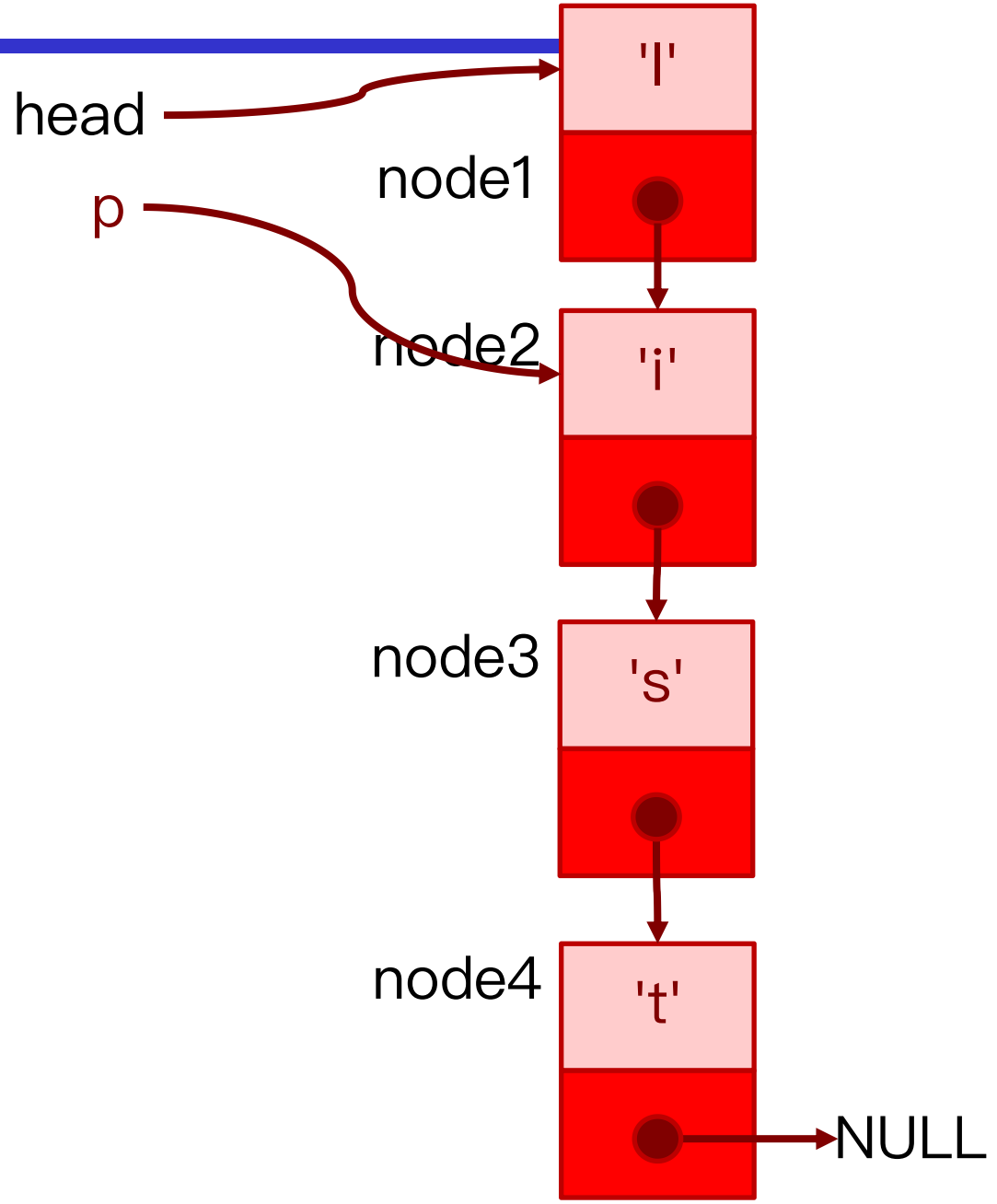
- Will always begin from head
- Visit every node until last node

- Example:

```
Node *p = head;
for( ; p != NULL; p = p->next)
  printf("%c", p->data);
```

- Output:

```
l
```



Singly-Linked List Traversal (6)

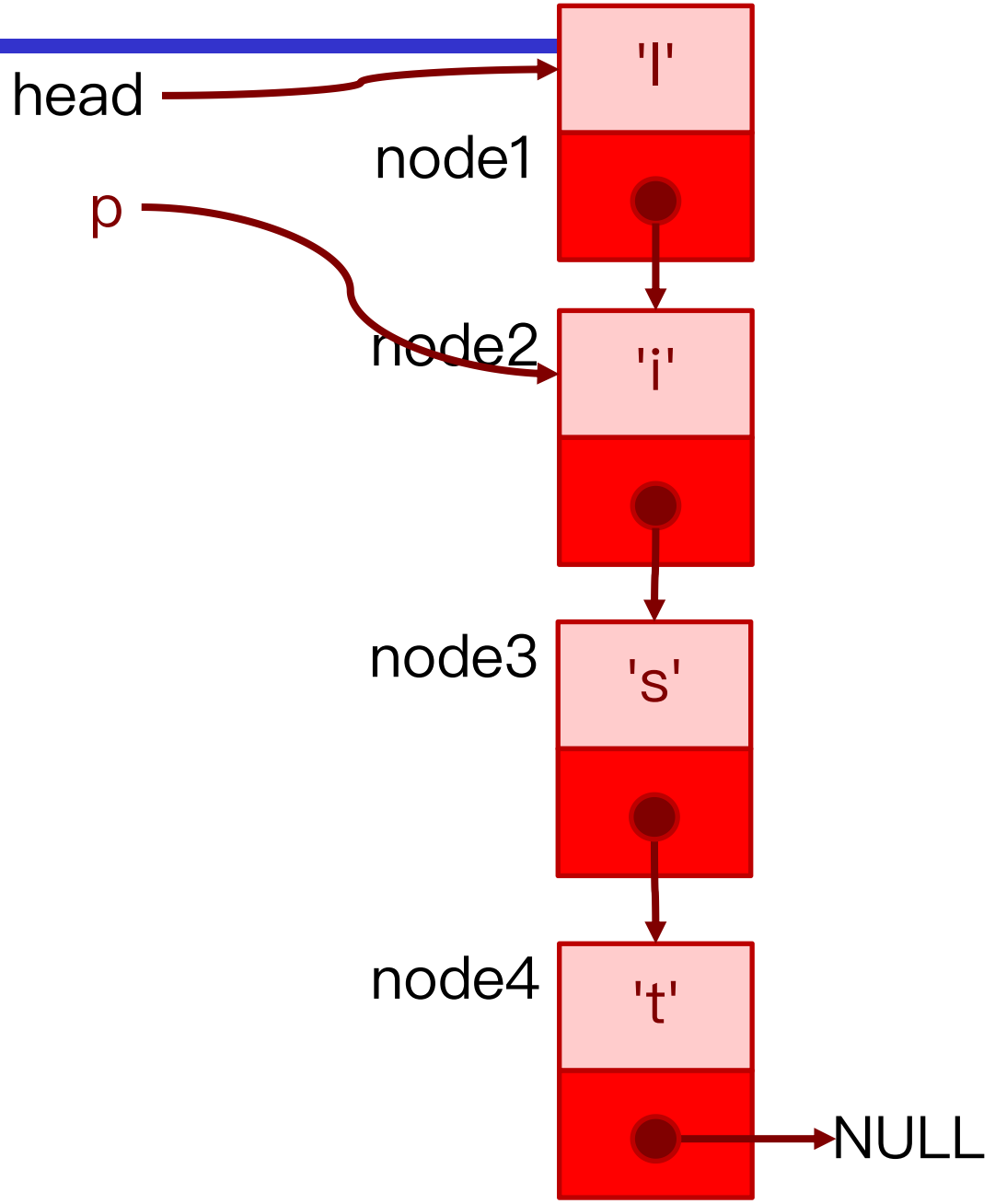
- Will always begin from head
- Visit every node until last node

- Example:

```
Node *p = head;
for( ; p != NULL; p = p->next)
  printf("%c", p->data);
```

- Output:

```
I
```



Singly-Linked List Traversal (7)

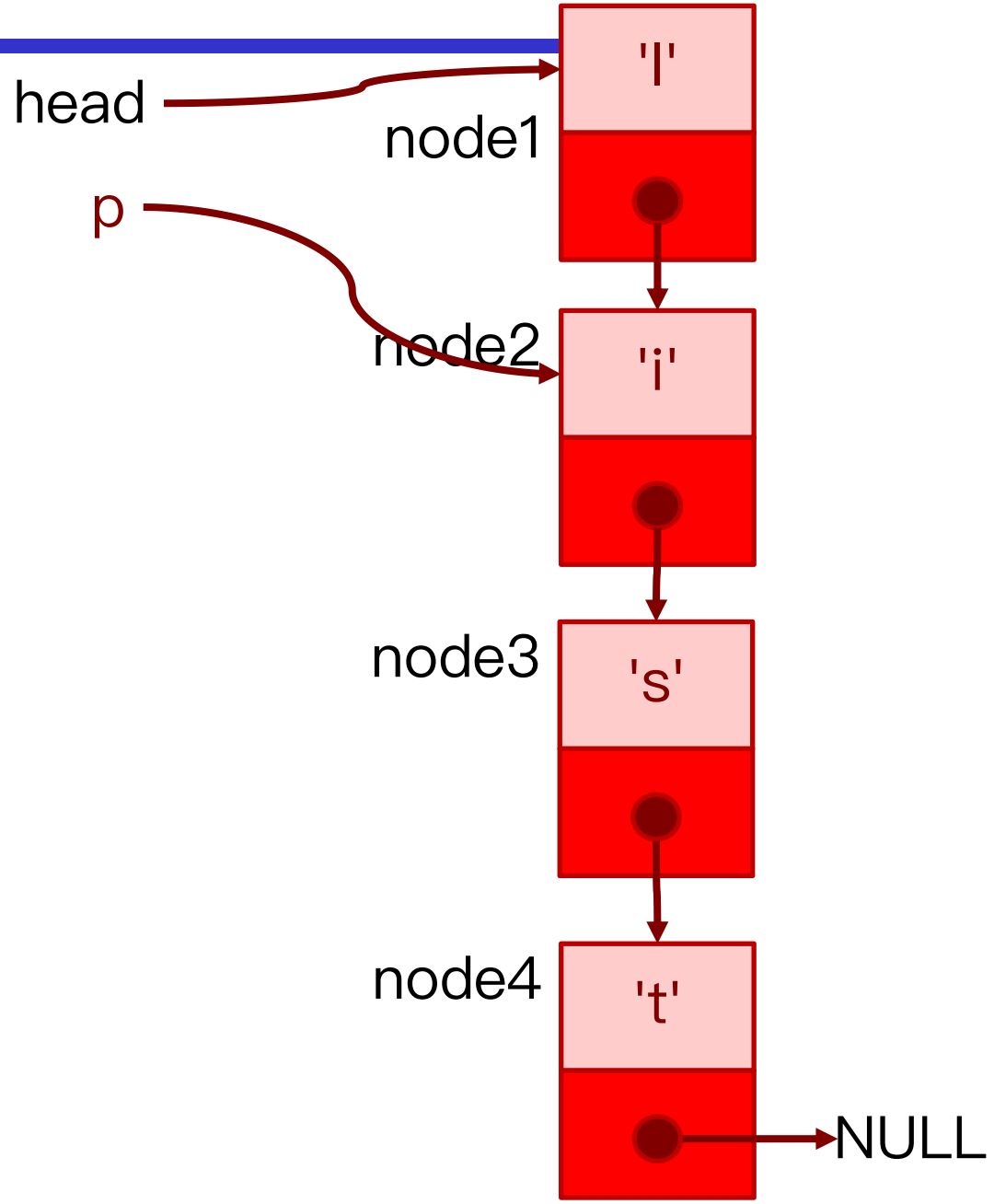
- Will always begin from head
- Visit every node until last node

- Example:

```
Node *p = head;
for( ; p != NULL; p = p->next)
  printf("%c", p->data);
```

- Output:

```
li
```



Singly-Linked List Traversal (8)

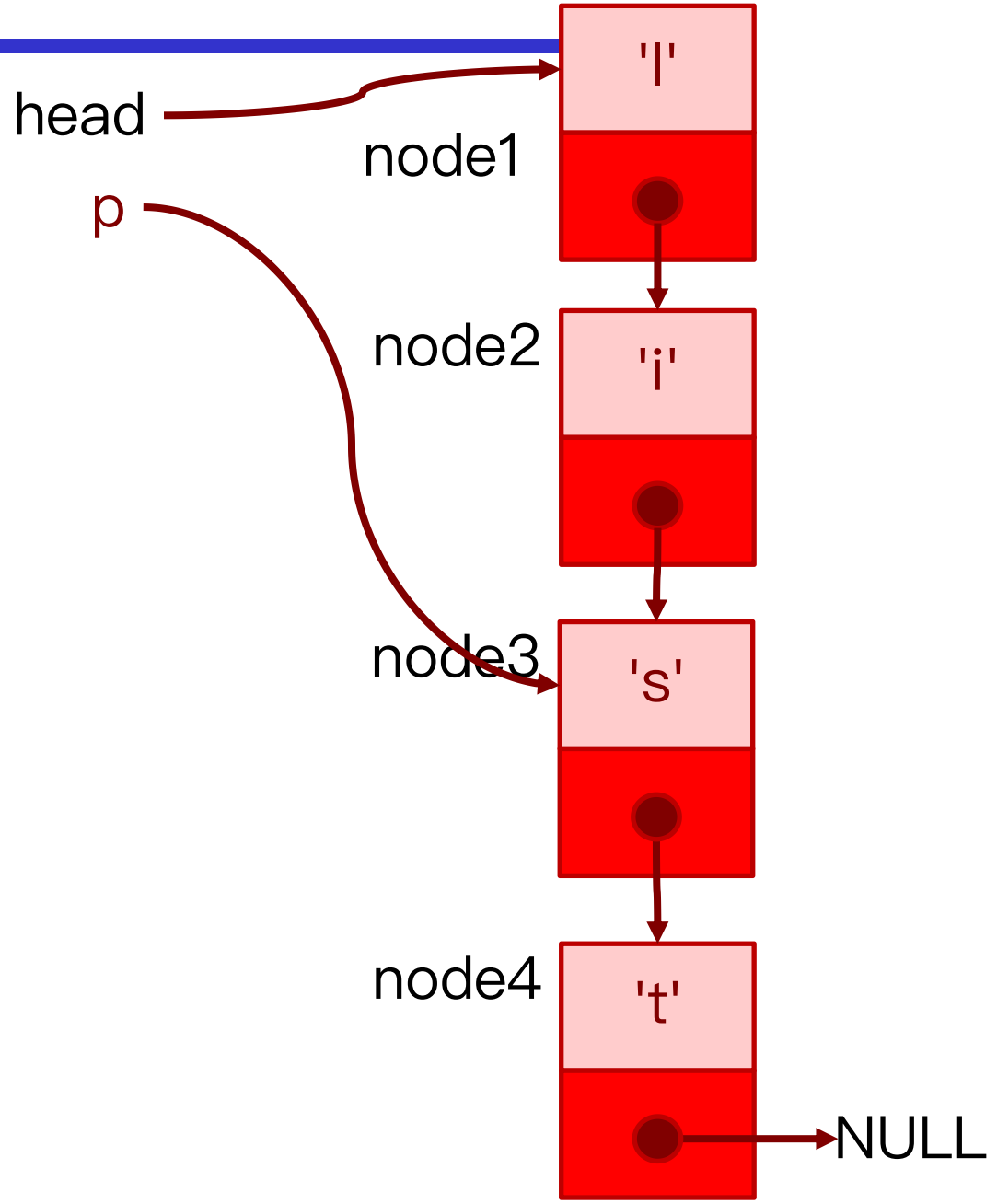
- Will always begin from head
- Visit every node until last node

- Example:

```
Node *p = head;
for( ; p != NULL; p = p->next)
  printf("%c", p->data);
```

- Output:

```
li
```



Singly-Linked List Traversal (9)

