

Richard Litt

Scan QR code to add me as a friend.

Week 10 Lecture

XMUT-NWEN 241 - 2026 T2

Systems Programming

Richard Littauer

School of Engineering and Computer Science

Victoria University of Wellington

richard.littauer@vuw.ac.nz

Admin

- Assignment 3
  - Due date is 22 November.

Intro to Linked Lists

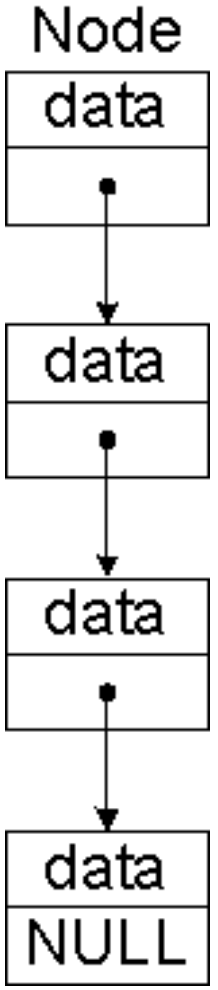
Dynamic Data Structures

- 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

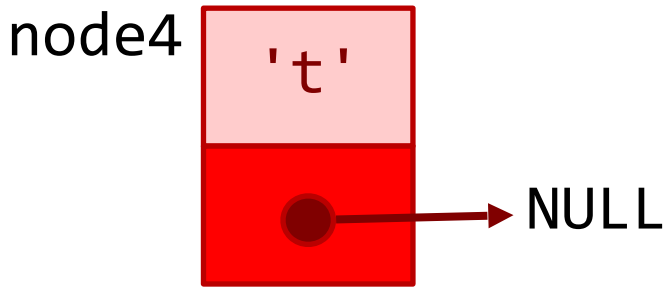
Singly-Linked List (2)

- 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;
```



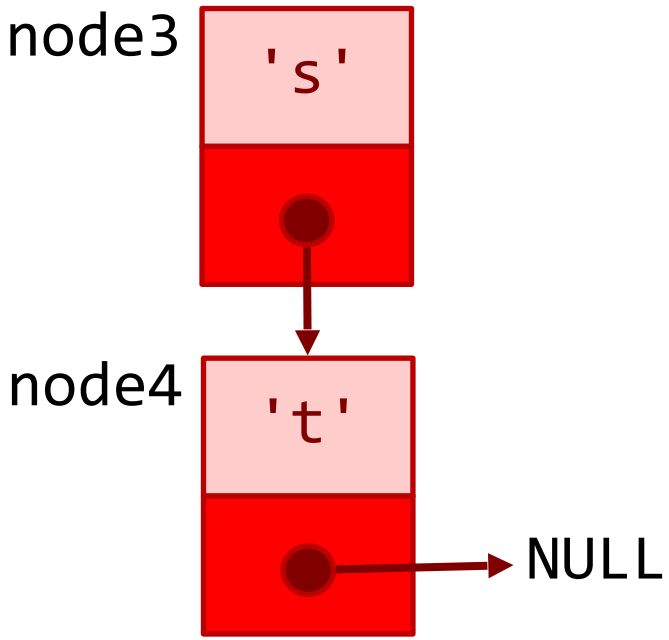
Singly-Linked List (3)

- 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;
```



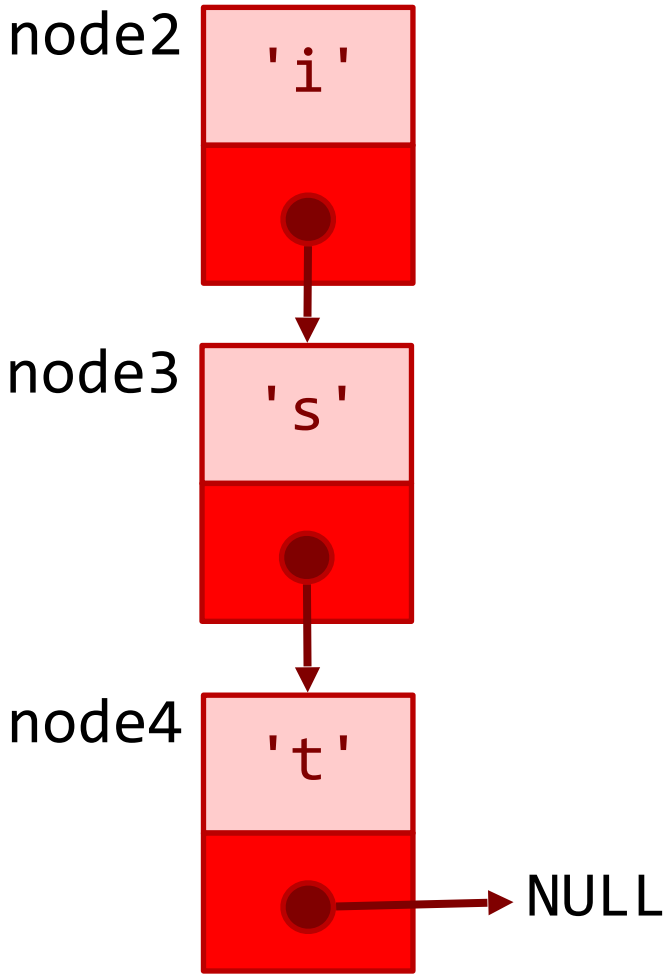
Singly-Linked List (4)

- 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;
```



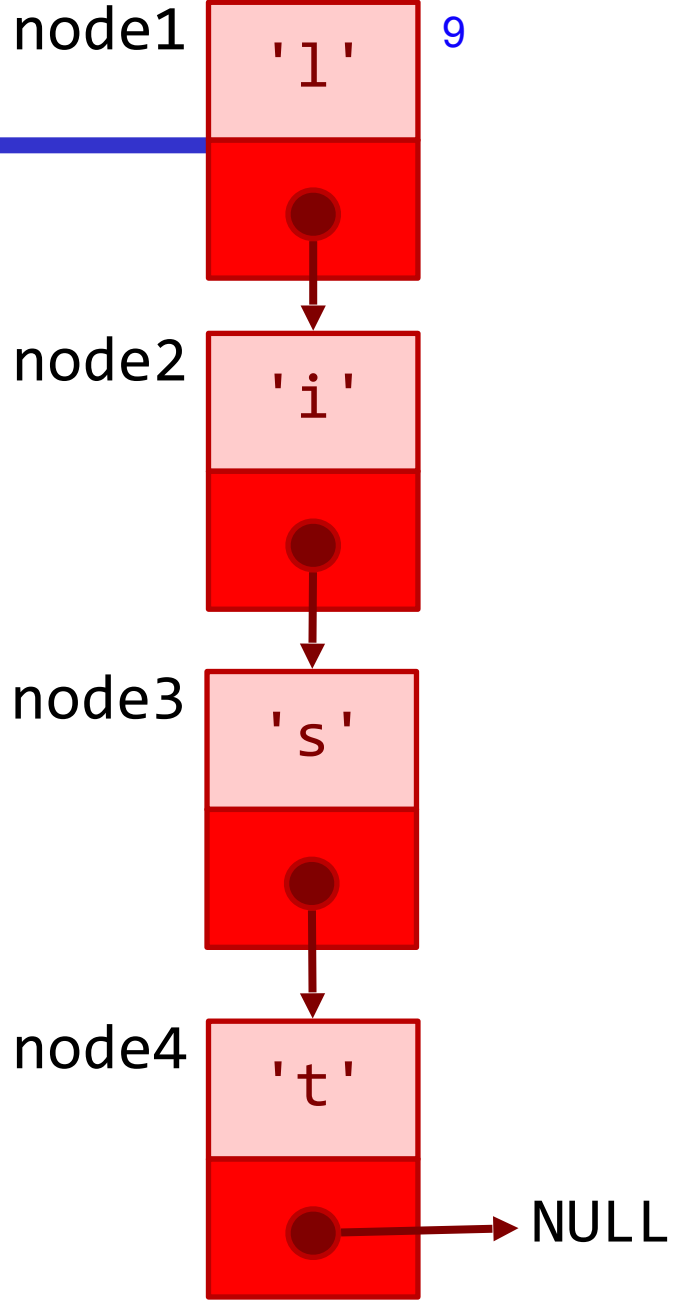
Singly-Linked List (5)

- 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;
```



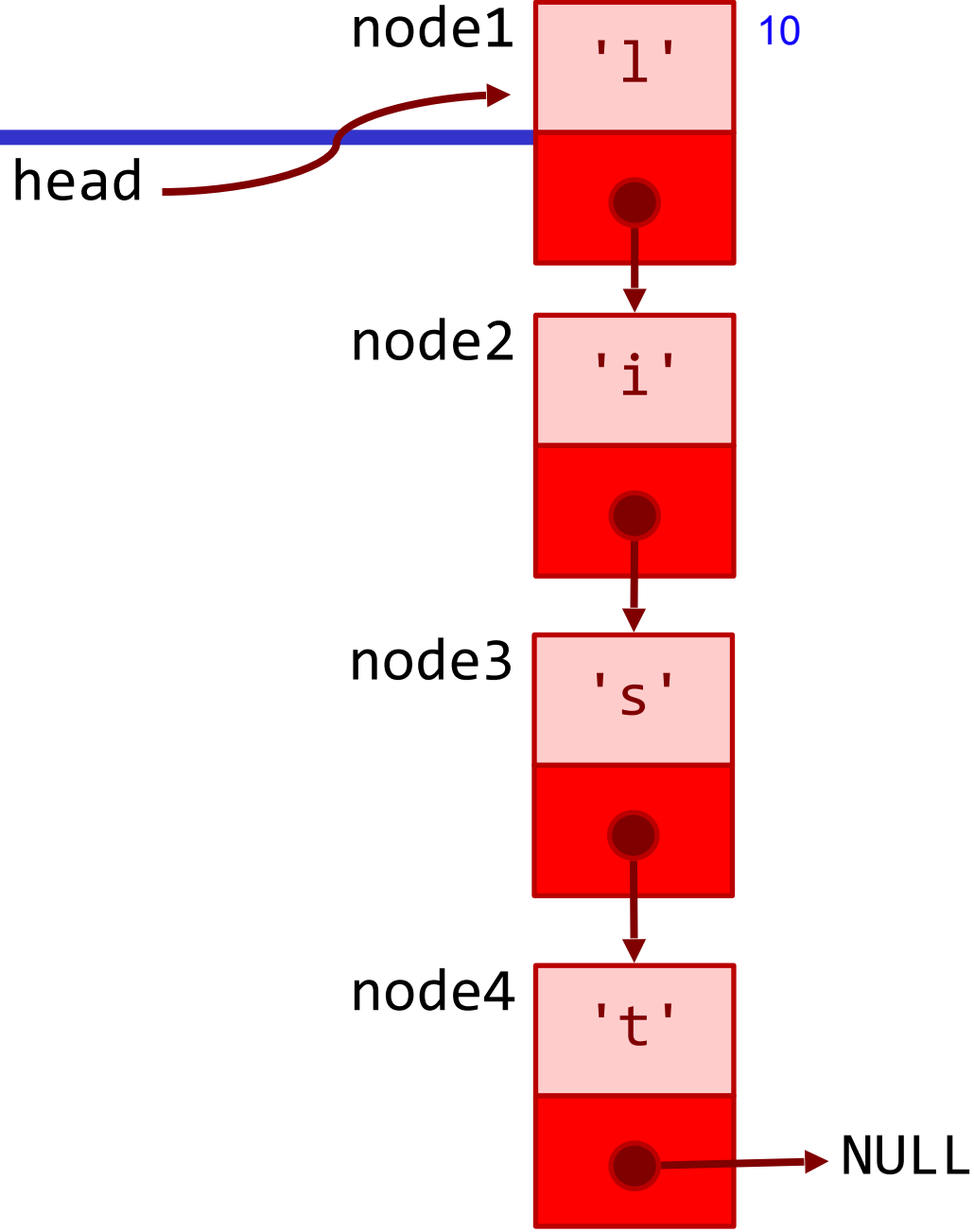
Singly-Linked List (6)

- 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 (1)

- Advantages

- Simple and readable for small, fixed-size lists
- No malloc or free needed (no heap allocation)

- Limitation

- Not flexible — size is fixed at compile-time
- Can't grow/shrink dynamically like malloc-based lists

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

->

- `->` means: follow this pointer to the struct it points at, then get this field from it.

```
p->x
```

- Go to the address in `p`, and in the struct there, take `x`.
  This is the same as:

```
(*pp).x
```

- What does this do?

```
q->front->data
```

Singly-Linked List (1)

- What will this print?

```
Node node4 = {'t', NULL};
Node node3 = {'a', &node4};
Node node2 = {'h', &node3};
Node node1 = {'W', &node2};
Node *head = &node1;

int main(void)
{
    printf("head        = %p\n", (void *) head);
    printf("&node1      = %p\n", (void *) &node1);
    printf("head->data = %c\n", head->data);
    printf("head->next->data = %c\n", head->next->data);
    printf("head->next->next->next->data = %c\n", head->next->next->next->data);
    return 0;
}
```

Singly-Linked List (1)

- Will this work?