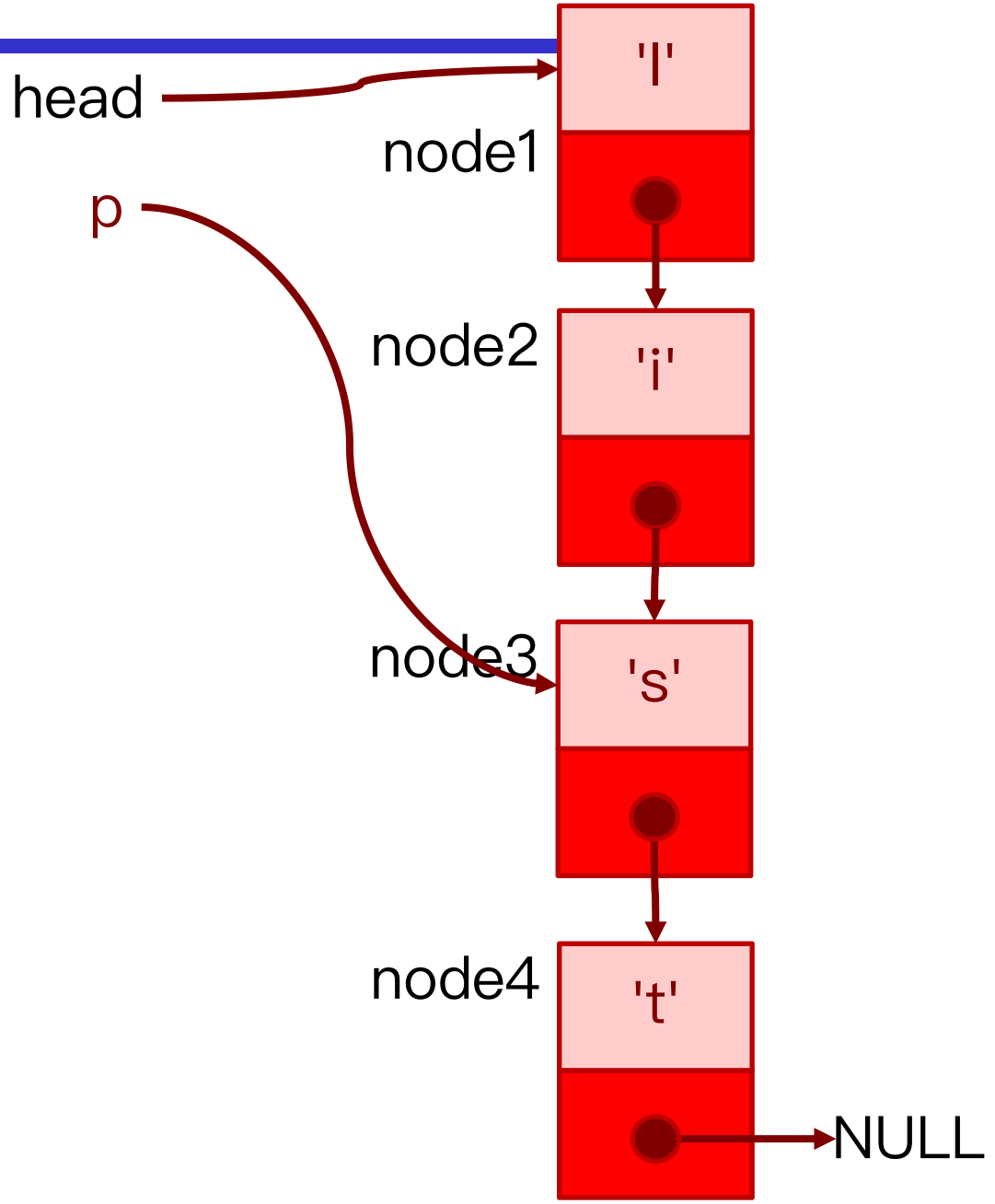
- Will always begin from head
- Visit every node until last node

- Example:

```
Node *p = head;
for( ; p != NULL; p = p->next)
  printf("%c", p->data);
```

- Output:

```
li
```



Singly-Linked List Traversal (10)

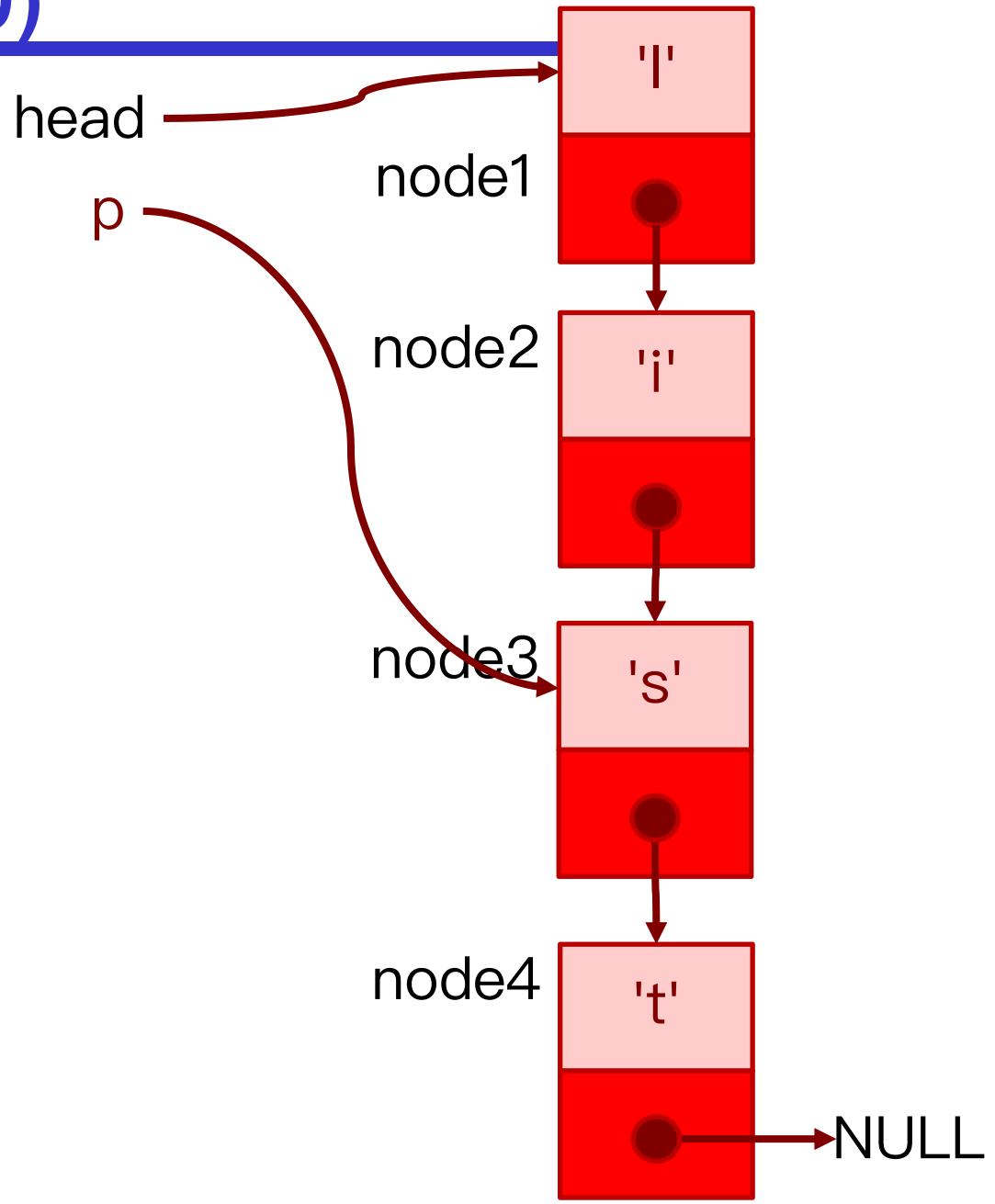
- Will always begin from head
- Visit every node until last node

- Example:

```
Node *p = head;
for( ; p != NULL; p = p->next)
  printf("%c", p->data);
```

- Output:

```
lis
```



Singly-Linked List Traversal (11)

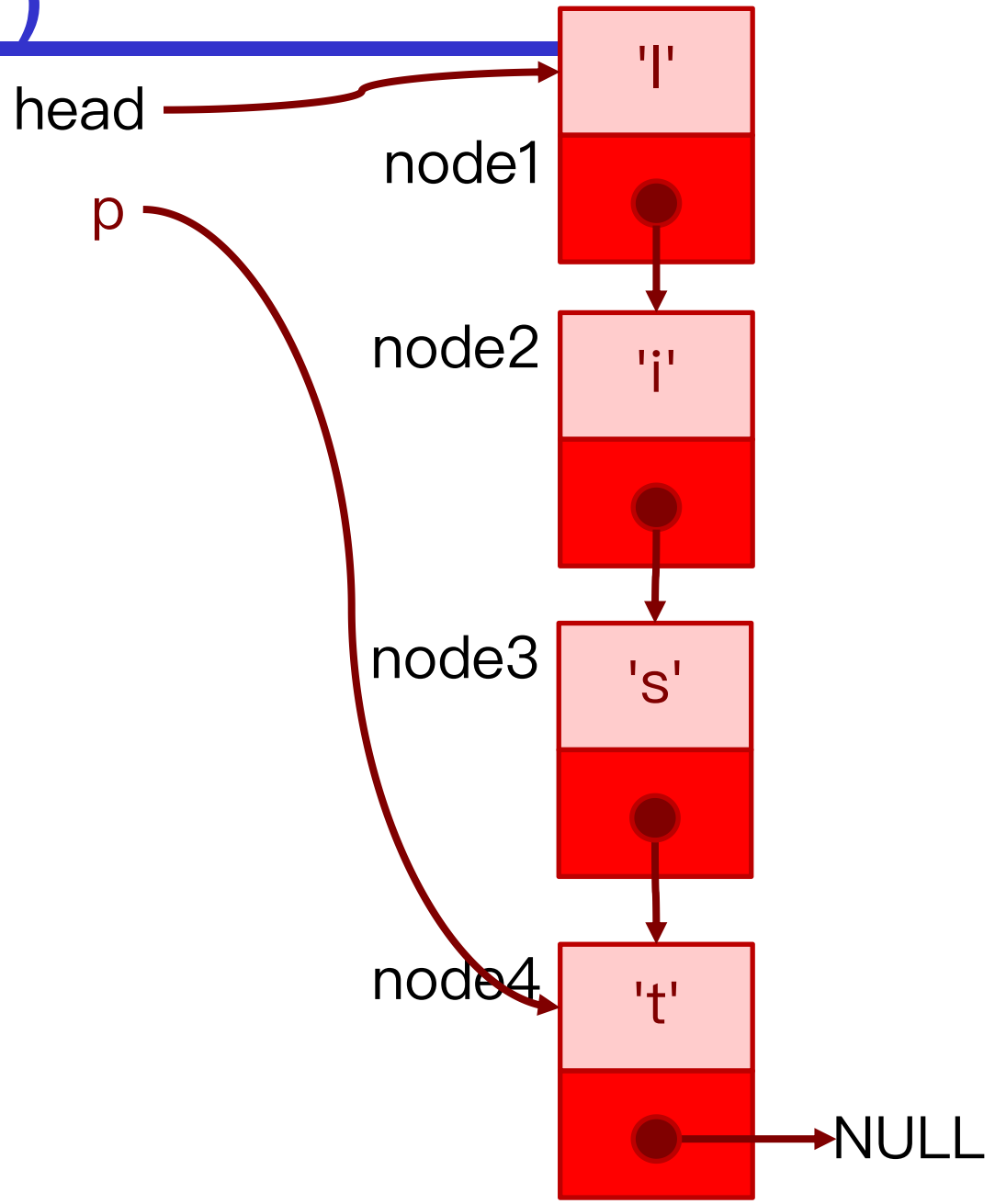
- Will always begin from head
- Visit every node until last node

- Example:

```
Node *p = head;
for( ; p != NULL; p = p->next)
  printf("%c", p->data);
```

- Output:

```
lis
```



Singly-Linked List Traversal (12)

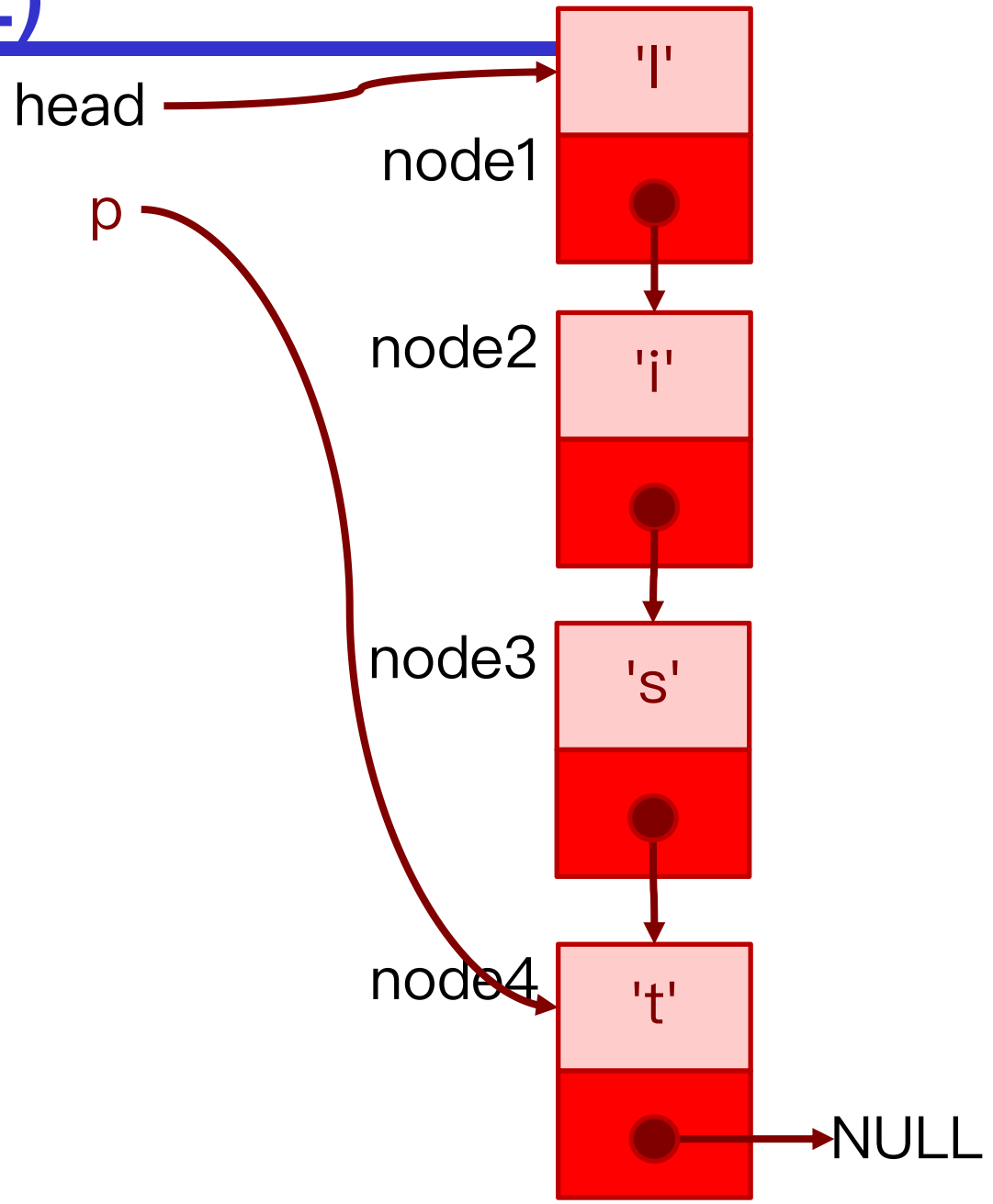
- Will always begin from head
- Visit every node until last node

- Example:

```
Node *p = head;
for( ; p != NULL; p = p->next)
  printf("%c", p->data);
```

- Output:

```
lis
```



Singly-Linked List Traversal (13)