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

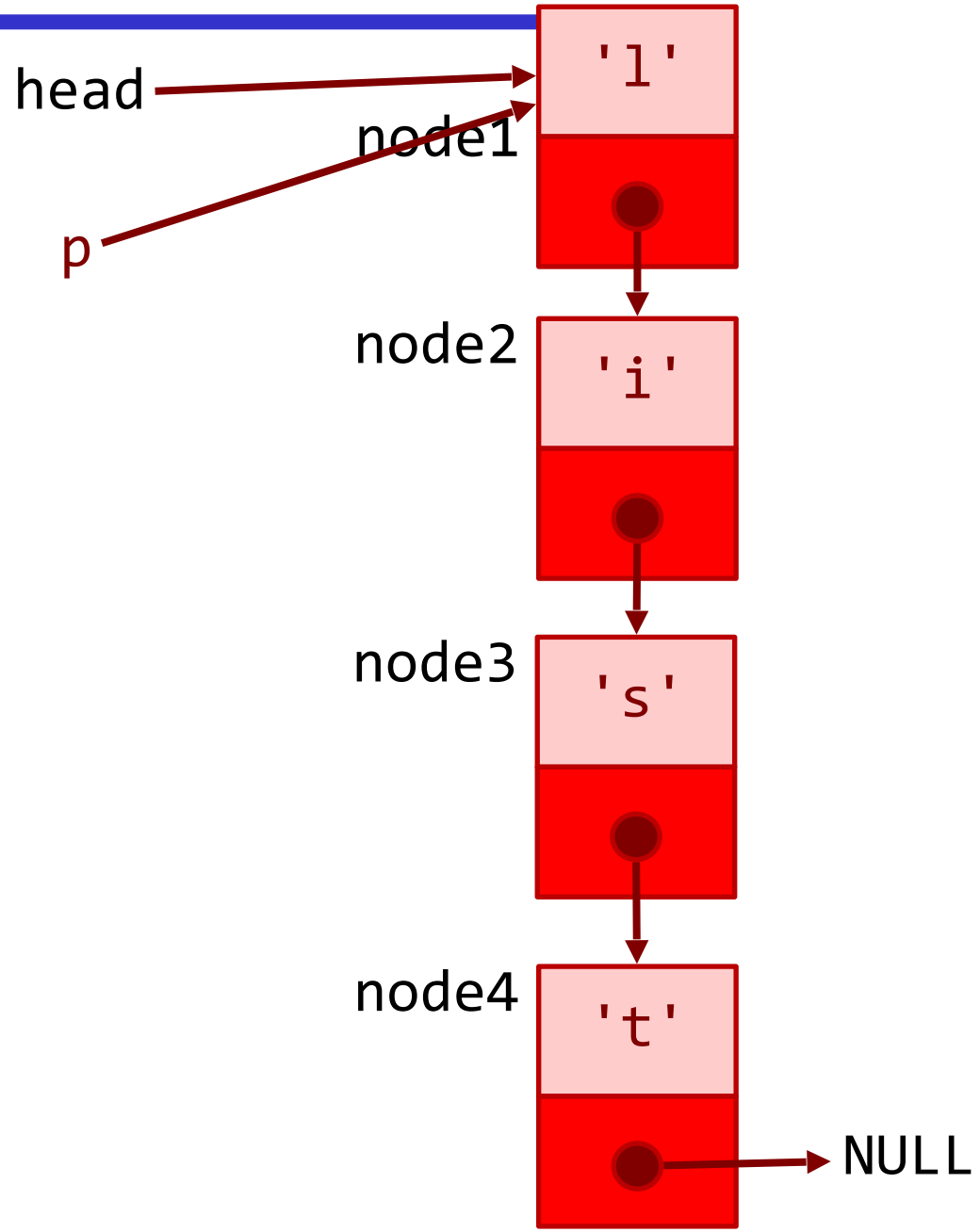
- Will always begin from head
- Visit every node until last node

- Example:

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

- Output:





Singly-Linked List Traversal (3)

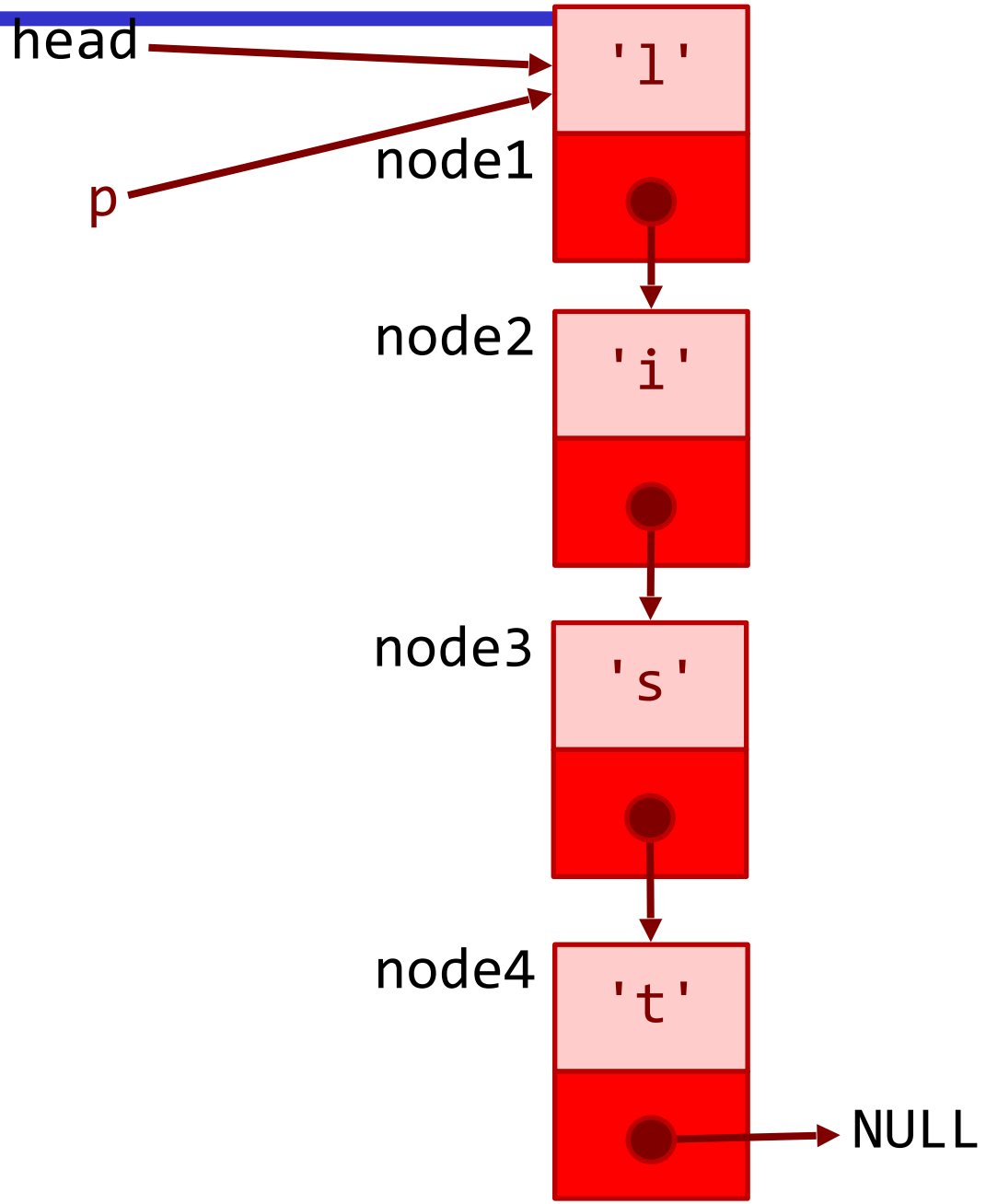
- Will always begin from head
- Visit every node until last node

- Example:

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

- Output:





Singly-Linked List Traversal (4)

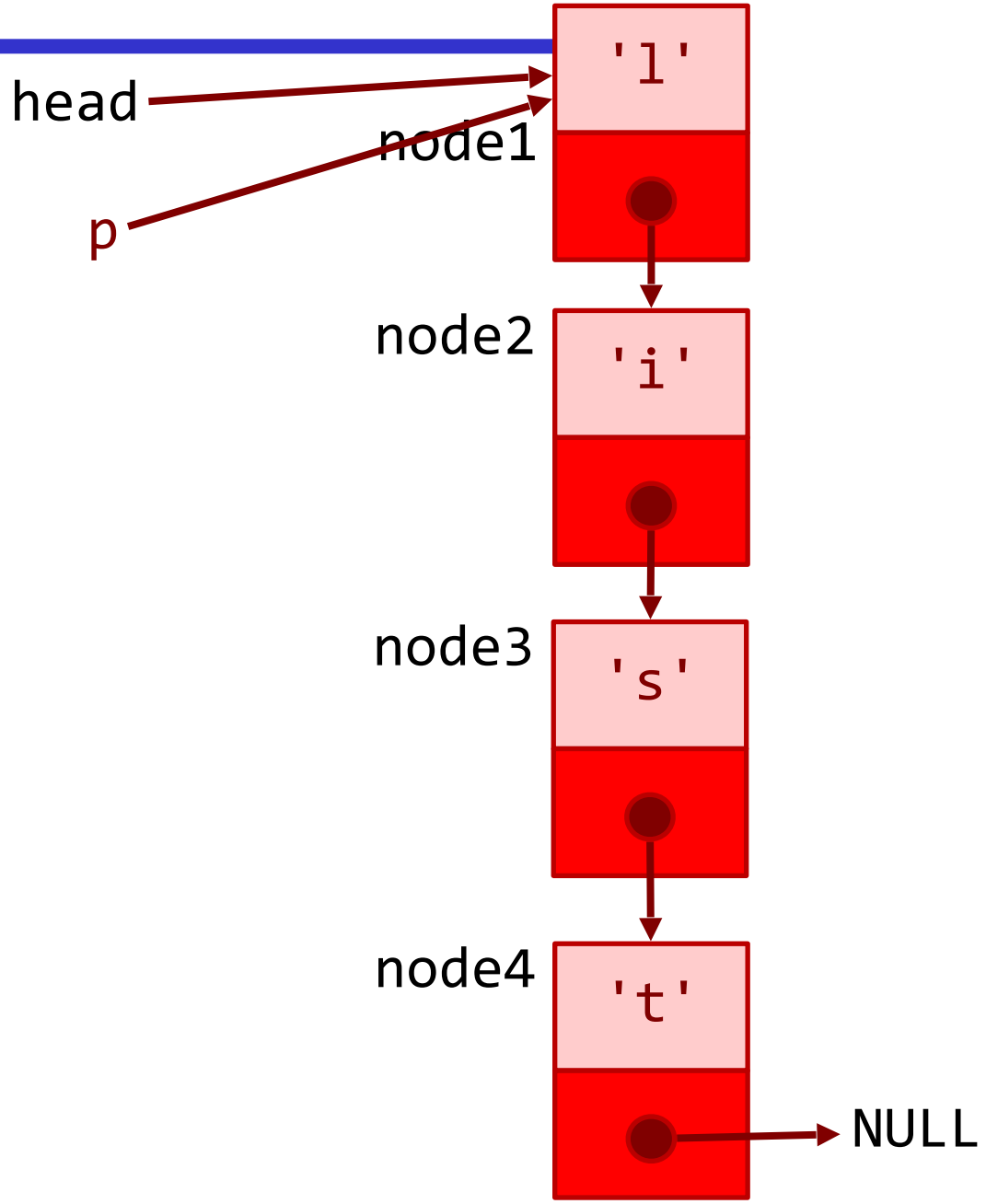
- Will always begin from head
- Visit every node until last node

- Example:

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

- Output:

```
l
```



Singly-Linked List Traversal (5)