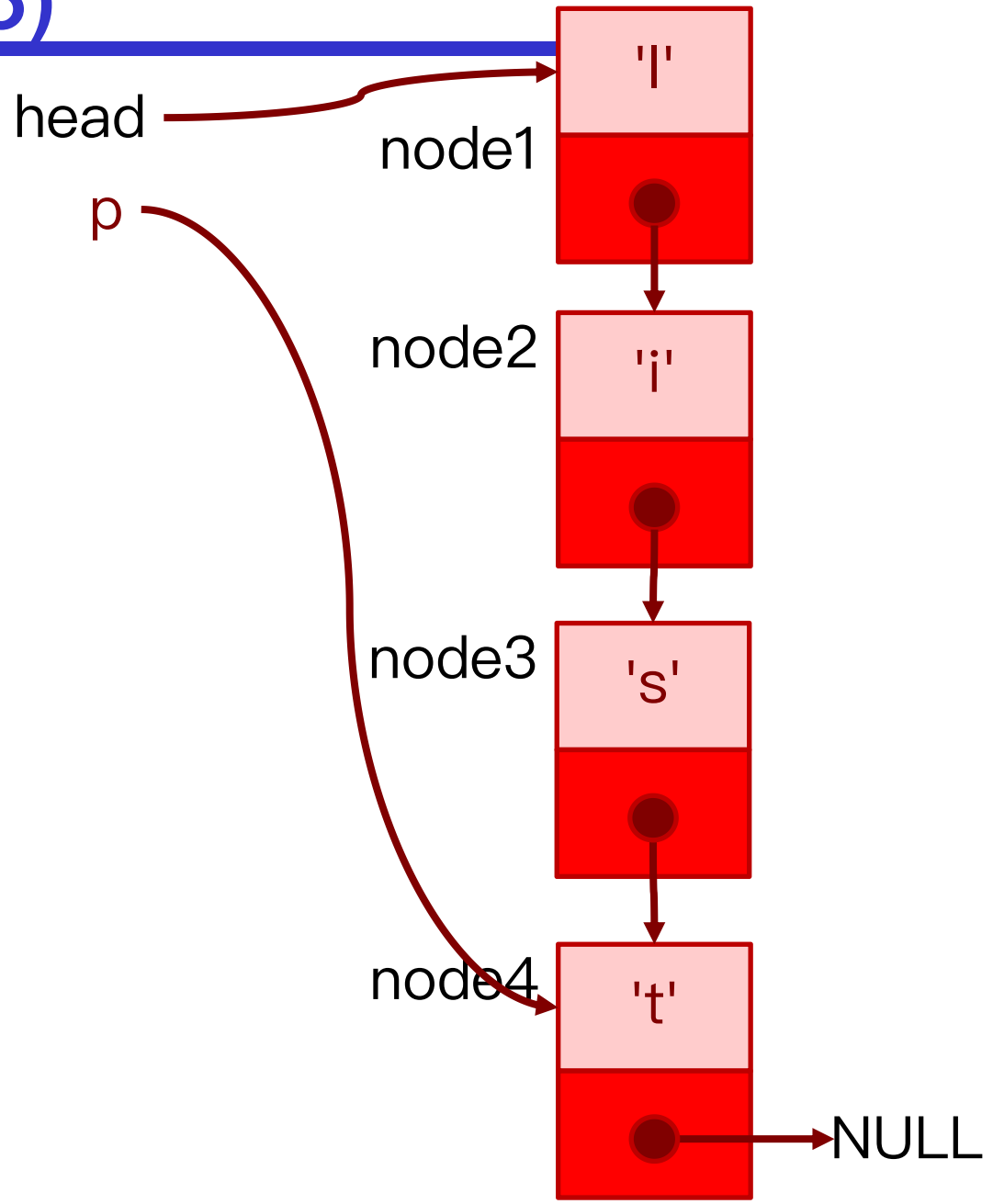
- Will always begin from head
- Visit every node until last node

- Example:

```
Node *p = head;
for( ; p != NULL; p = p->next)
  printf("%c", p->data);
```

- Output:

```
list
```



Singly-Linked List Traversal (14)

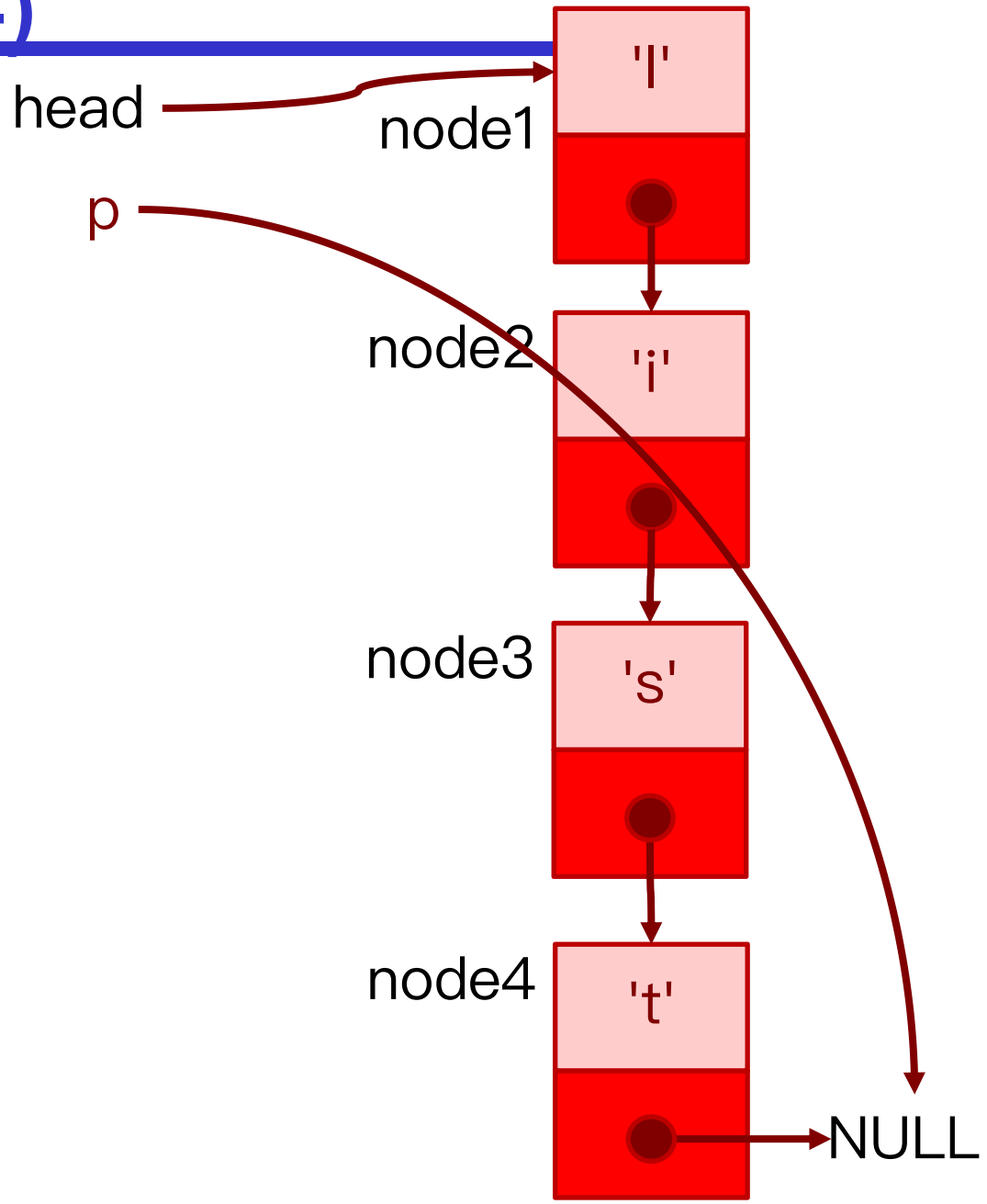
- Will always begin from head
- Visit every node until last node

- Example:

```
Node *p = head;
for( ; p != NULL; p = p->next)
  printf("%c", p->data);
```

- Output:

```
list
```



Singly-Linked List Traversal (15)

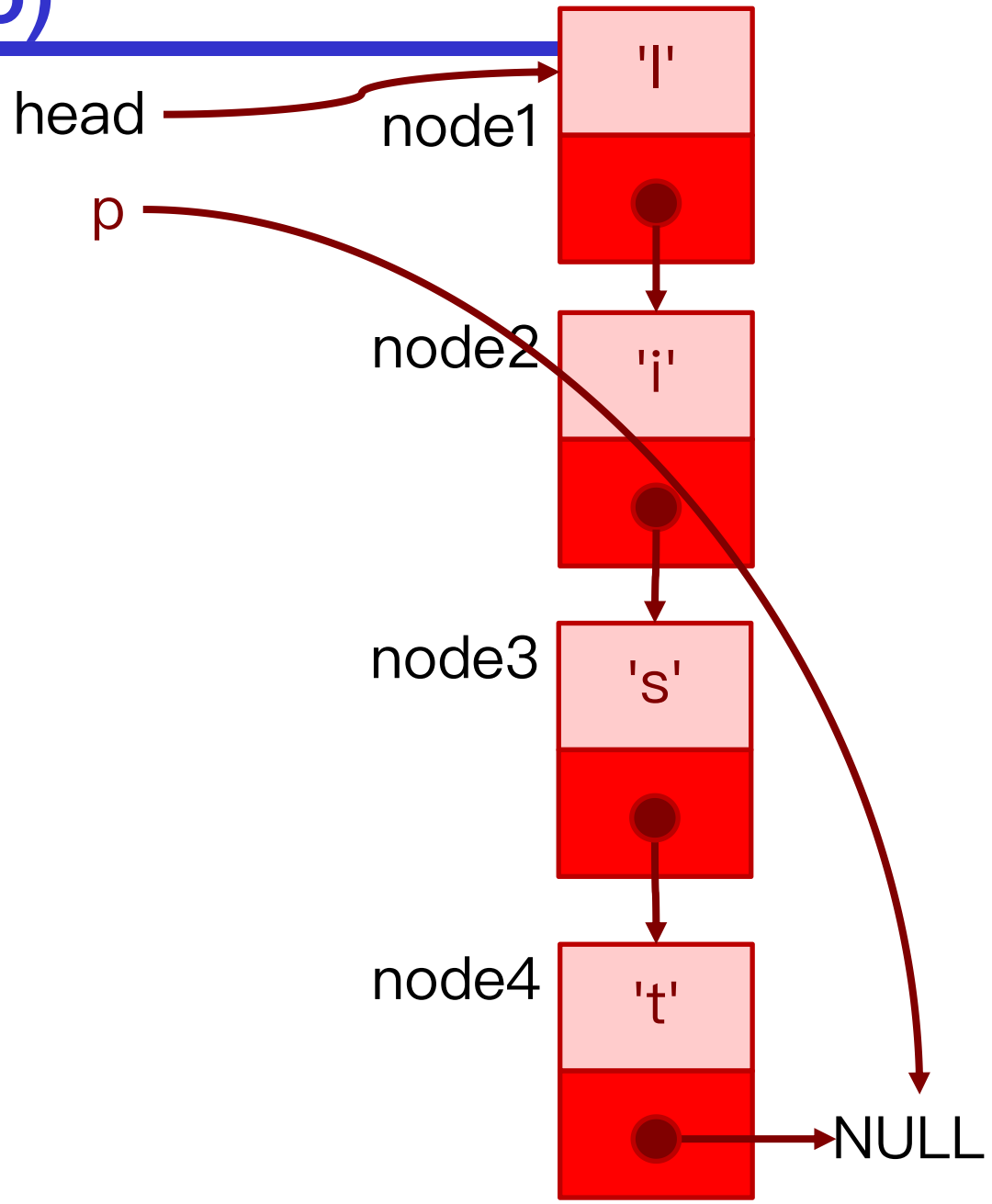
- Will always begin from head
- Visit every node until last node

- Example:

```
Node *p = head;
for( ; p != NULL; p = p->next)
  printf("%c", p->data);
```

- Output:

```
list
```



Problems with Previous Example

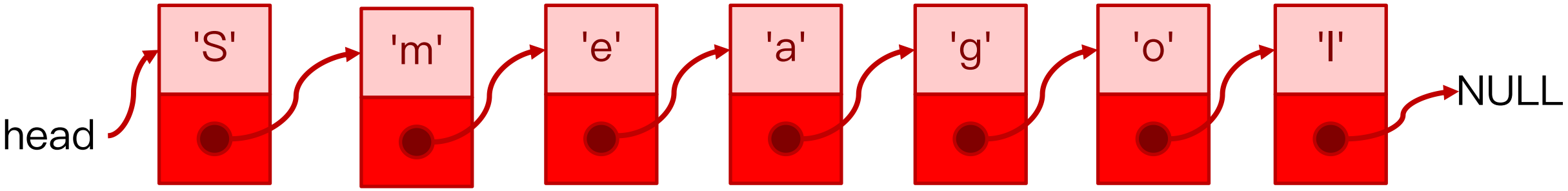
- Need to know list elements during coding
- What if the list elements are not known prior to program execution?
- The right way: **use dynamic memory allocation**

Motivating Example

- Ask user to input arbitrary string
- Convert string to a singly-linked list, with each node containing a character
- User Input:

- Linked List:

```
Smeagol
```



Preliminaries

- Node type definition

```
typedef struct node
{ char data;
  struct node *next;
} Node;
```

- Node variables declaration and initialization

```
Node *head = NULL;
```

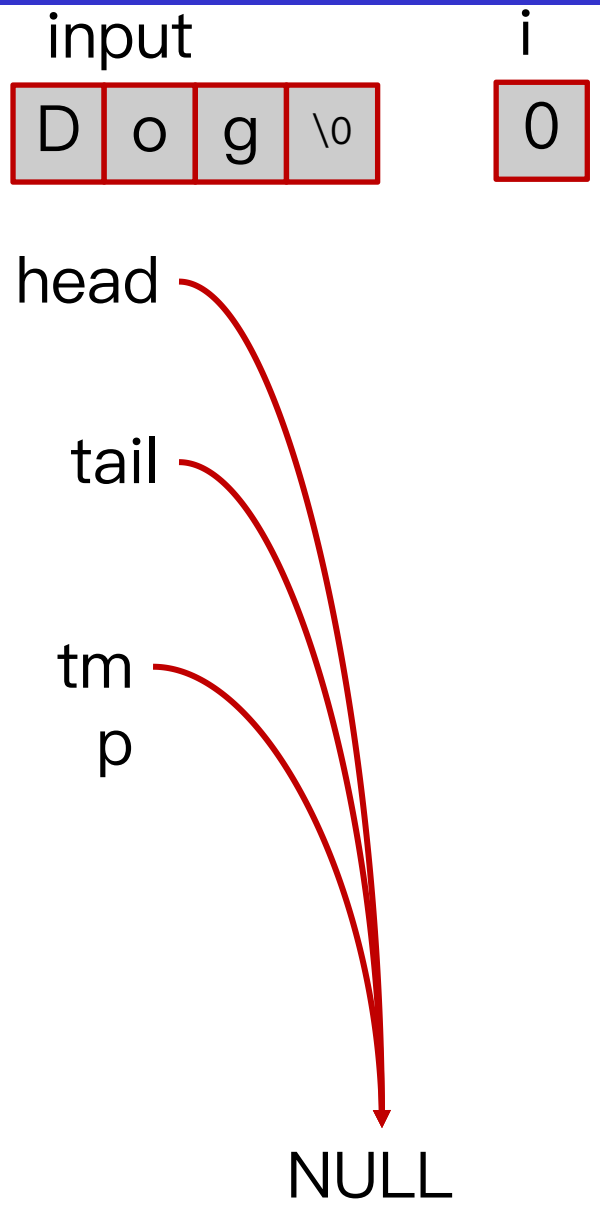
The Rest of The Code

```
char input[100];
int i = 0;
Node *tail = NULL, *tmp;

scanf("%s", input);

while(input[i] != '\0') {
    tmp = (Node *)malloc(sizeof(Node));
    tmp->data = input[i];
    tmp->next = NULL;
    if(head == NULL) { head = tmp; tail = head; }
    else { tail->next = tmp; tail = tmp; }
    i++;
}
```

Walkthrough

- Suppose user input is Dog

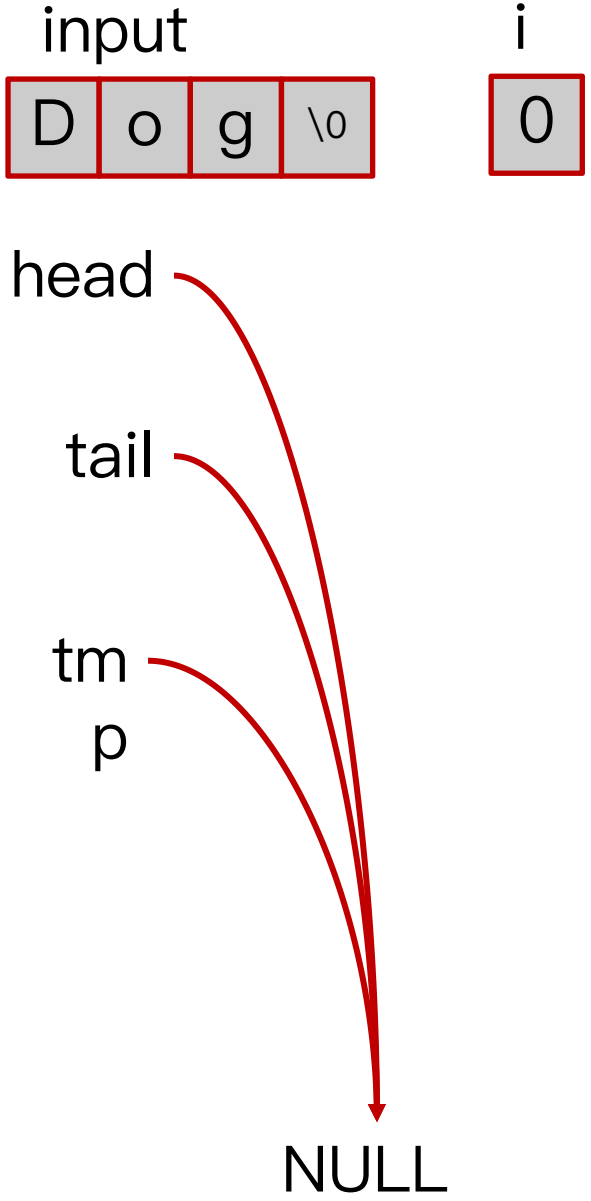
```
Node *head = NULL;

char input[100];
int i = 0;

Node *tmp = NULL;
Node *tail = NULL;

scanf("%s", input);

...
```



Walkthrough

- Suppose user input is Dog

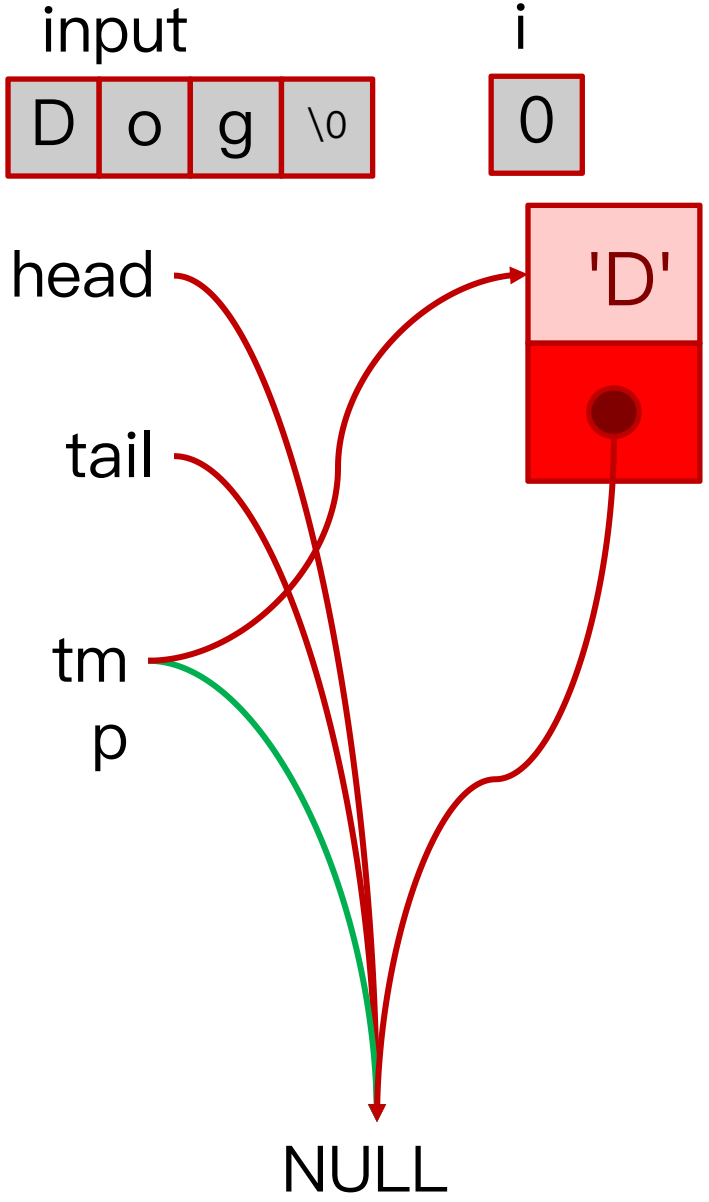
```
while(input[i] != '\0') {
    tmp = (Node *)malloc(sizeof(Node));
    tmp->data = input[i];
    tmp->next = NULL;
    if(head == NULL) {
        head = tmp;
        tail = head;
    } else {
        tail->next = tmp;
        tail = tmp;
    }
    i++;
}
```



Walkthrough

- Suppose user input is Dog

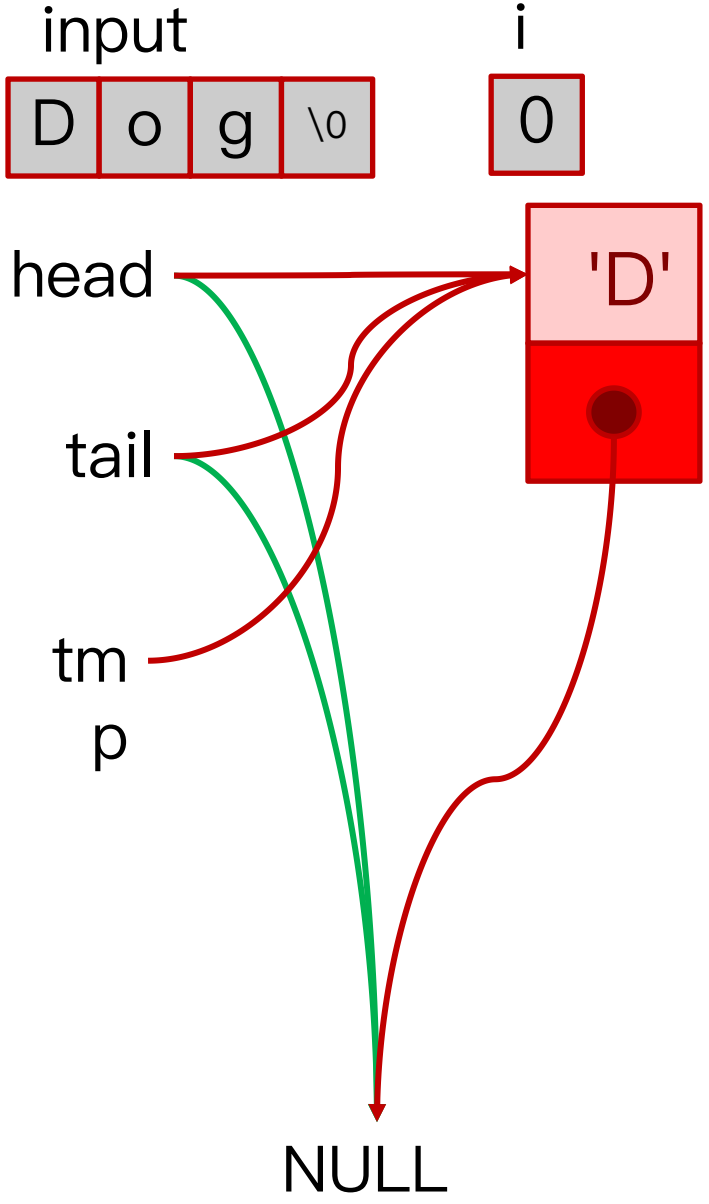
```
while(input[i] != '\0') {
    tmp = (Node *)malloc(sizeof(Node));
    tmp->data = input[i];
    tmp->next = NULL;
    if(head == NULL) {
        head = tmp;
        tail = head;
    } else {
        tail->next = tmp;
        tail = tmp;
    }
    i++;
}
```



Walkthrough

- Suppose user input is Dog

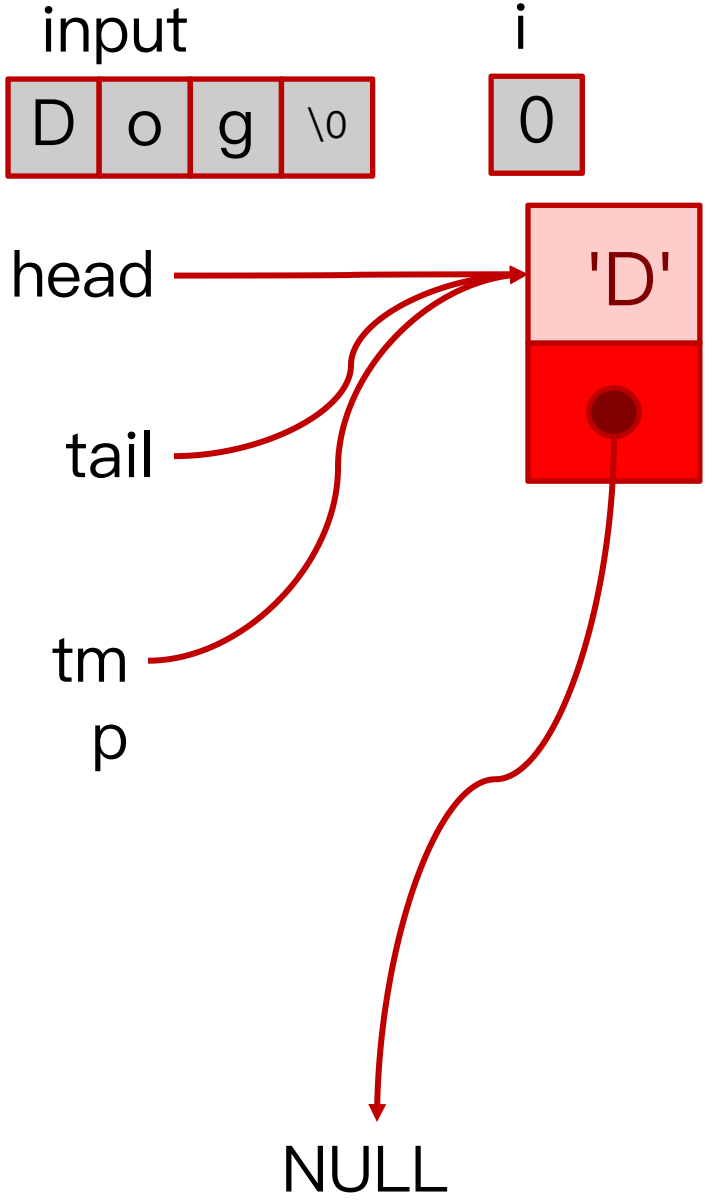
```
while(input[i] != '\0') {
    tmp = (Node *)malloc(sizeof(Node));
    tmp->data = input[i];
    tmp->next = NULL;
    if(head == NULL) {
        head = tmp;
        tail = head;
    } else {
        tail->next = tmp;
        tail = tmp;
    }
    i++;
}
```



Walkthrough

- Suppose user input is Dog

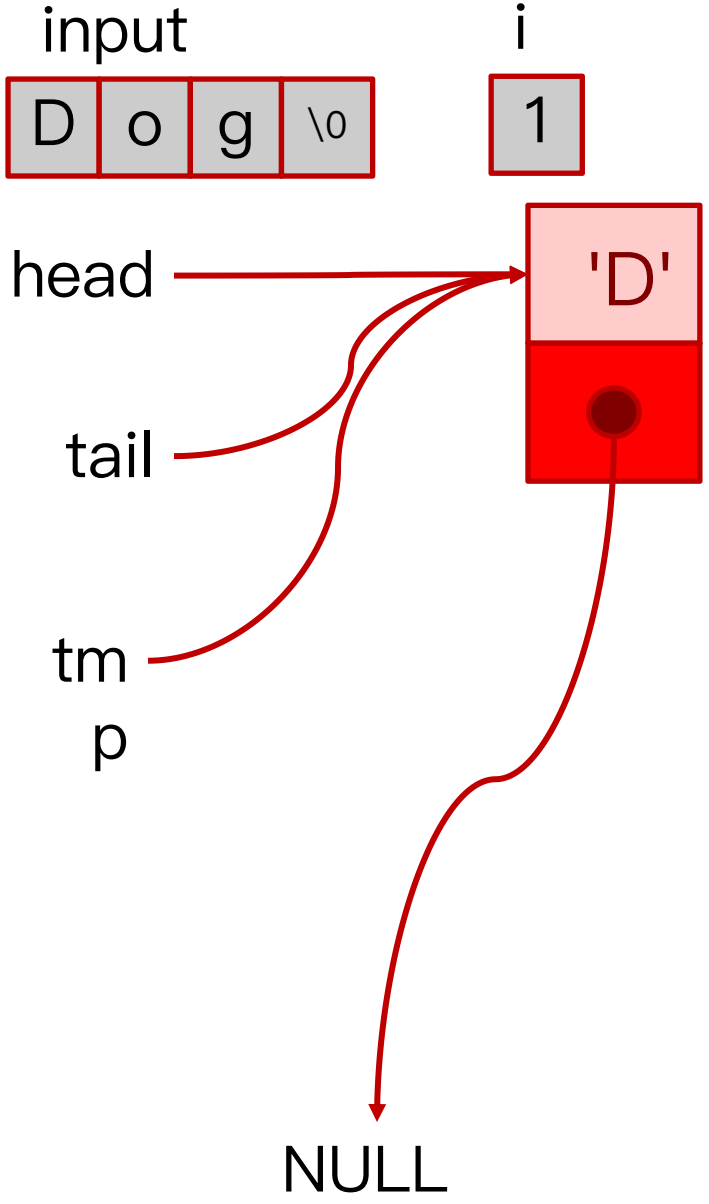
```
while(input[i] != '\0') {
    tmp = (Node *)malloc(sizeof(Node));
    tmp->data = input[i];
    tmp->next = NULL;
    if(head == NULL) {
        head = tmp;
        tail = head;
    } else {
        tail->next = tmp;
        tail = tmp;
    }
    i++;
}
```



Walkthrough

- Suppose user input is Dog

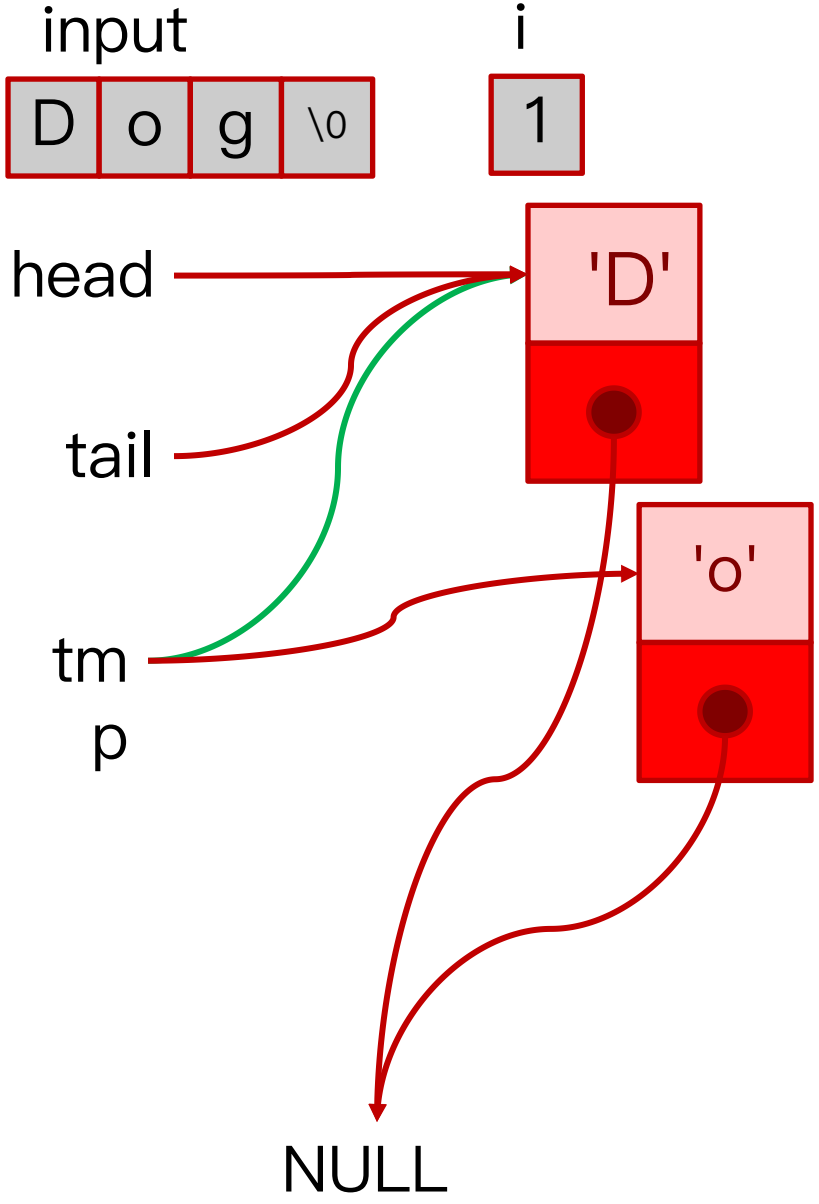
```
while(input[i] != '\0') {
    tmp = (Node *)malloc(sizeof(Node));
    tmp->data = input[i];
    tmp->next = NULL;
    if(head == NULL) {
        head = tmp;
        tail = head;
    } else {
        tail->next = tmp;
        tail = tmp;
    }
    i++;
}
```