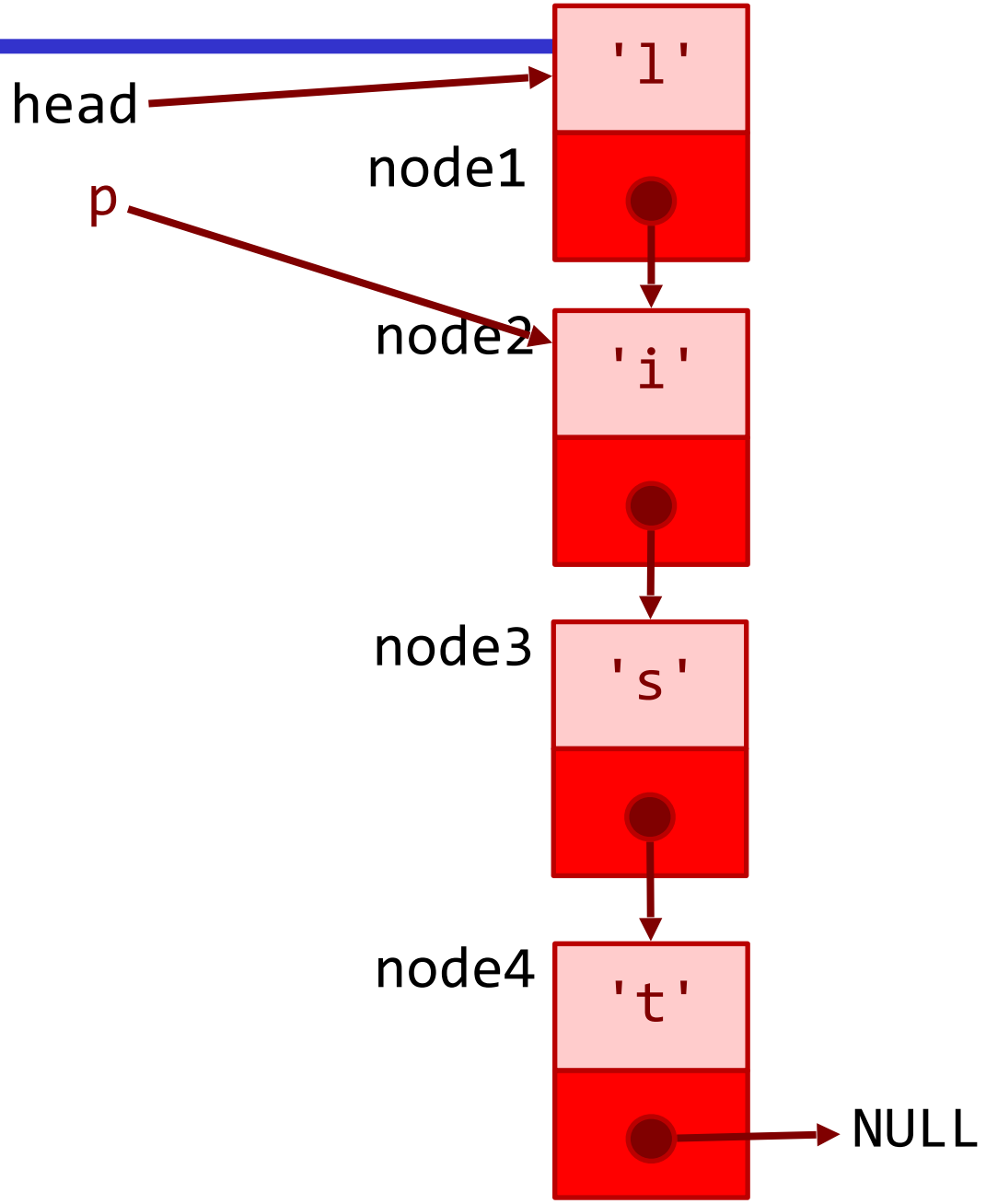
- Will always begin from head
- Visit every node until last node

- Example:

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

- Output:

```
l
```



Singly-Linked List Traversal (6)

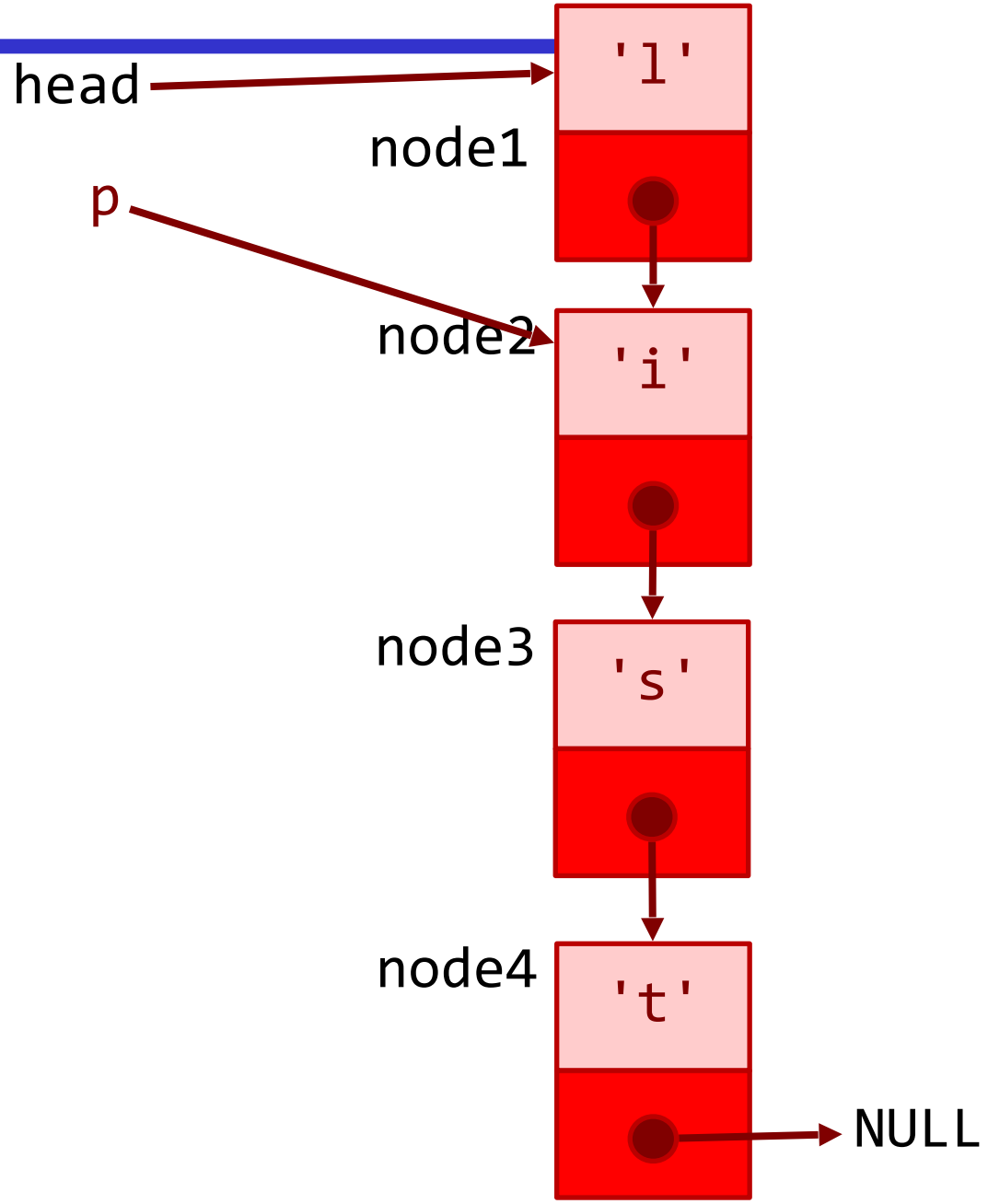
- Will always begin from head
- Visit every node until last node

- Example:

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

- Output:

```
l
```



Singly-Linked List Traversal (7)

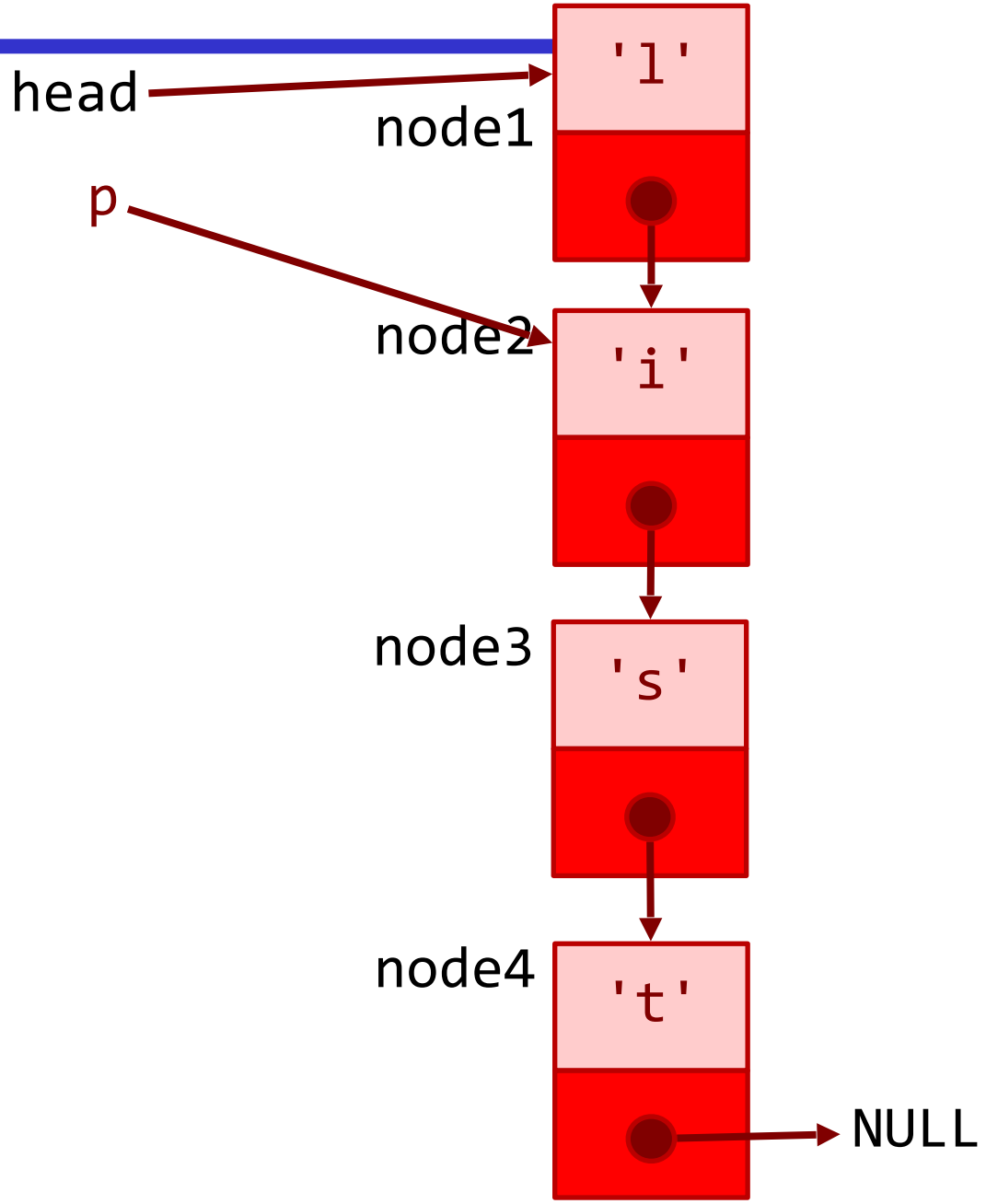
- Will always begin from head
- Visit every node until last node

- Example:

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

- Output:

```
li
```



Singly-Linked List Traversal (8)