Walkthrough

- Suppose user input is Dog

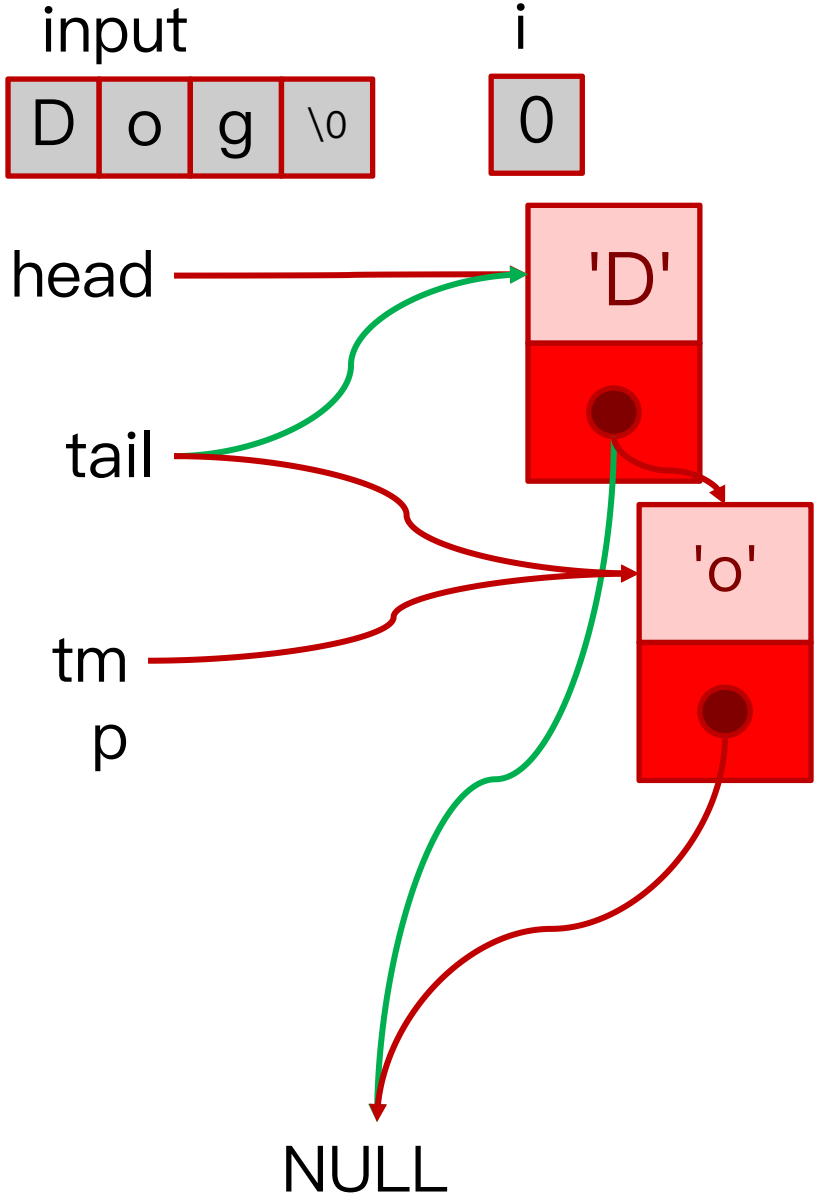
```
while(input[i] != '\0') {
    tmp = (Node *)malloc(sizeof(Node));
    tmp->data = input[i];
    tmp->next = NULL;
    if(head == NULL) {
        head = tmp;
        tail = head;
    } else {
        tail->next = tmp;
        tail = tmp;
    }
    i++;
}
```



Walkthrough

- Suppose user input is Dog

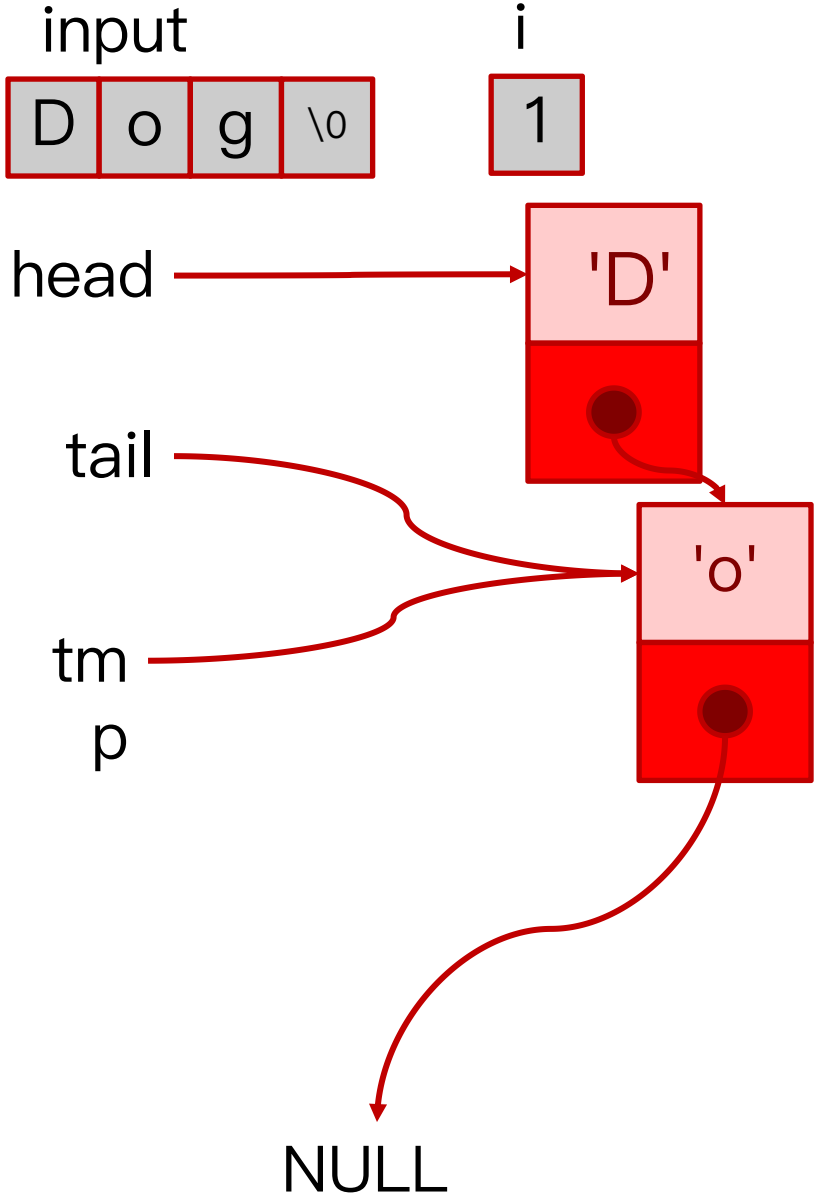
```
while(input[i] != '\0') {
    tmp = (Node *)malloc(sizeof(Node));
    tmp->data = input[i];
    tmp->next = NULL;
    if(head == NULL) {
        head = tmp;
        tail = head;
    } else {
        tail->next = tmp;
        tail = tmp;
    }
    i++;
}
```



Walkthrough

- Suppose user input is Dog

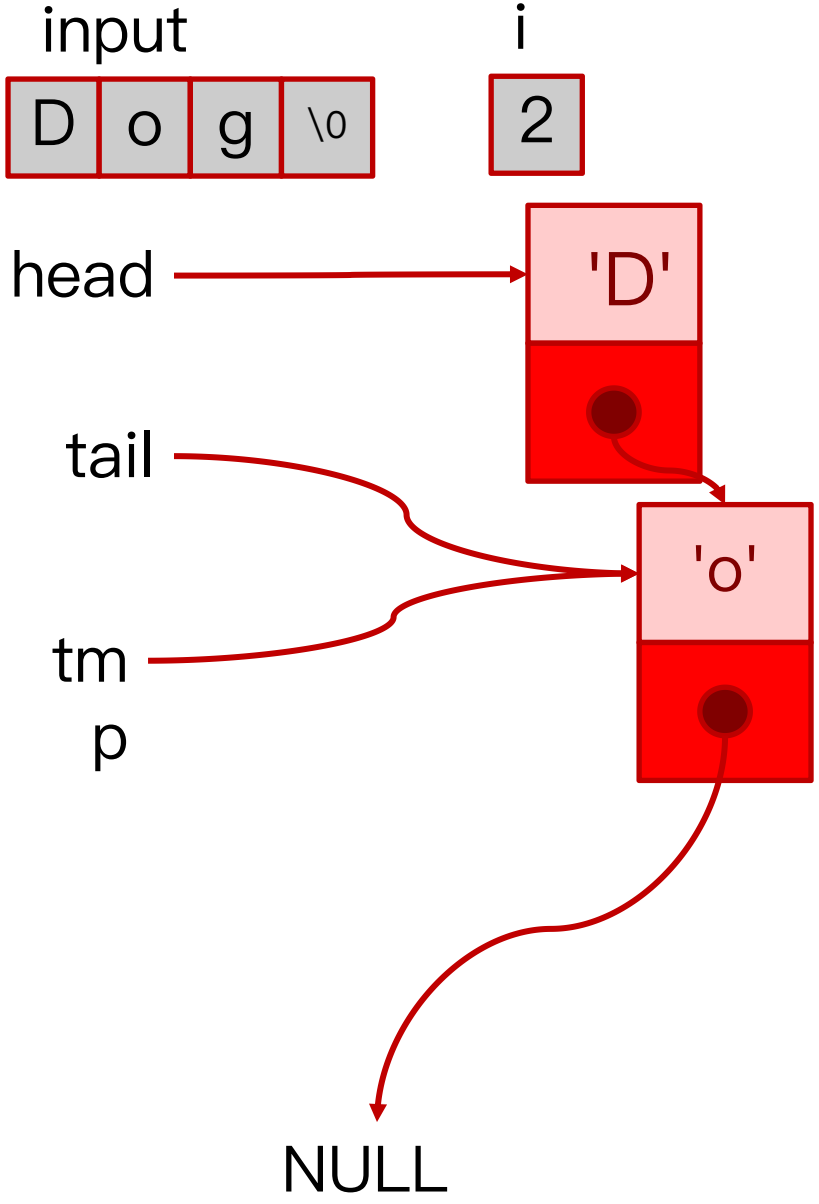
```
while(input[i] != '\0') {
    tmp = (Node *)malloc(sizeof(Node));
    tmp->data = input[i];
    tmp->next = NULL;
    if(head == NULL) {
        head = tmp;
        tail = head;
    } else {
        tail->next = tmp;
        tail = tmp;
    }
    i++;
}
```



Walkthrough

- Suppose user input is Dog

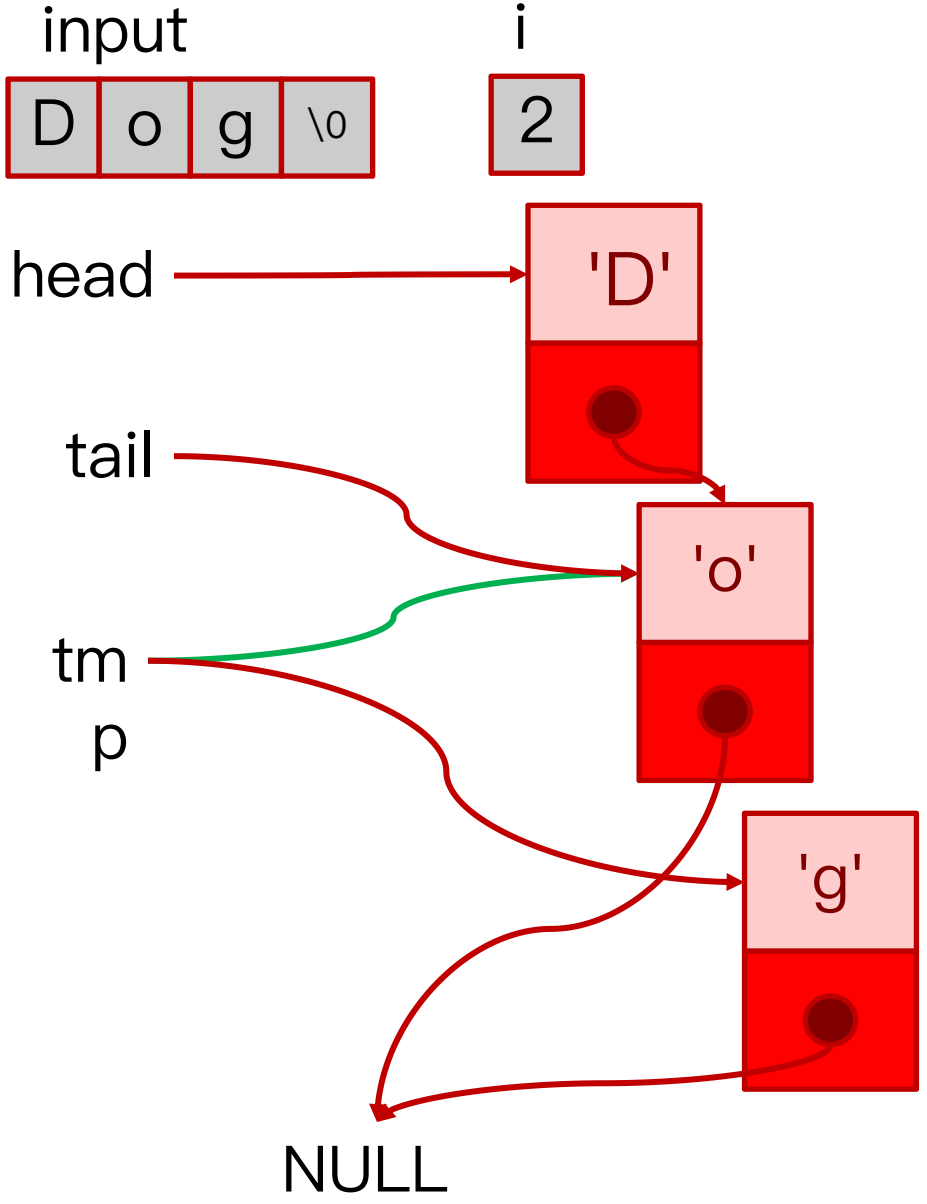
```
while(input[i] != '\0') {
    tmp = (Node *)malloc(sizeof(Node));
    tmp->data = input[i];
    tmp->next = NULL;
    if(head == NULL) {
        head = tmp;
        tail = head;
    } else {
        tail->next = tmp;
        tail = tmp;
    }
    i++;
}
```



Walkthrough

- Suppose user input is Dog

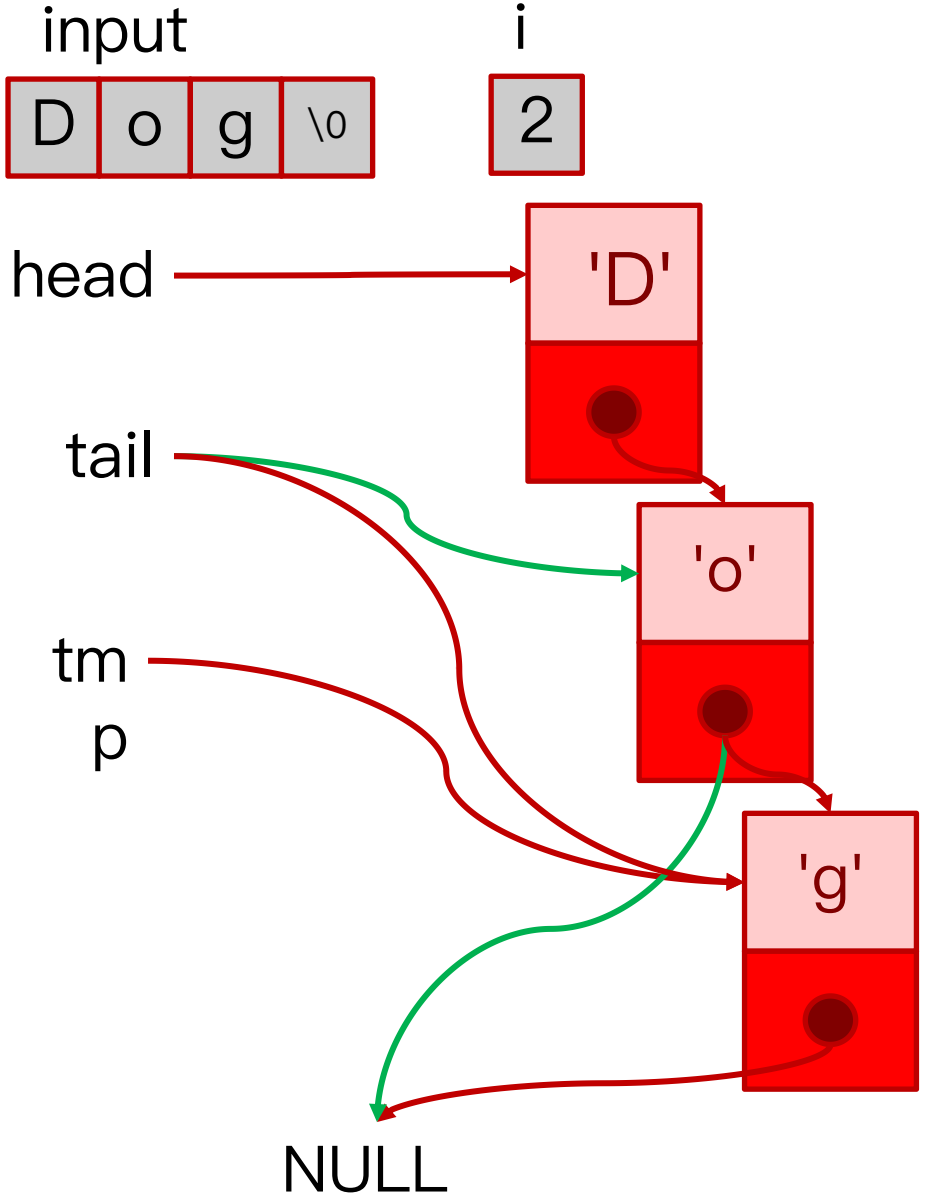
```
while(input[i] != '\0') {
    tmp = (Node *)malloc(sizeof(Node));
    tmp->data = input[i];
    tmp->next = NULL;
    if(head == NULL) {
        head = tmp;
        tail = head;
    } else {
        tail->next = tmp;
        tail = tmp;
    }
    i++;
}
```



Walkthrough

- Suppose user input is Dog

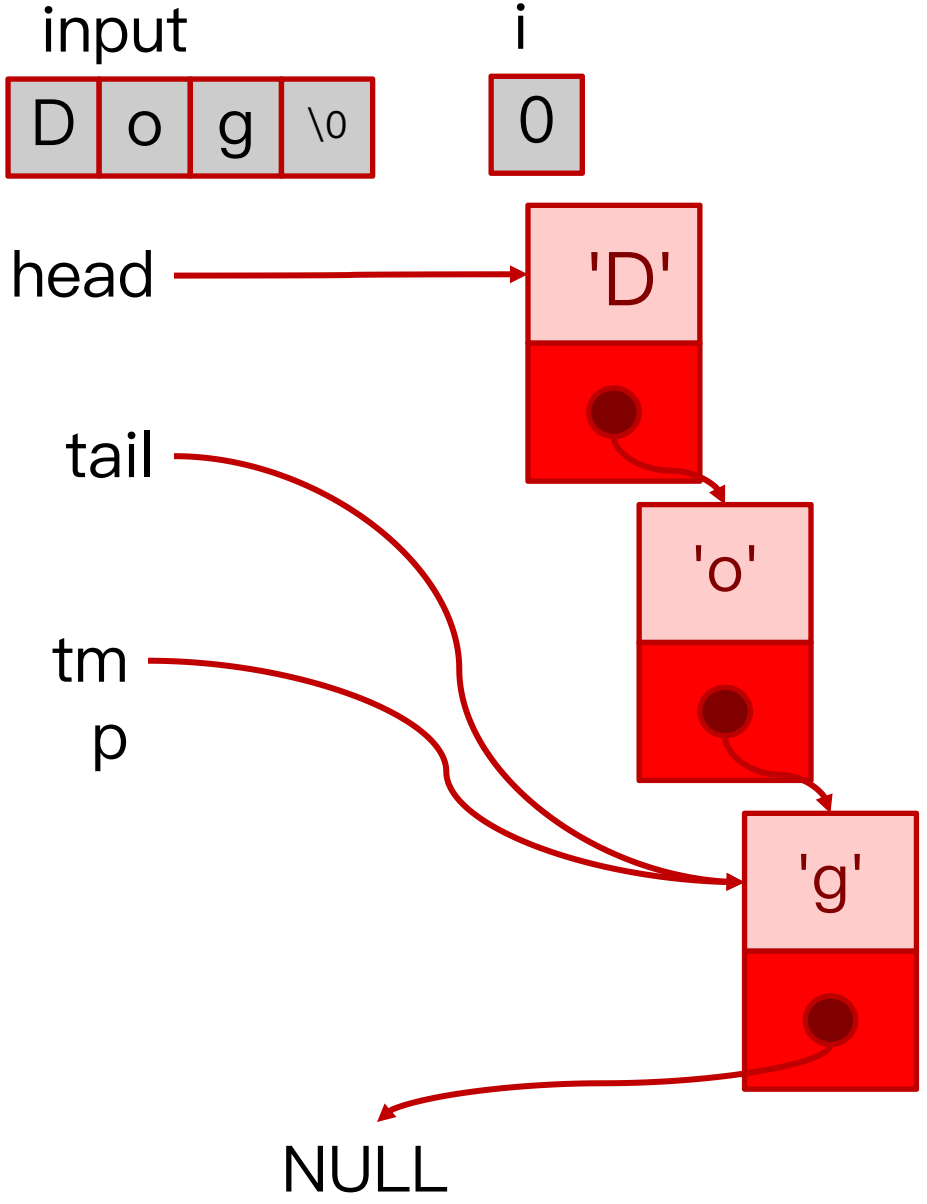
```
while(input[i] != '\0') {
    tmp = (Node *)malloc(sizeof(Node));
    tmp->data = input[i];
    tmp->next = NULL;
    if(head == NULL) {
        head = tmp;
        tail = head;
    } else {
        tail->next = tmp;
        tail = tmp;
    }
    i++;
}
```



Walkthrough

- Suppose user input is Dog

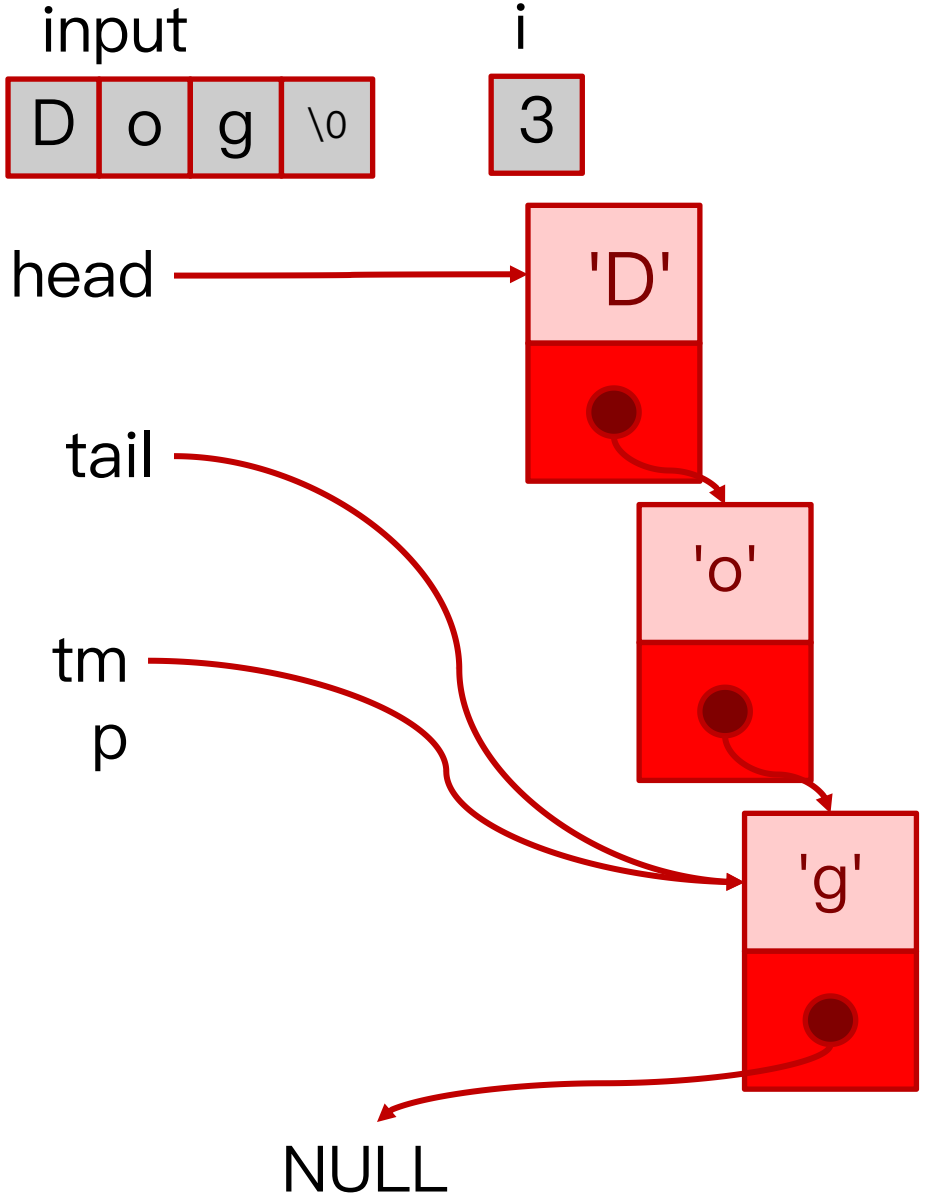
```
while(input[i] != '\0') {
    tmp = (Node *)malloc(sizeof(Node));
    tmp->data = input[i];
    tmp->next = NULL;
    if(head == NULL) {
        head = tmp;
        tail = head;
    } else {
        tail->next = tmp;
        tail = tmp;
    }
    i++;
}
```



Walkthrough

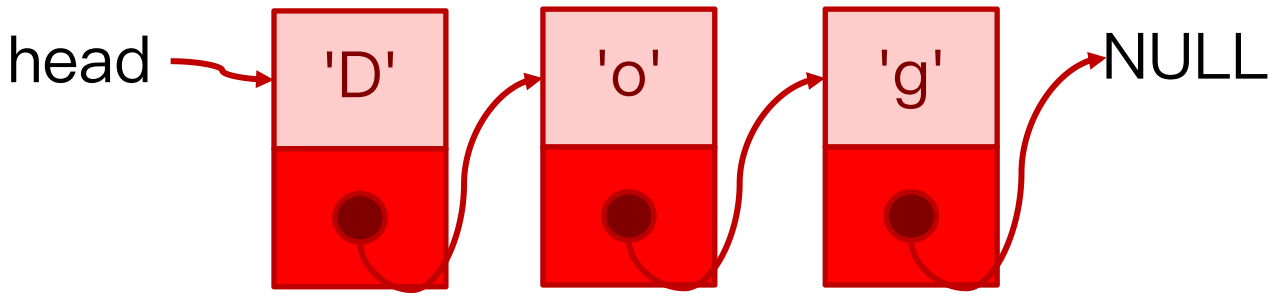
- Suppose user input is Dog

```
while(input[i] != '\0') {
    tmp = (Node *)malloc(sizeof(Node));
    tmp->data = input[i];
    tmp->next = NULL;
    if(head == NULL) {
        head = tmp;
        tail = head;
    } else {
        tail->next = tmp;
        tail = tmp;
    }
    i++;
}
```



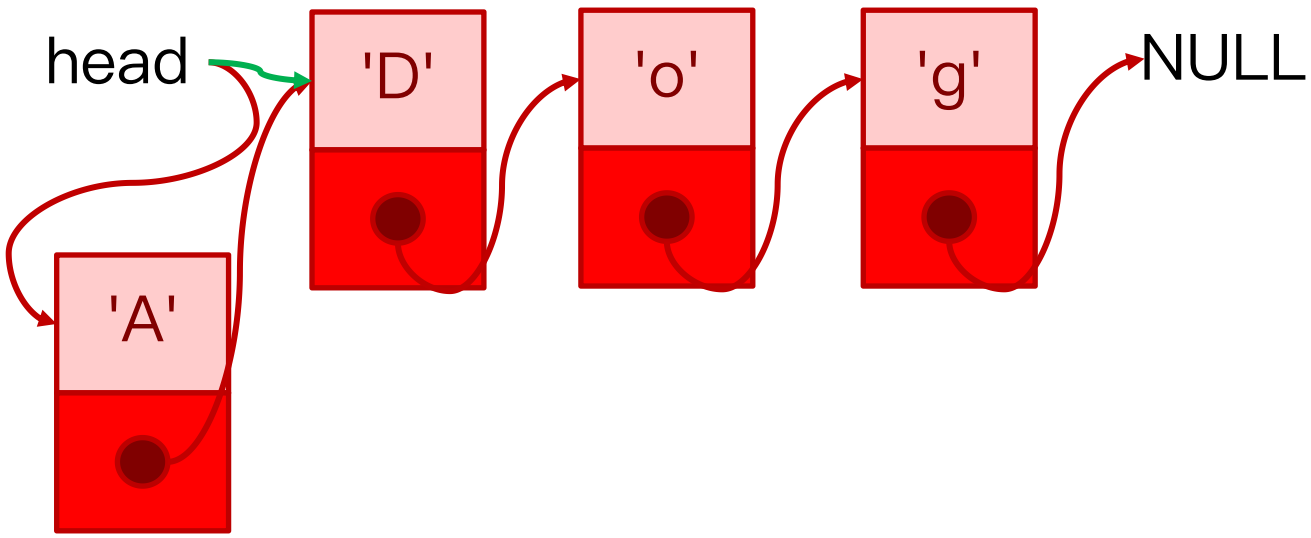
Inserting a Node (At Head)

- Easy if insertion is before first node



Inserting a Node (At Head)

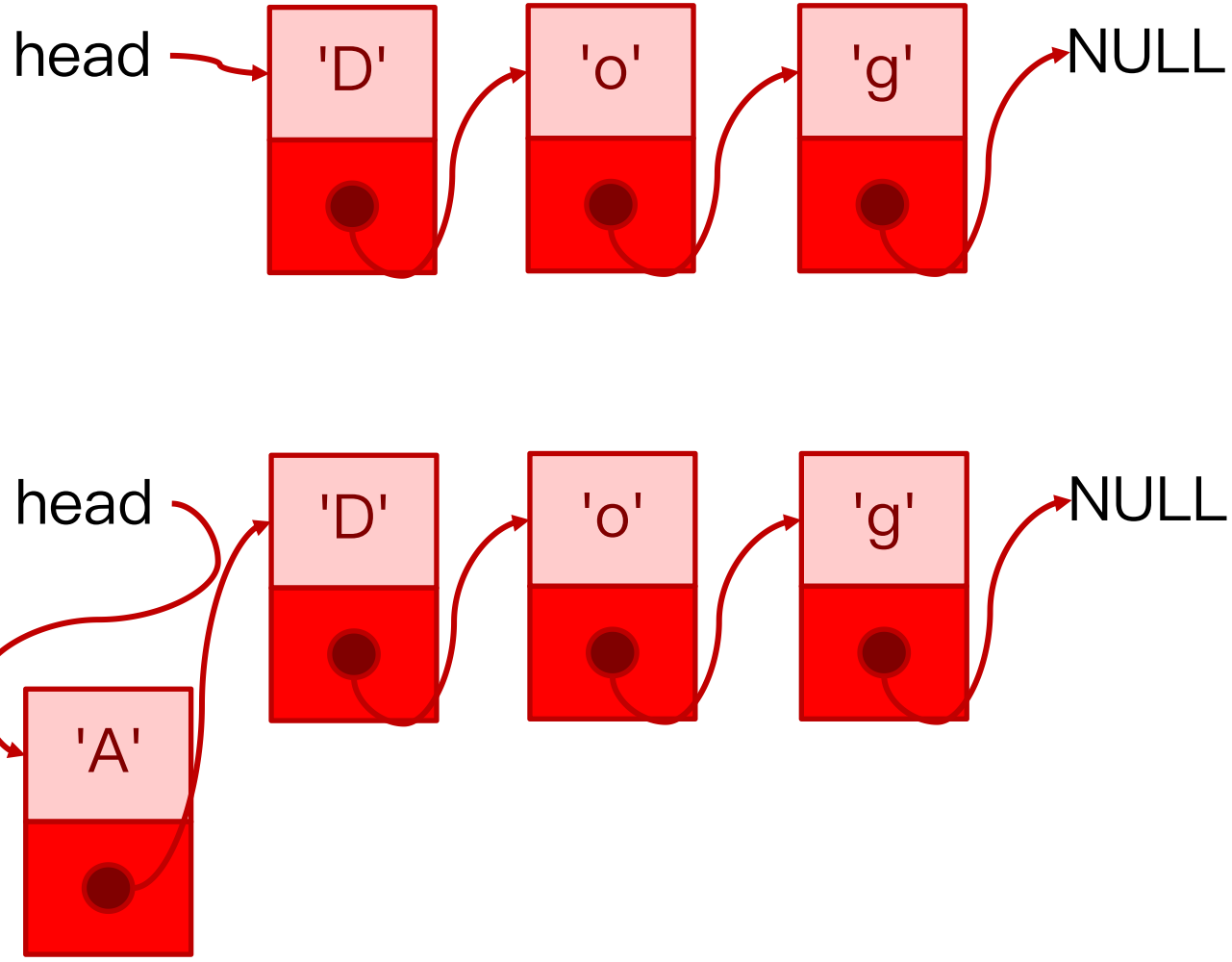
- Easy if insertion is before first node




```
Node *to_insert;

/* Allocate space for
   to_insert and
   initialize */
...

to_insert->next = head;
head = to_insert;
```