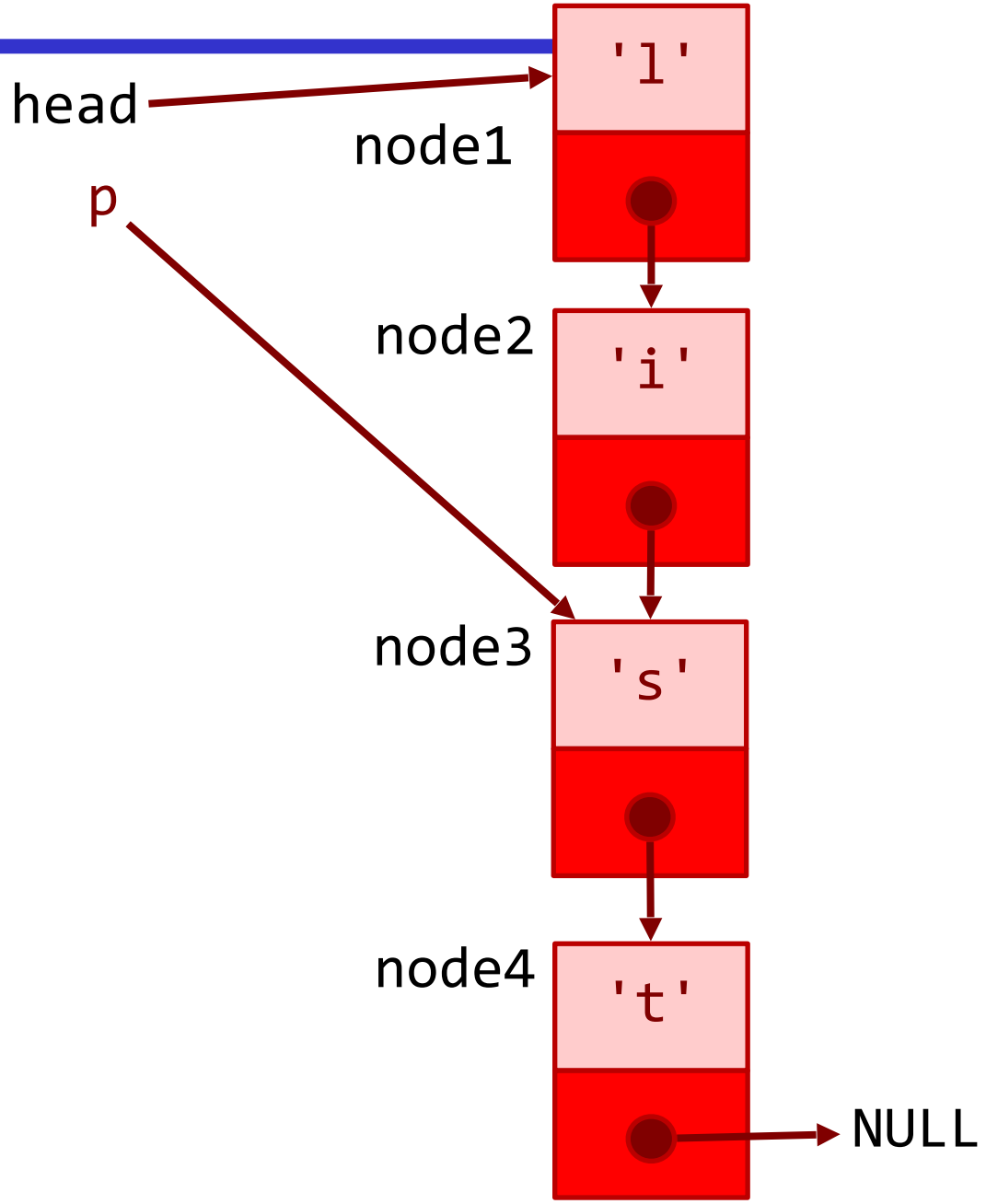
- Will always begin from head
- Visit every node until last node

- Example:

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

- Output:

```
li
```



Singly-Linked List Traversal (9)

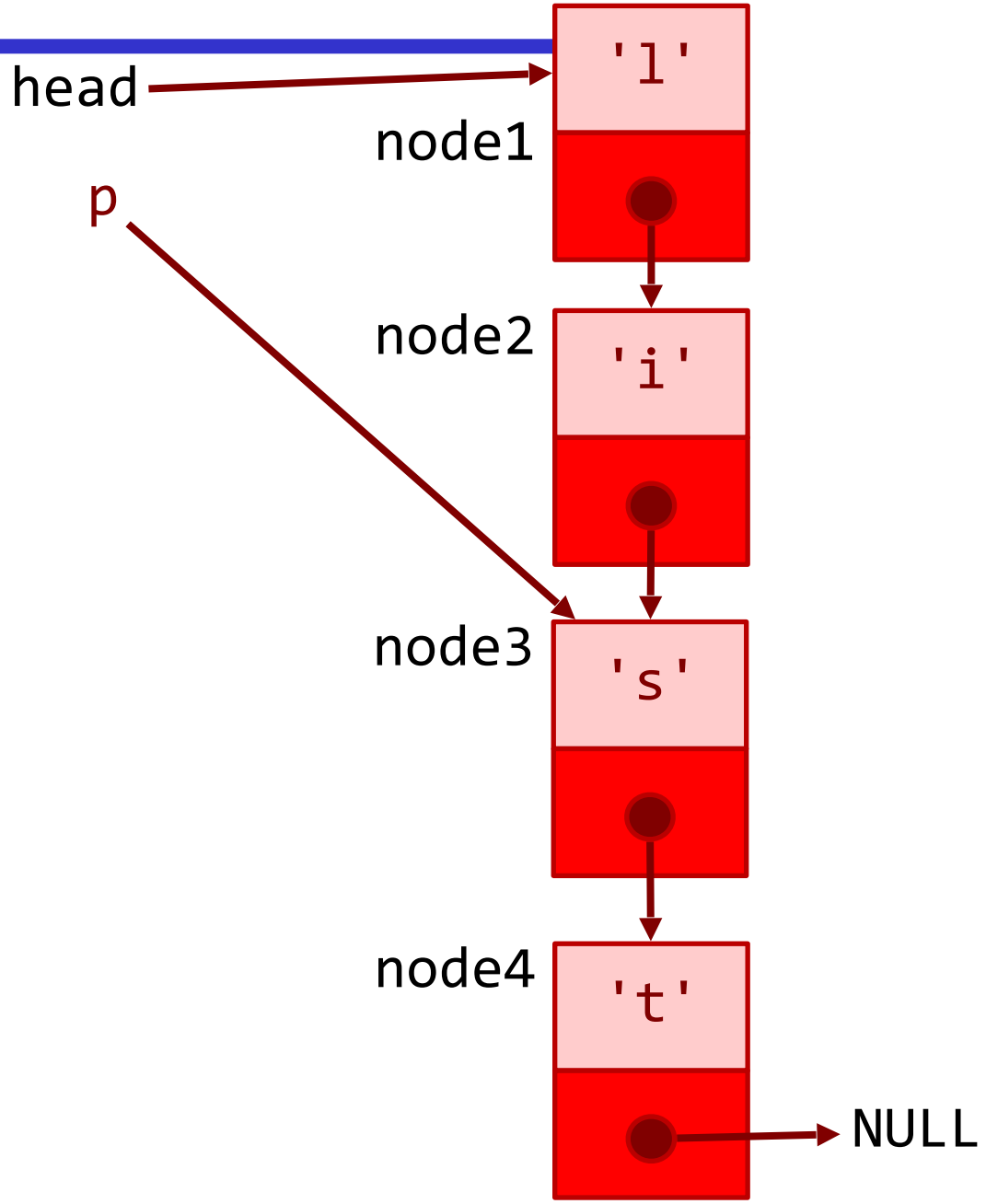
- Will always begin from head
- Visit every node until last node

- Example:

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

- Output:

```
li
```



Singly-Linked List Traversal (10)

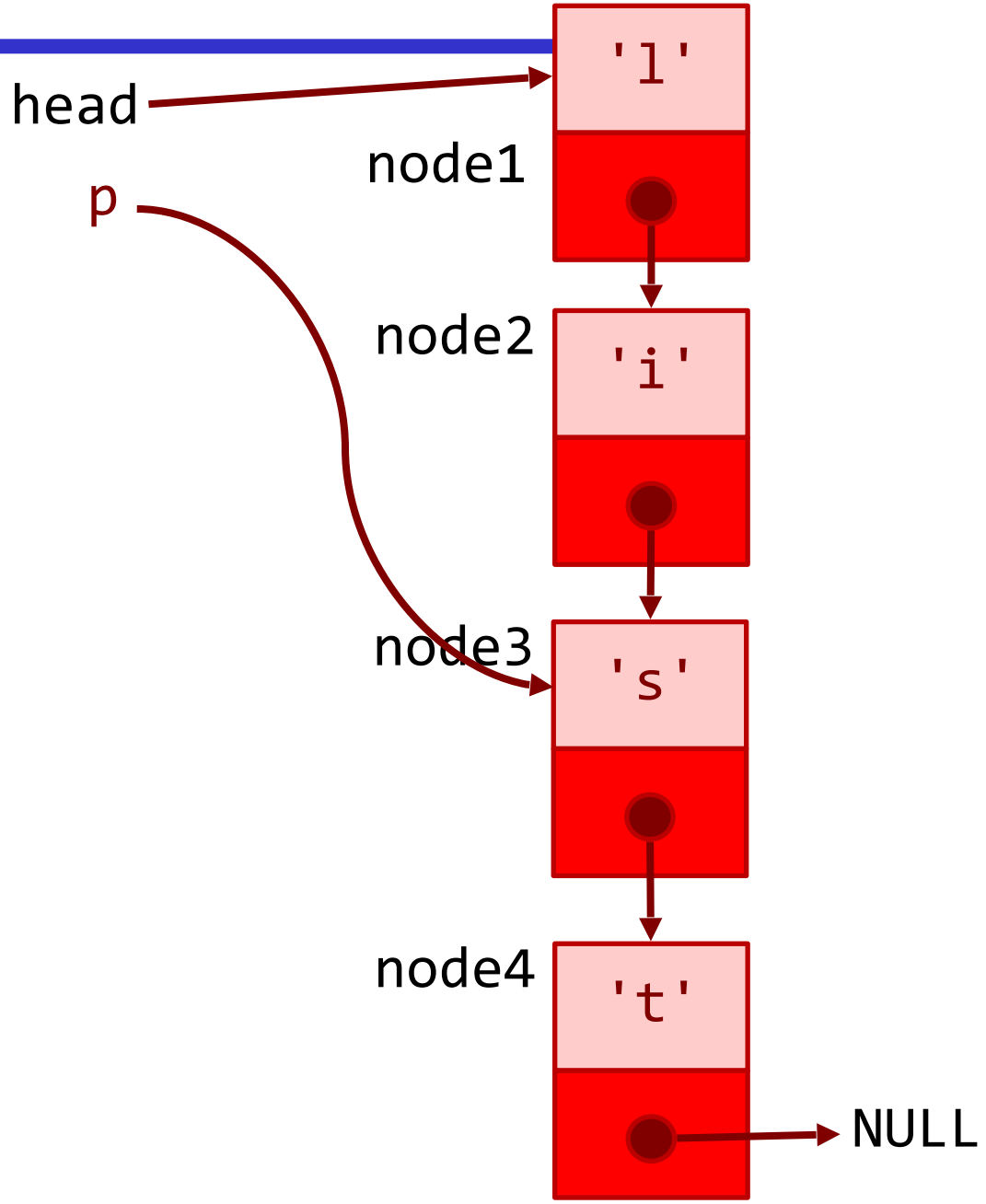
- Will always begin from head
- Visit every node until last node

- Example:

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

- Output:

```
lis
```



Singly-Linked List Traversal (11)

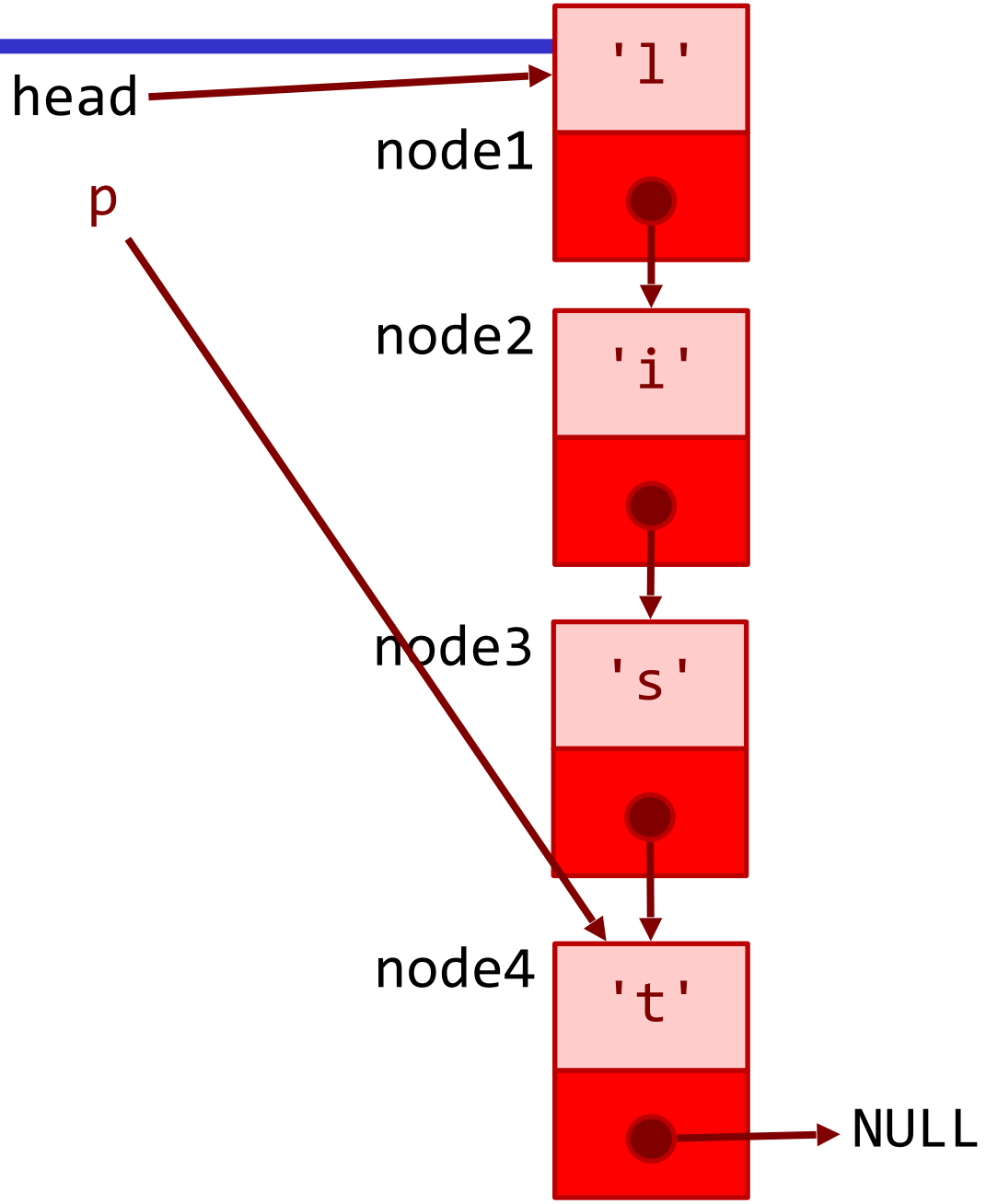
- Will always begin from head
- Visit every node until last node