Inserting a Node (At Head)

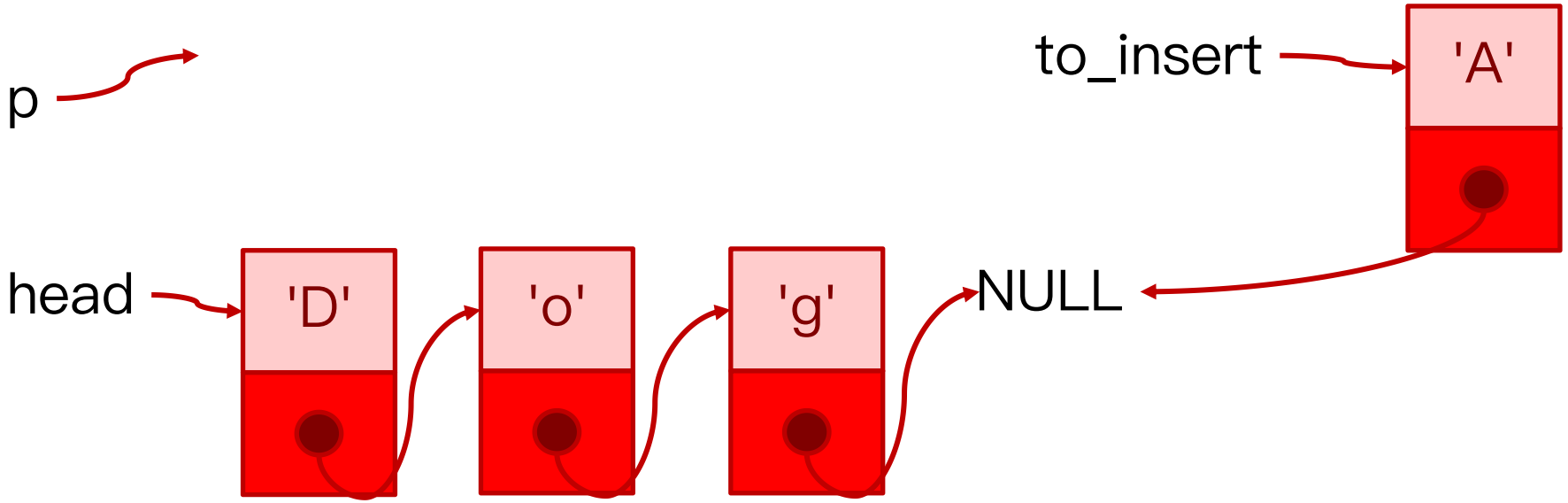
- Easy if insertion is before first node




```
Node *to_insert;

/* Allocate space for
   to_insert and
   initialize */
...

to_insert->next = head;
head = to_insert;
```

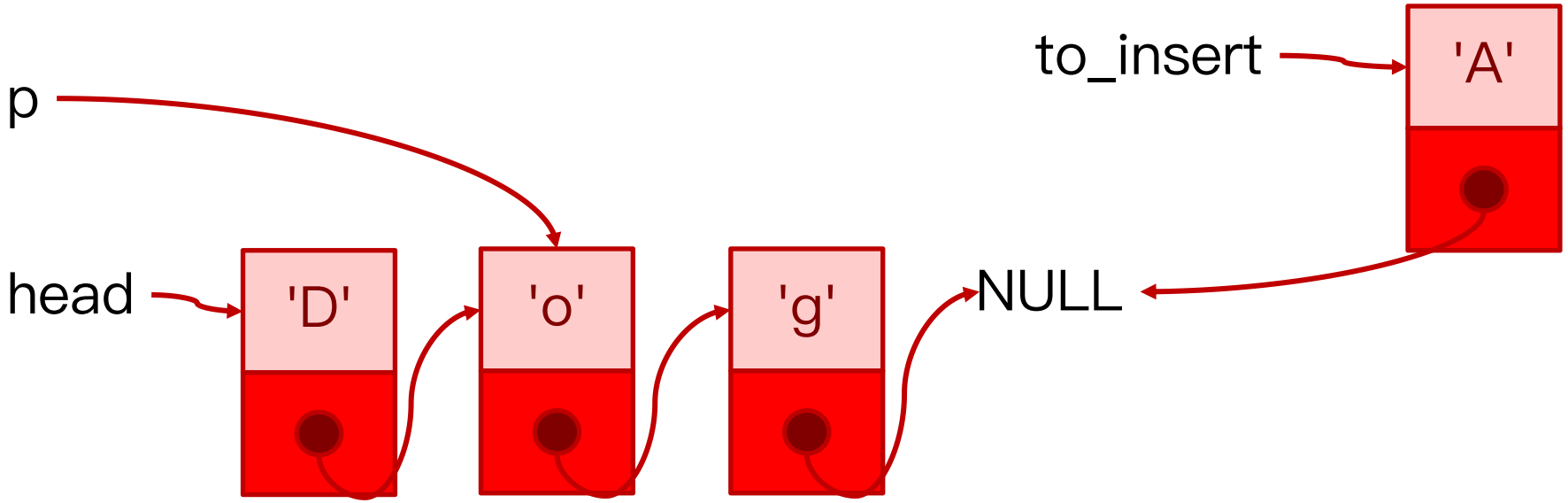
Inserting a Node (Not at Head)

- Need to traverse list until insertion point
- Illustration: Insert 'A' in between 'o' and 'g'.



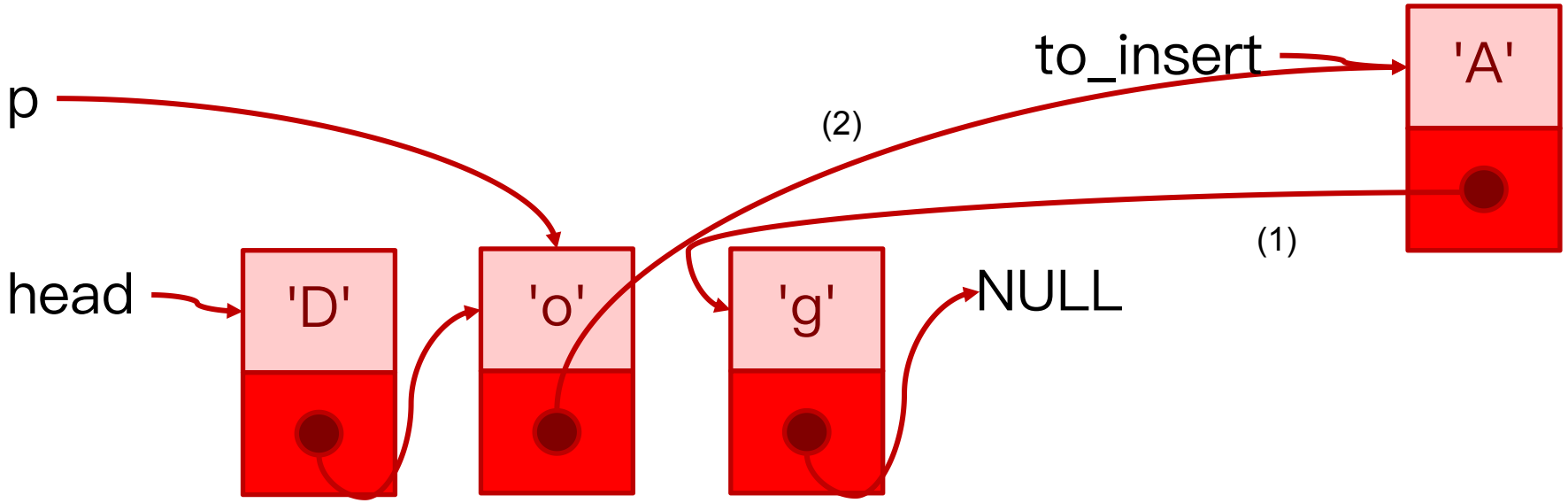
Inserting a Node (Not at Head)

- Illustration: Insert 'A' in between 'o' and 'g‘
  - Traverse p until 'o'



Inserting a Node (Not at Head)

- Illustration: Insert 'A' in between 'o' and 'g‘
  - Traverse p until 'o'
  - Insert node

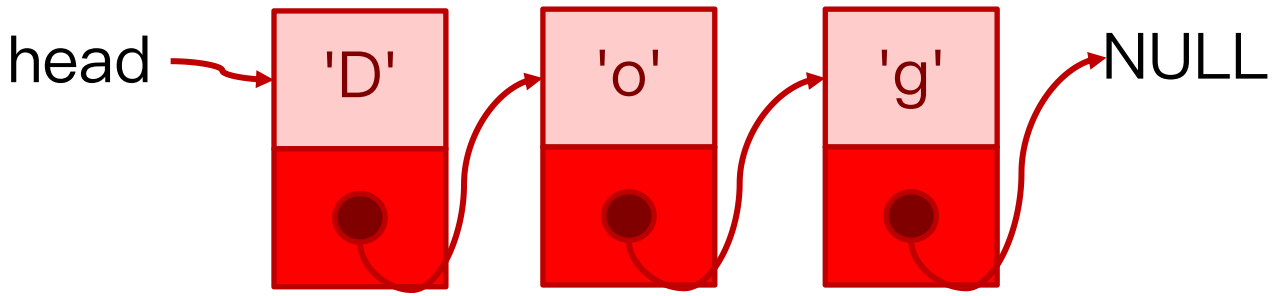


Inserting a Node (Not at Head)

```
Node *to_insert;

/* Allocate space for to_insert and initialize */
...

Node *p = head;

/* Traverse until desired node */
while(p != NULL && p->data != 'o')
        p = p->next;

/* Insert */
to_insert->next = p->next
p->next = to_insert;
```

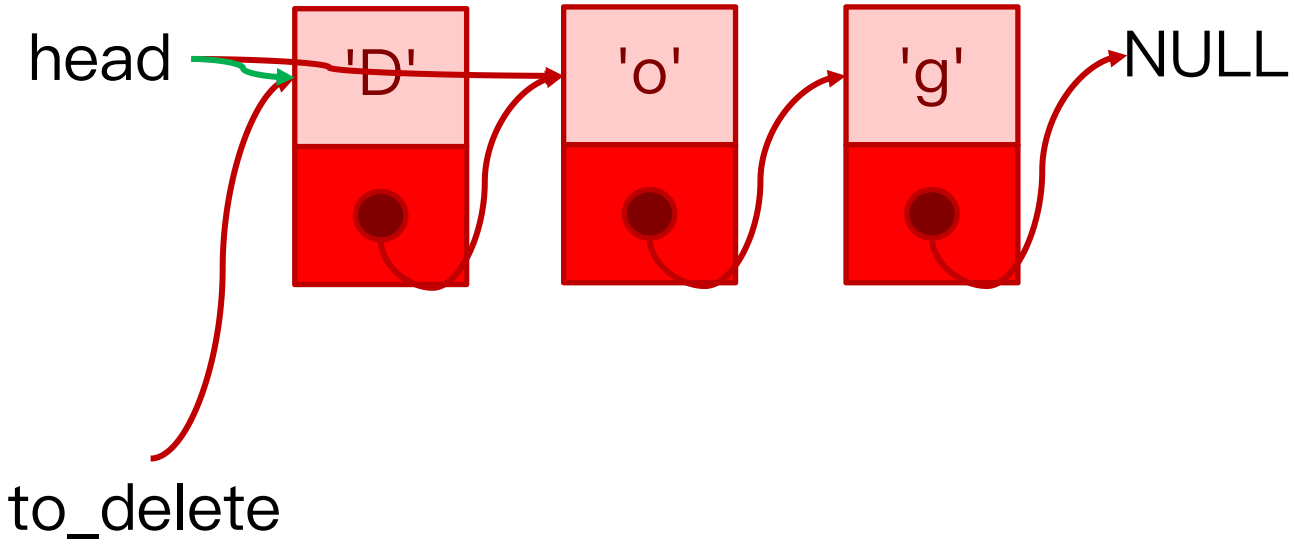
Deleting a Node (At Head)

- Easy if node to delete is first node



Deleting a Node (At Head)

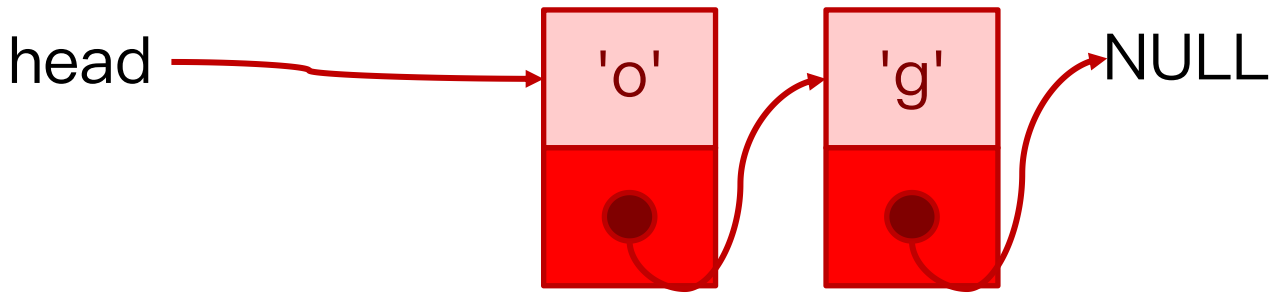
- Easy if node to delete is first node




```
Node *to_delete;

to_delete = head;
head = to_delete->next;

free(to_delete);
```

Deleting a Node (At Head)