- Example:

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

- Output:

```
lis
```



Singly-Linked List Traversal (12)

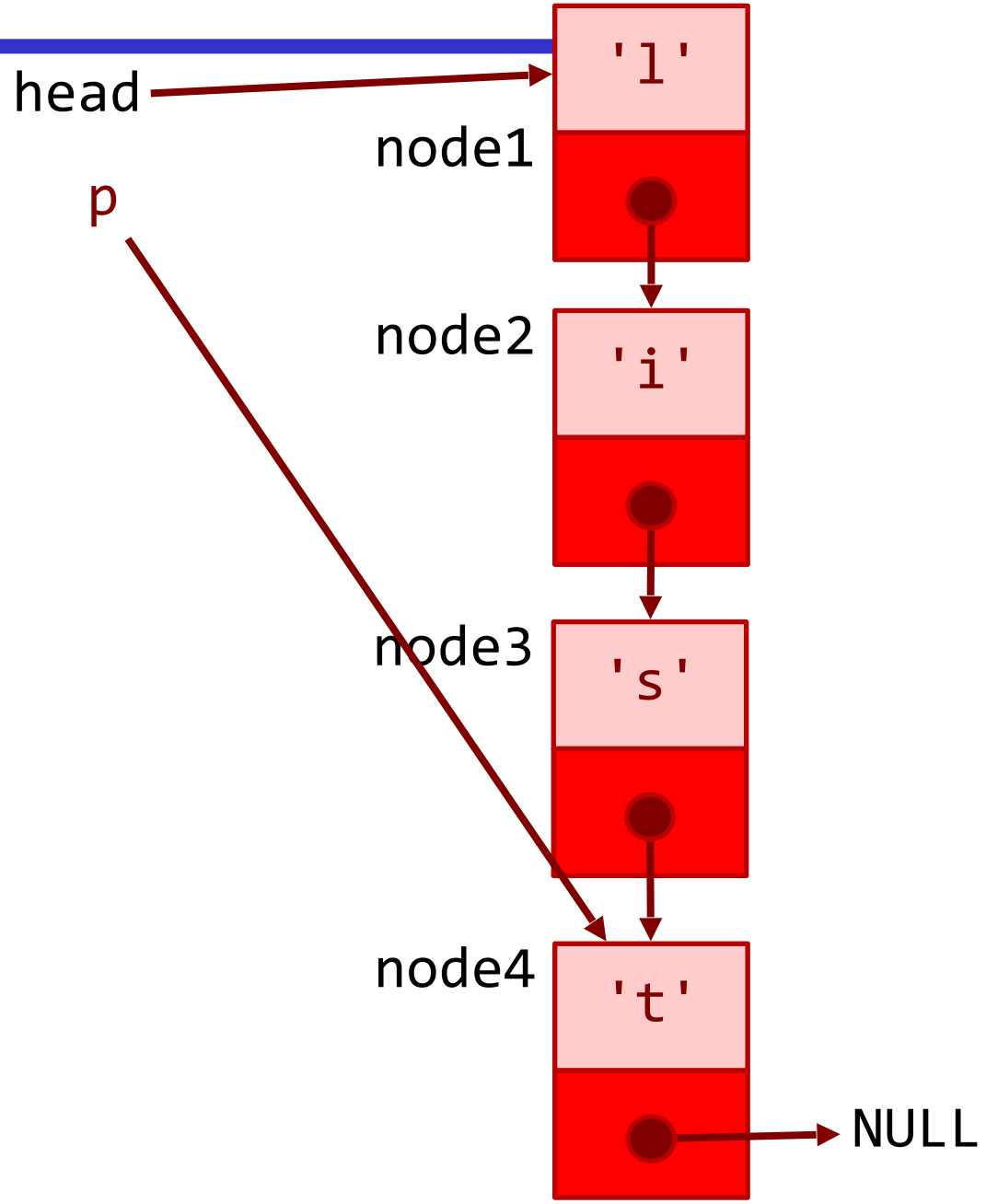
- Will always begin from head
- Visit every node until last node

- Example:

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

- Output:

```
lis
```



Singly-Linked List Traversal (13)

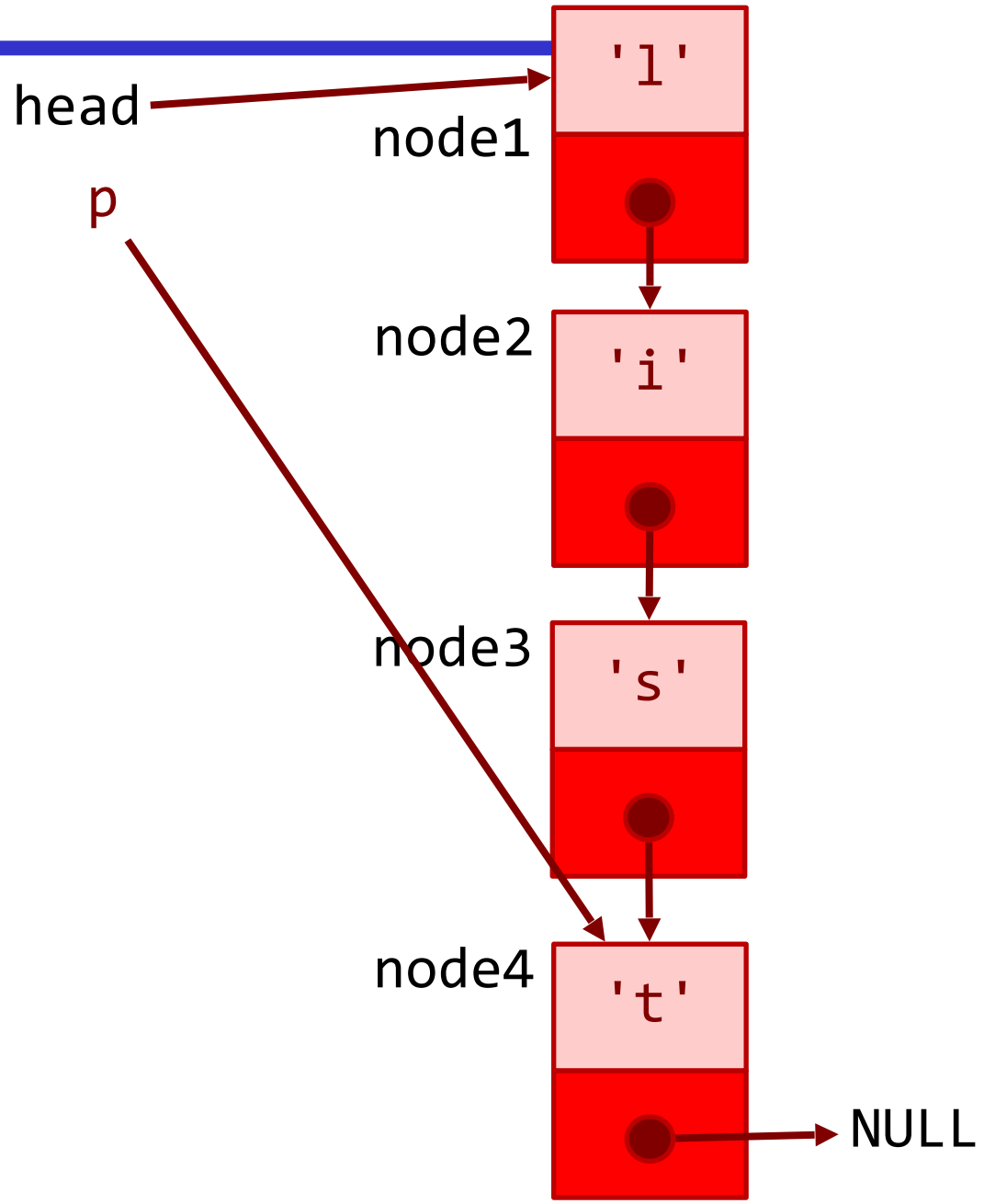
- Will always begin from head
- Visit every node until last node

- Example:

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

- Output:

```
list
```



Singly-Linked List Traversal (14)

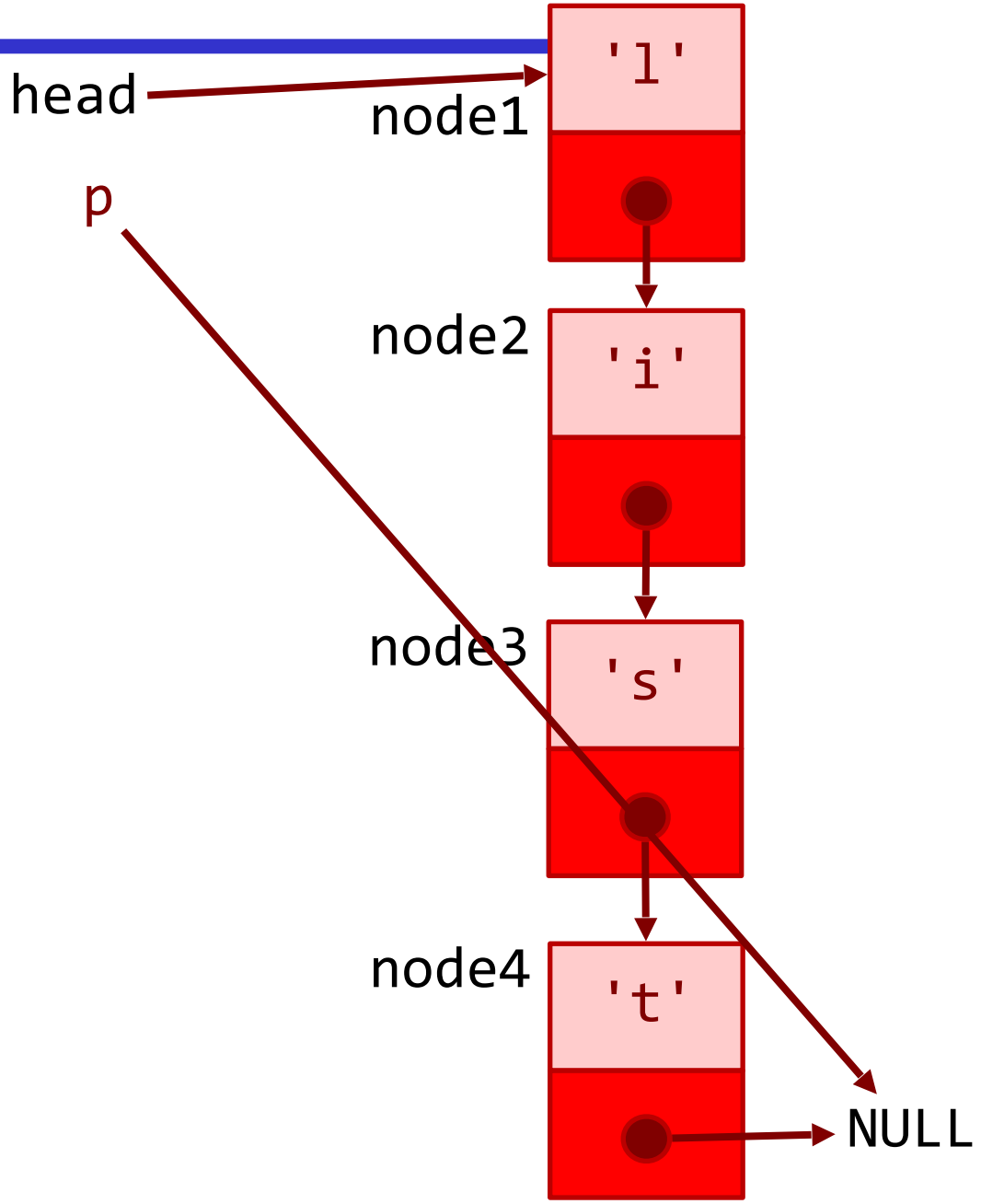
- Will always begin from head
- Visit every node until last node

- Example:

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

- Output:

```
list
```



Singly-Linked List Traversal (15)

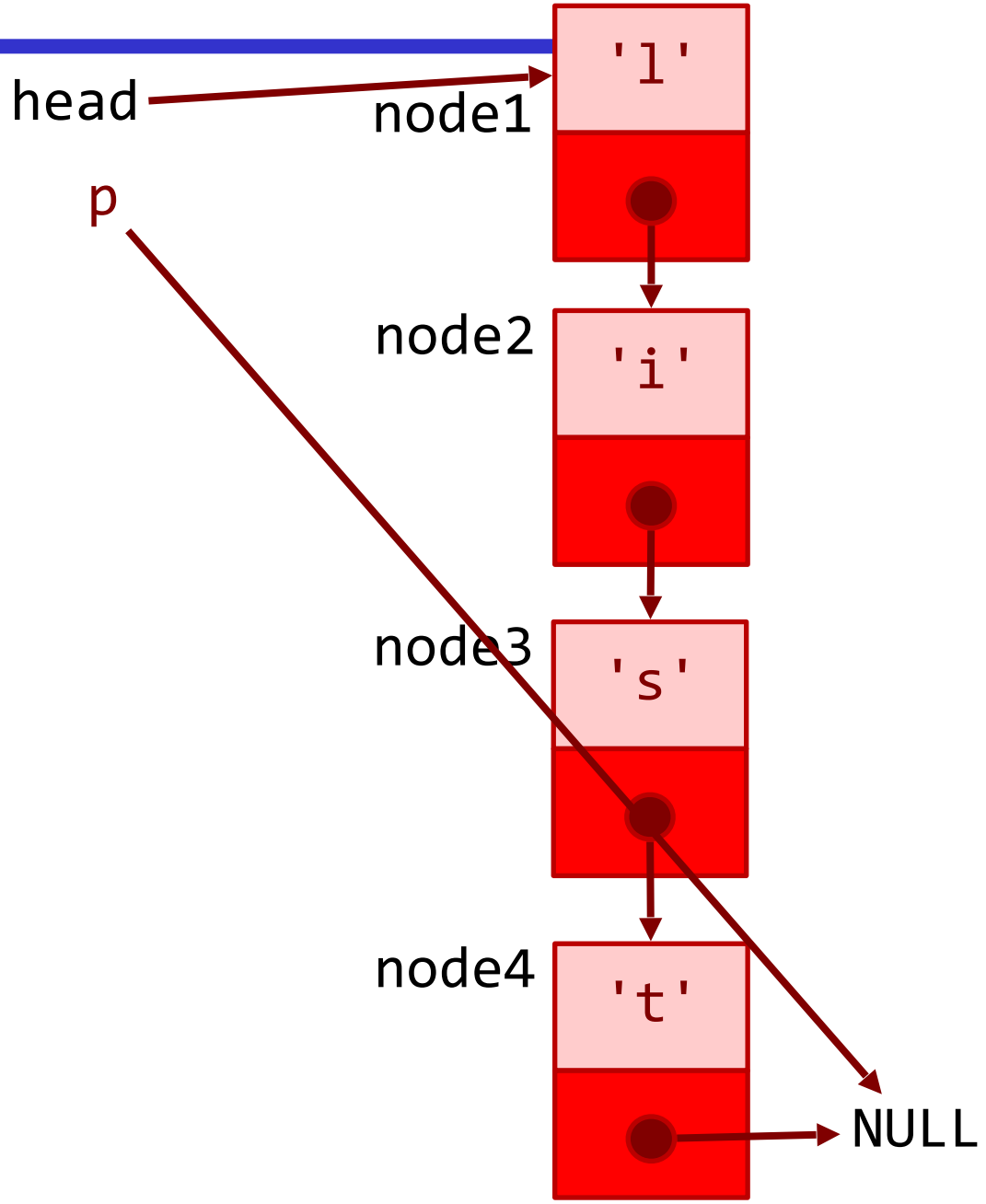
- Will always begin from head
- Visit every node until last node

- Example:

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

- Output:

```
list
```



Previous example

- Can also be run with while

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

int main(void)
{
    Node *p = head;

    while(p != NULL) {
        printf("%c", p->data);
        p = p->next;
    }
    printf("\n");
    return 0;
}
```

- Or by traversing head

```
for( ; head != NULL; head = head->next)
    printf("%c", head->data);
printf("\n");
```

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

Structs elsewhere

- Should this work?

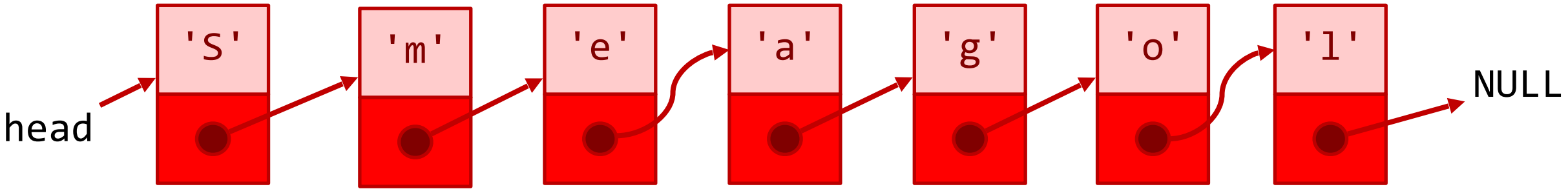
```
Node global_node = {'g', NULL};

int main(void)
{
    Node local_node = {'l', NULL};
    Node *heap_node = (Node *)malloc(sizeof(Node));

    ...

    free(heap_node);
    return 0;
}
```

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;
```

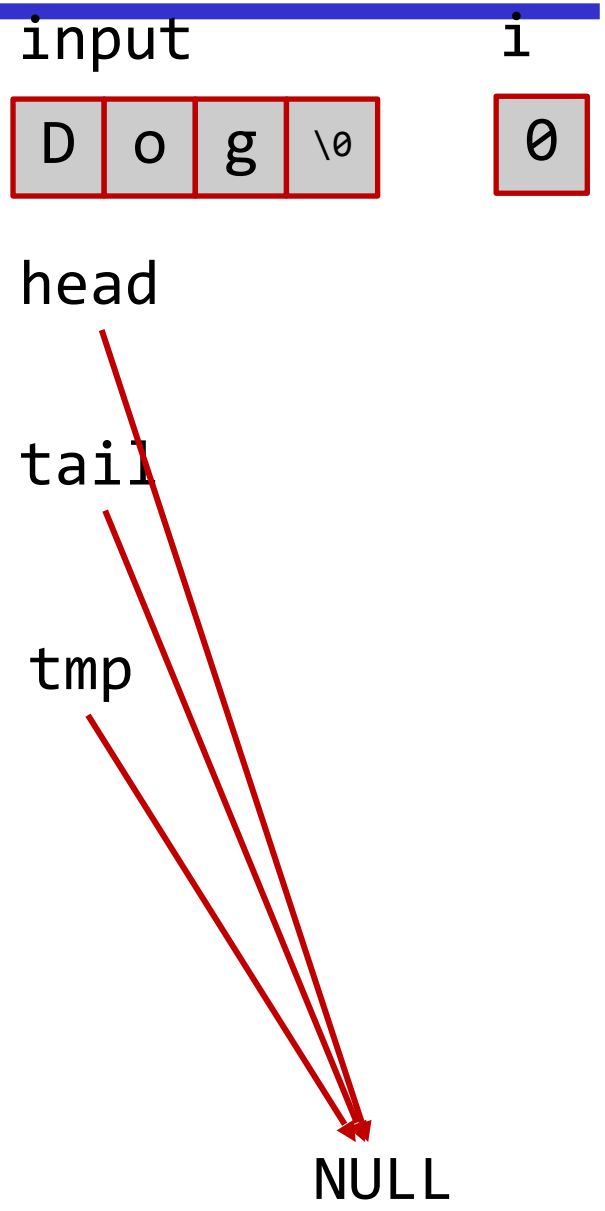
The Rest of The Code

```
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);

…
```

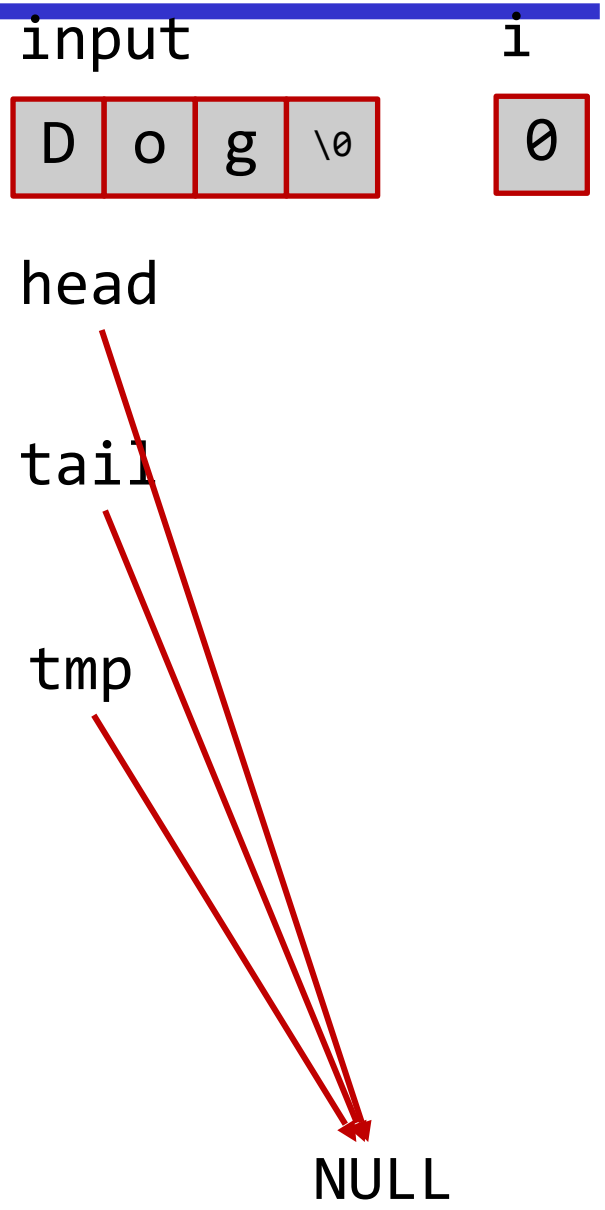


Walkthrough

- 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++;
}
```

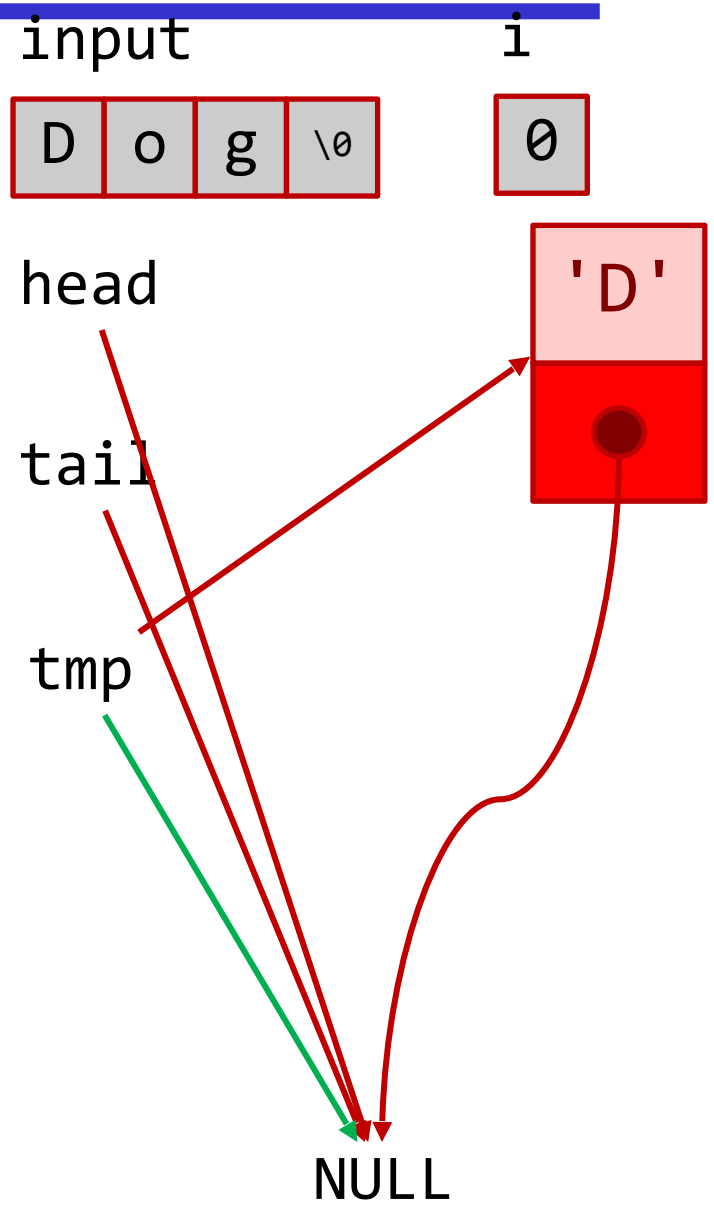
While input not "\0"



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++;
}
```