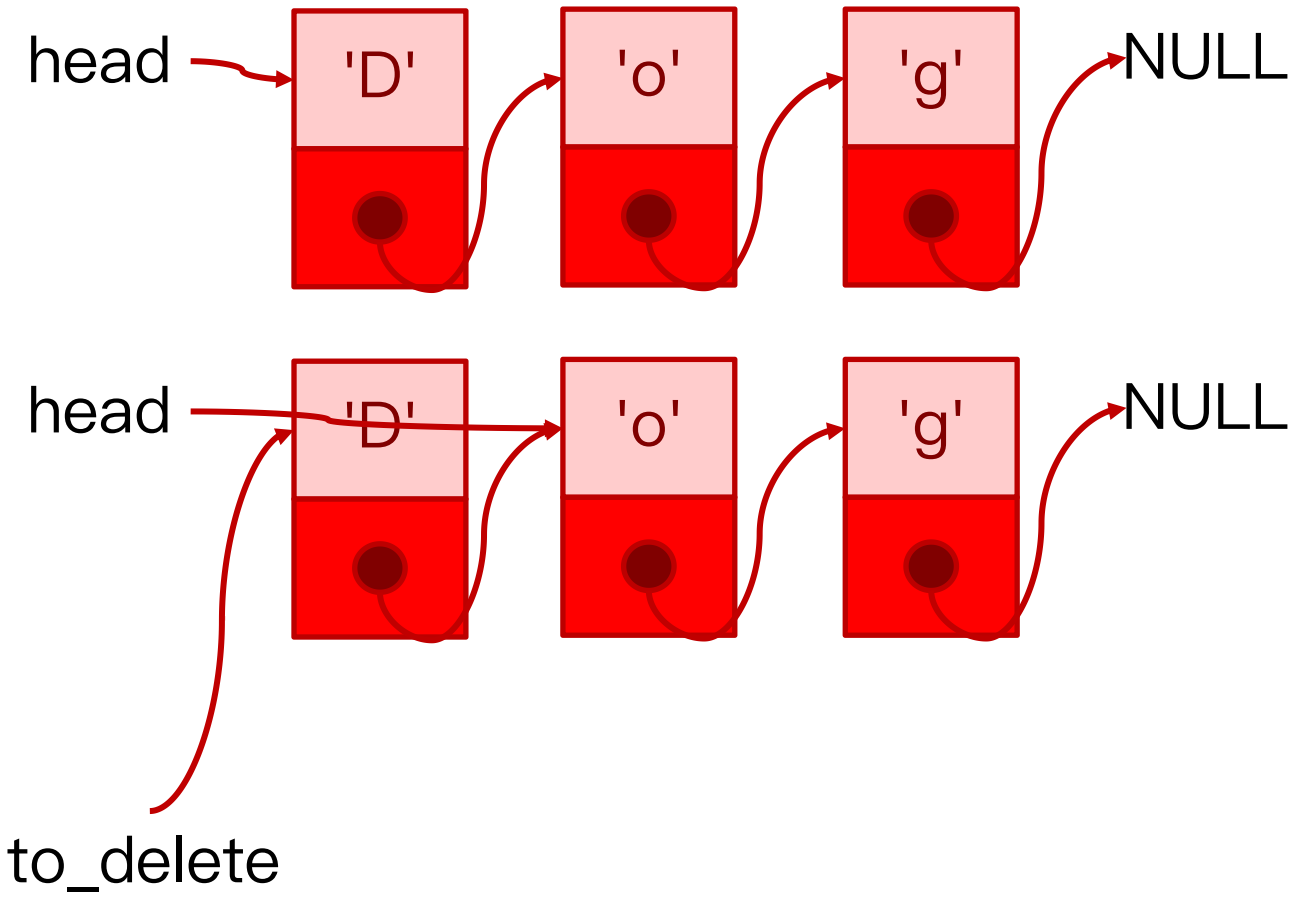
- Easy if node to delete is first node




```
Node *to_delete;

to_delete = head;
head = to_delete->next;

free(to_delete);
```

Deleting a Node (At Head)

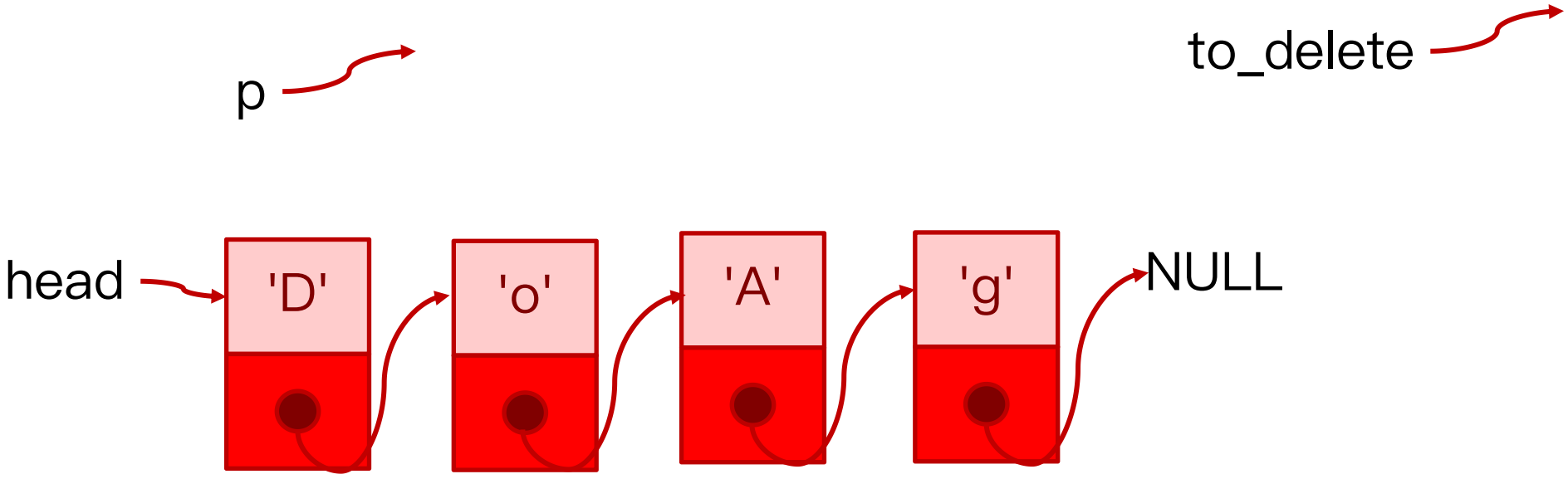
- Easy if node to delete is first node




```
Node *to_delete;

to_delete = head;
head = to_delete->next;

free(to_delete);
```

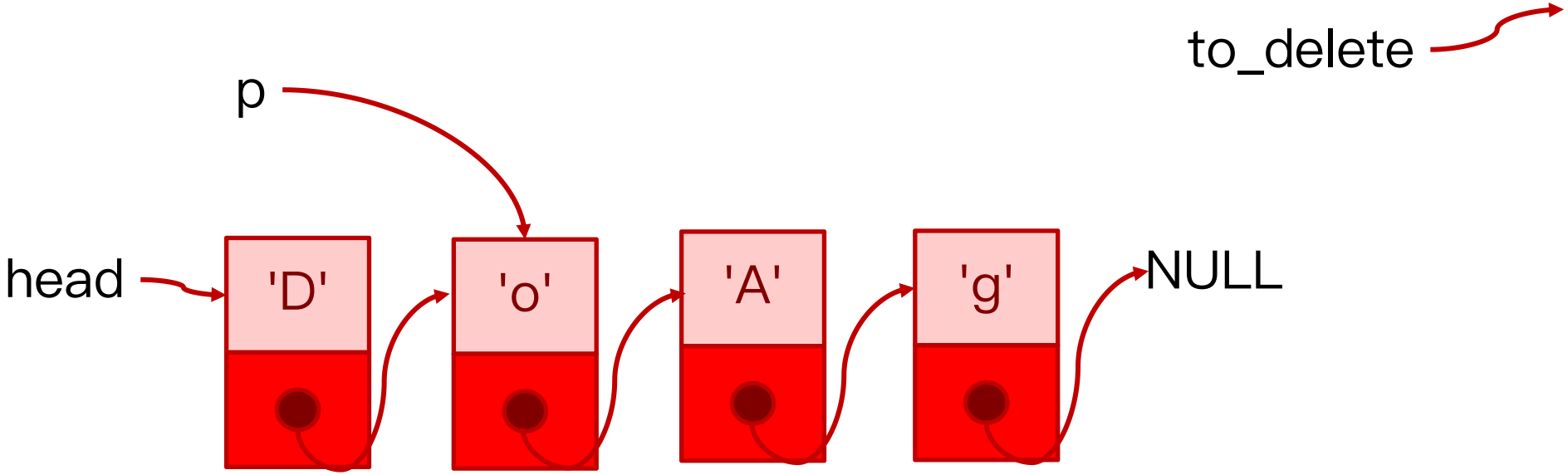
Deleting a Node (Not at Head)

- Need to traverse list until just before the node to be deleted
- Illustration: Delete 'A' in between 'o' and 'g'.



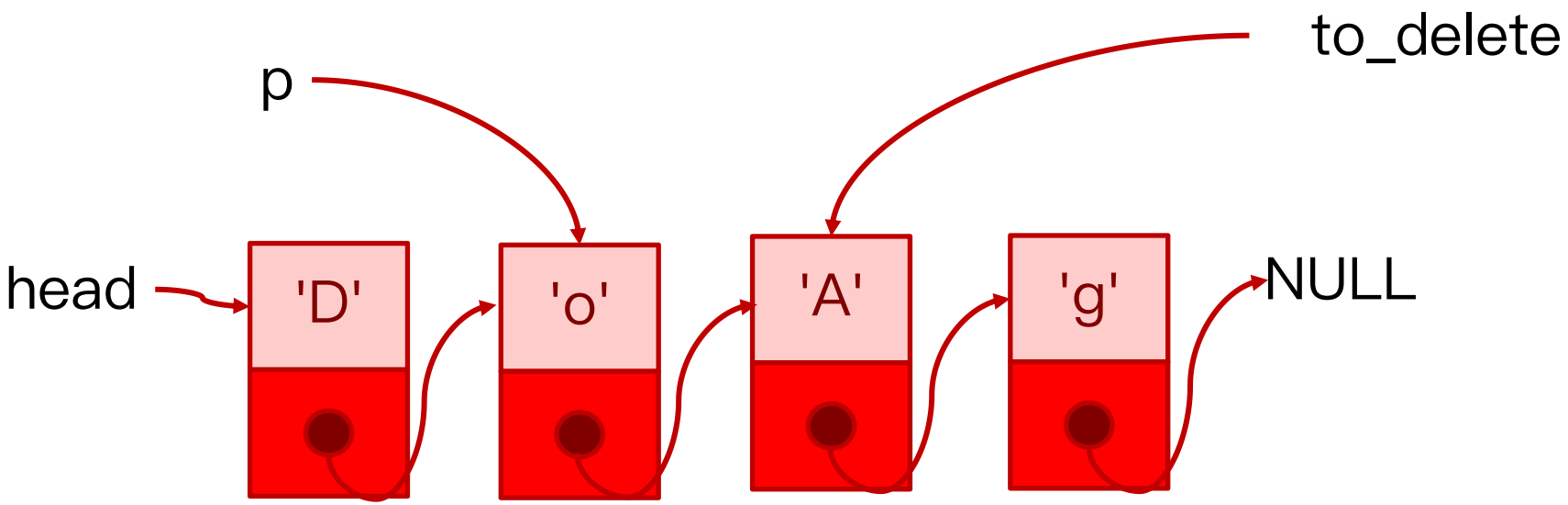
Deleting a Node (Not at Head)

- Illustration: Delete 'A' in between 'o' and 'g'.
  - Traverse p until 'o'



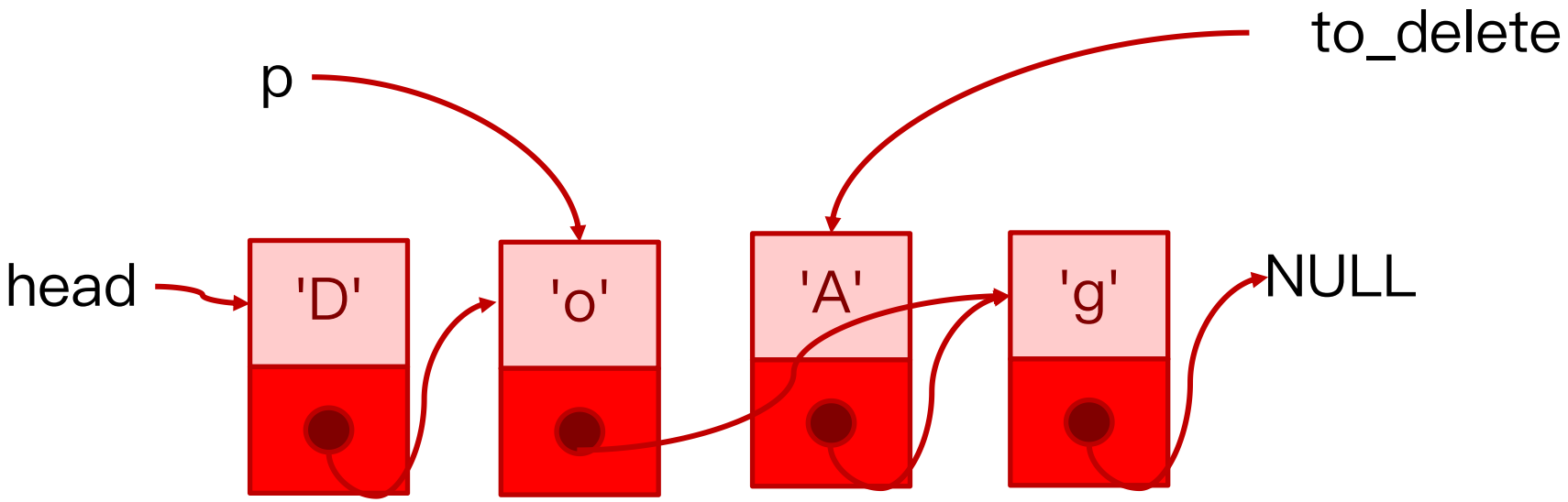
Deleting a Node (Not at Head)

- Illustration: Delete 'A' in between 'o' and 'g'.
  - Traverse p until 'o'
  - Let to_delete point to the next node (the node to be deleted)



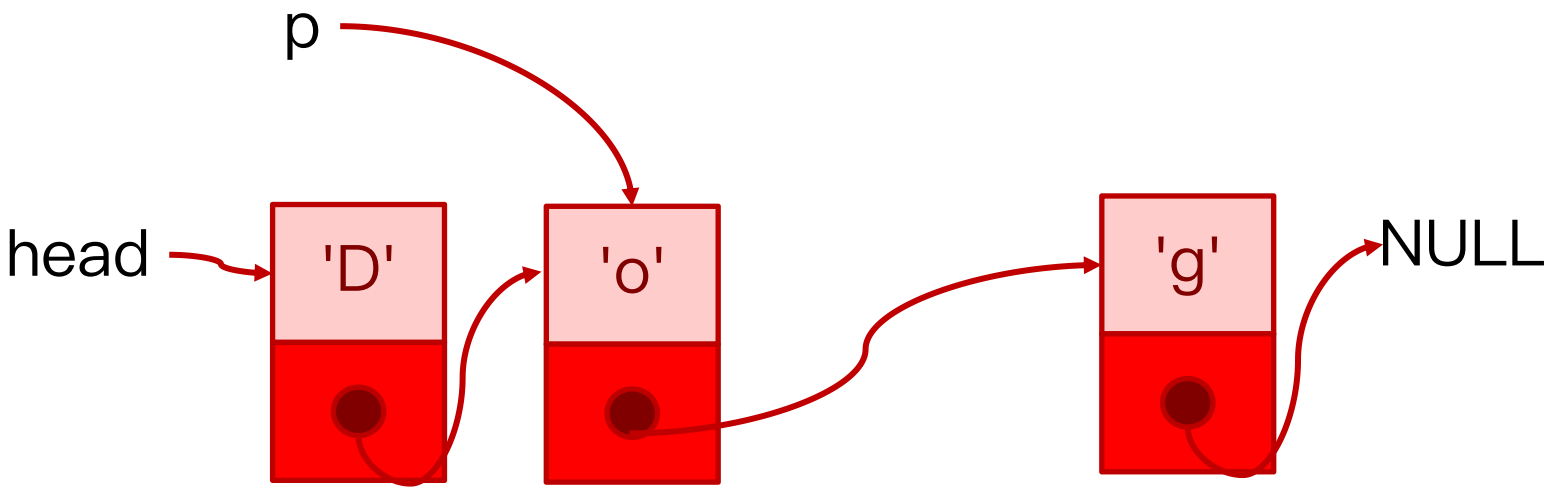
Deleting a Node (Not at Head)

- Illustration: Delete 'A' in between 'o' and 'g'.
  - Traverse p until 'o'
  - Let to_delete point to the next node (the node to be deleted)
  - Delete node



Deleting a Node (Not at Head)

- Illustration: Delete 'A' in between 'o' and 'g'.
  - Traverse p until 'o'
  - Let to_delete point to the next node (the node to be deleted)
  - Delete node



Deleting a Node (Not at Head)

```
Node *to_delete;

Node *p = head;

/* Traverse until desired node */
while(p->next != NULL && p->next->data != 'A')
        p = p->next;

/* Delete */
if(p->next != NULL) {
        to_delete = p->next;
        p->next = to_delete->next;
        free(to_delete);
}
```

Things to Consider

- What if list is empty?
- What if traversal reaches end (NULL)?