`(Node *)` = Type casting Node as a pointer

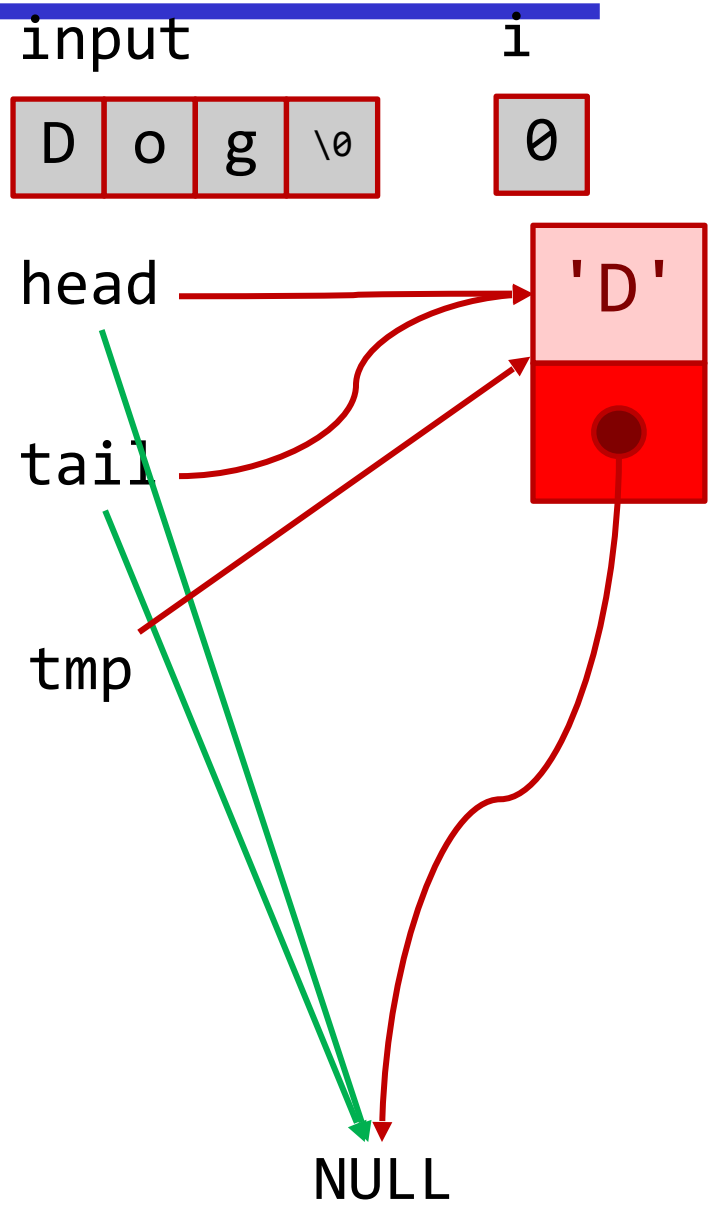
`malloc(sizeof(Node)`

- `malloc` finds a free block of memory on the **heap** large enough to store one `Node` (usually 8 to 16 bytes depending on system).
- It returns a `void*` (generic pointer), which points to the starting address of that block.

Walkthrough

- Suppose user input is Dog

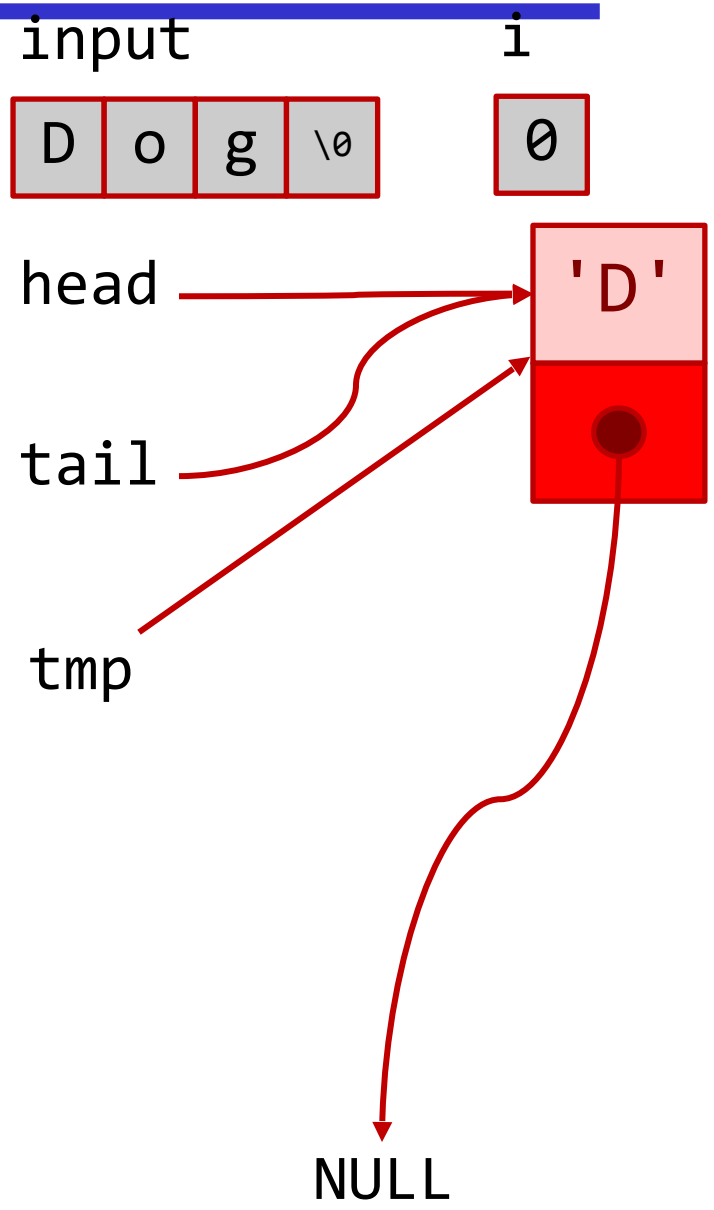
```
while(input[i] != '\0') {
    tmp = (Node *)malloc(sizeof(Node));
    tmp->data = input[i];
    tmp->next = NULL;
    if(head == NULL) {
        head = tmp;
        tail = head;
    } else {
        tail->next = tmp;
        tail = tmp;
    }
    i++;
}
```



Walkthrough

- Suppose user input is Dog

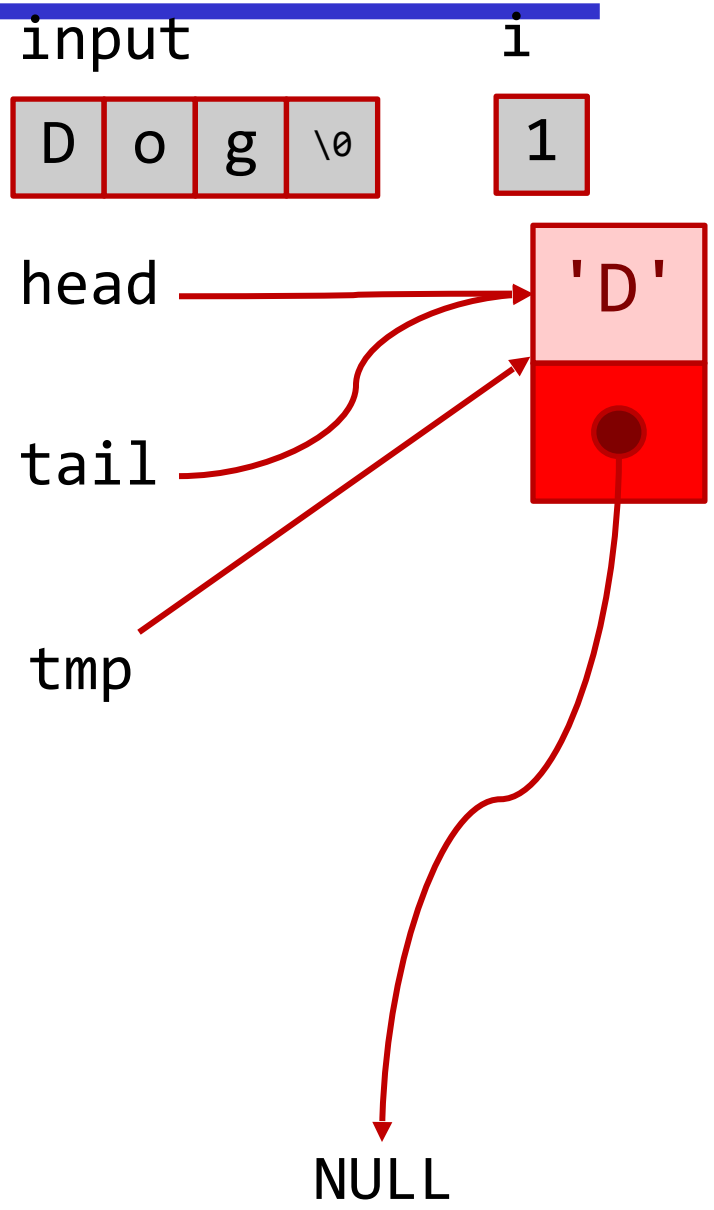
```
while(input[i] != '\0') {
    tmp = (Node *)malloc(sizeof(Node));
    tmp->data = input[i];
    tmp->next = NULL;
    if(head == NULL) {
        head = tmp;
        tail = head;
    } else {
        tail->next = tmp;
        tail = tmp;
    }
    i++;
}
```



Walkthrough

- Suppose user input is Dog

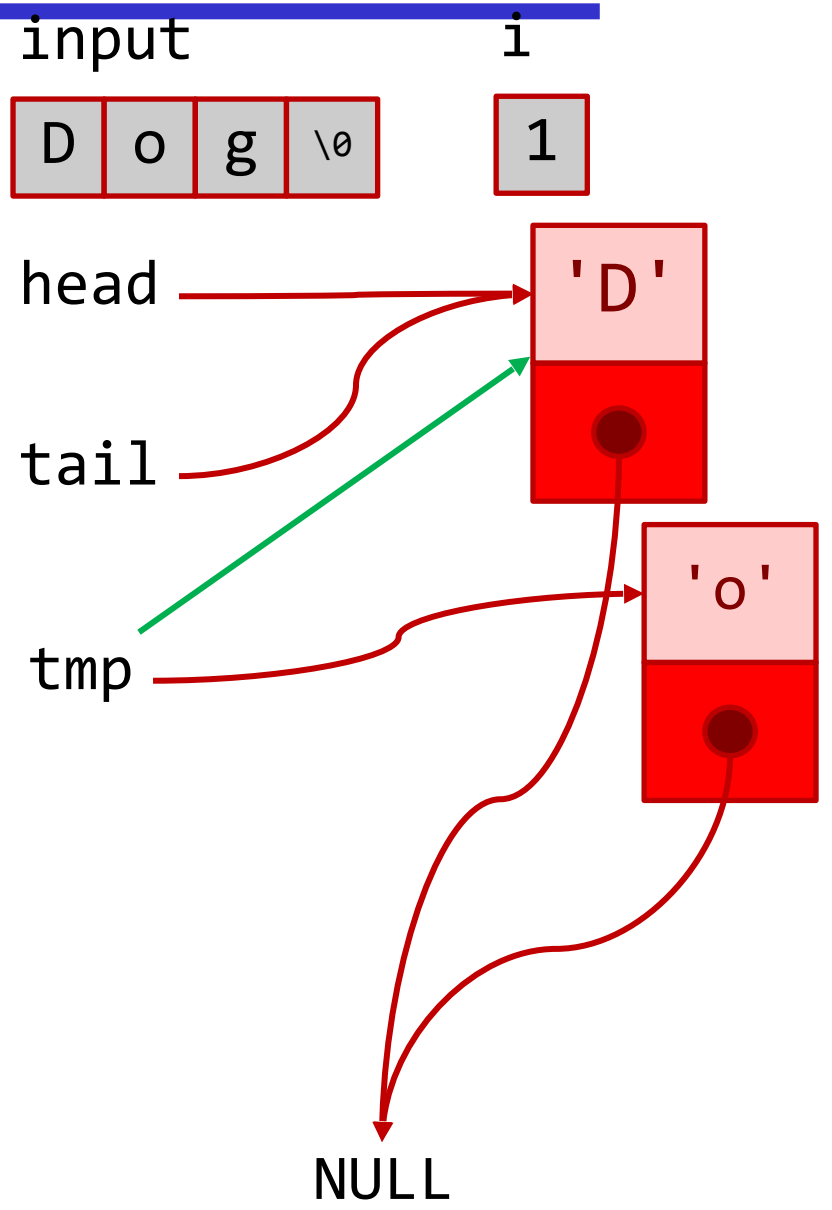
```
while(input[i] != '\0') {
    tmp = (Node *)malloc(sizeof(Node));
    tmp->data = input[i];
    tmp->next = NULL;
    if(head == NULL) {
        head = tmp;
        tail = head;
    } else {
        tail->next = tmp;
        tail = tmp;
    }
    i++;
}
```



Walkthrough

- Suppose user input is Dog

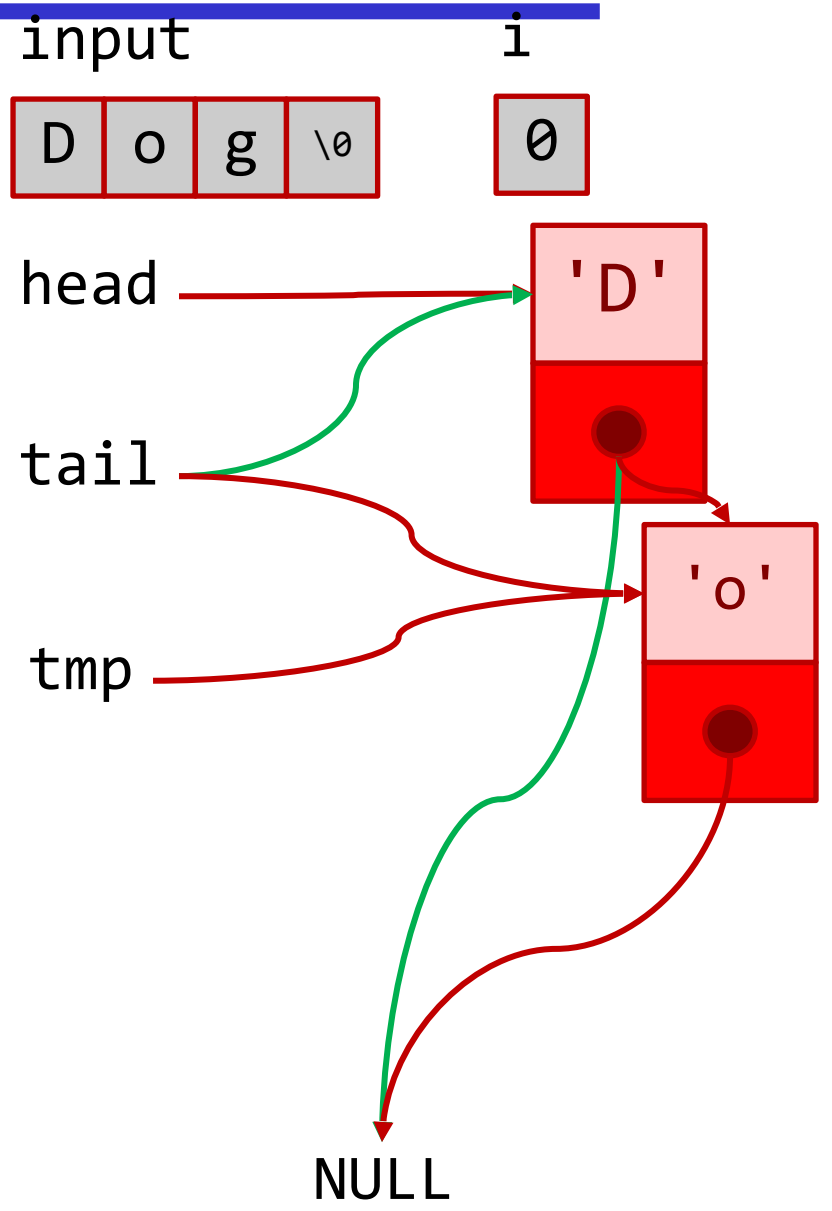
```
while(input[i] != '\0') {
    tmp = (Node *)malloc(sizeof(Node));
    tmp->data = input[i];
    tmp->next = NULL;
    if(head == NULL) {
        head = tmp;
        tail = head;
    } else {
        tail->next = tmp;
        tail = tmp;
    }
    i++;
}
```



Walkthrough

- Suppose user input is Dog

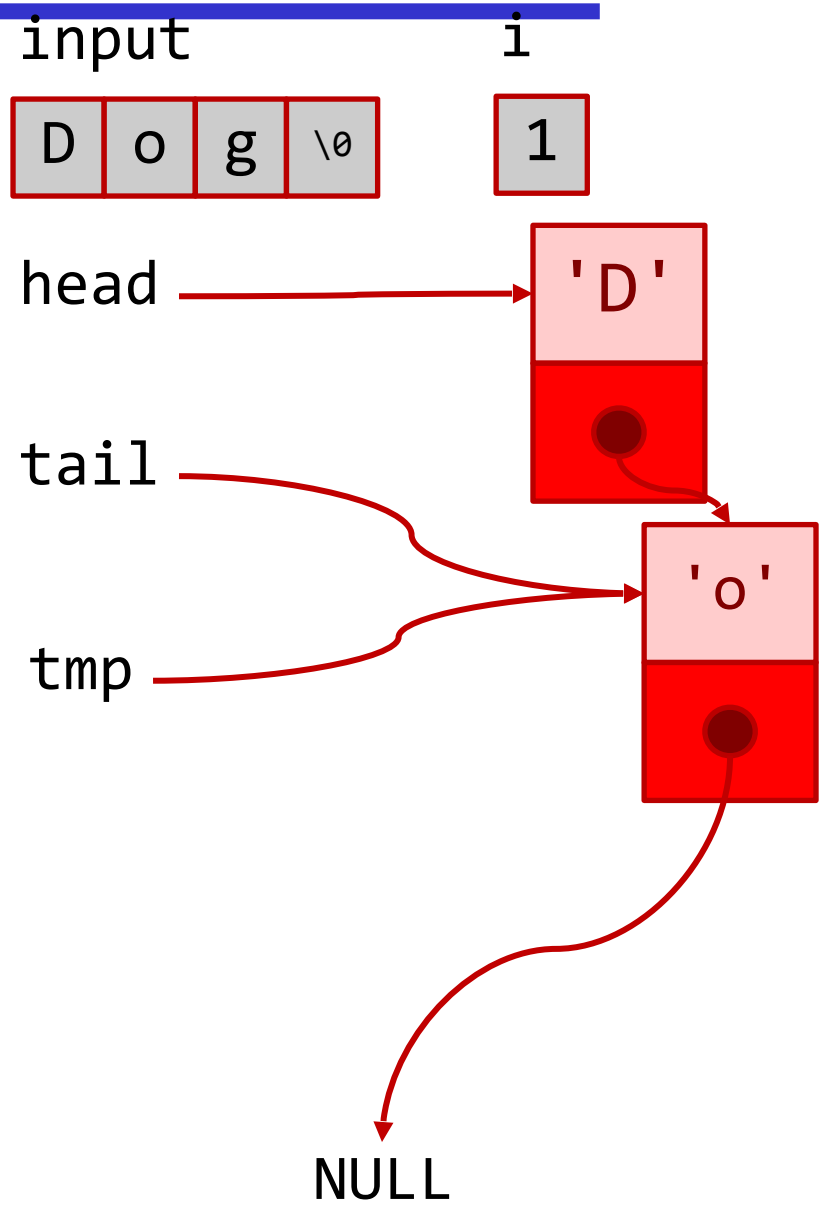
```
while(input[i] != '\0') {
    tmp = (Node *)malloc(sizeof(Node));
    tmp->data = input[i];
    tmp->next = NULL;
    if(head == NULL) {
        head = tmp;
        tail = head;
    } else {
        tail->next = tmp;
        tail = tmp;
    }
    i++;
}
```



Walkthrough

- Suppose user input is Dog

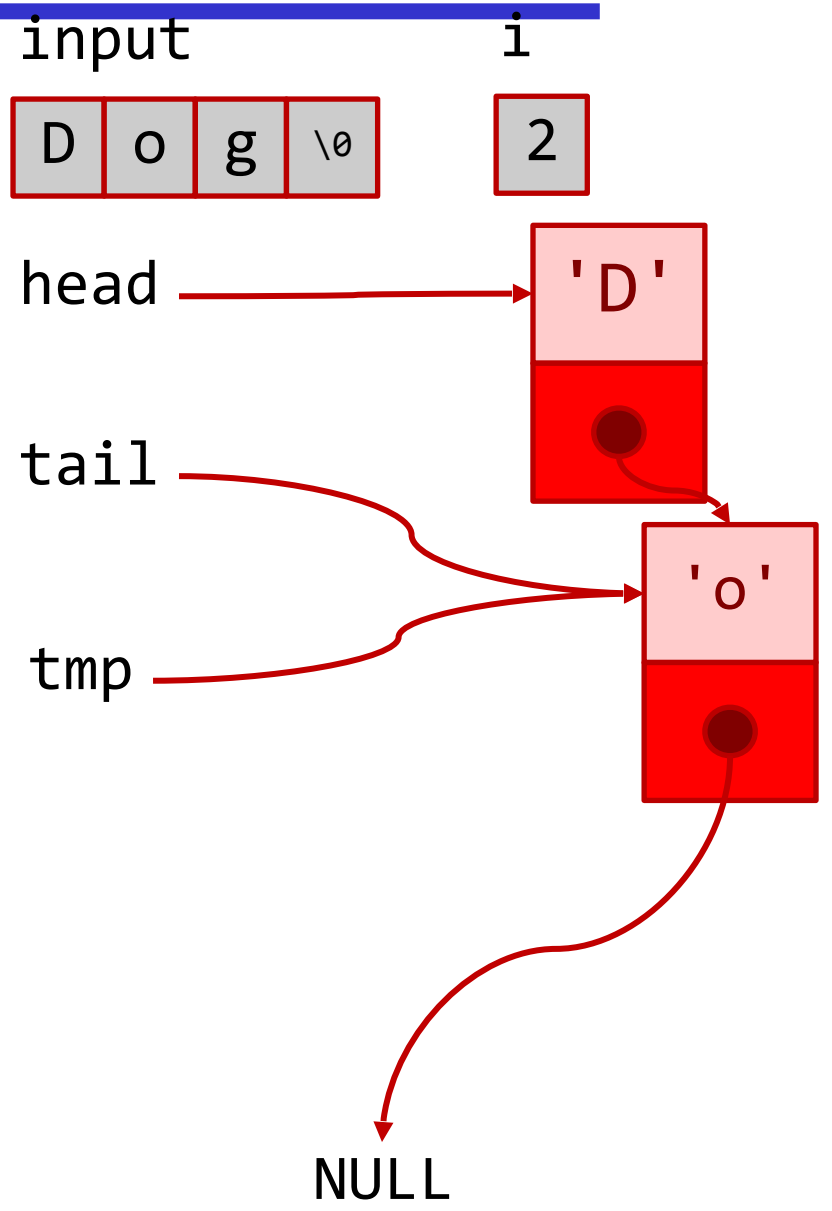
```
while(input[i] != '\0') {
    tmp = (Node *)malloc(sizeof(Node));
    tmp->data = input[i];
    tmp->next = NULL;
    if(head == NULL) {
        head = tmp;
        tail = head;
    } else {
        tail->next = tmp;
        tail = tmp;
    }
    i++;
}
```



Walkthrough

- Suppose user input is Dog

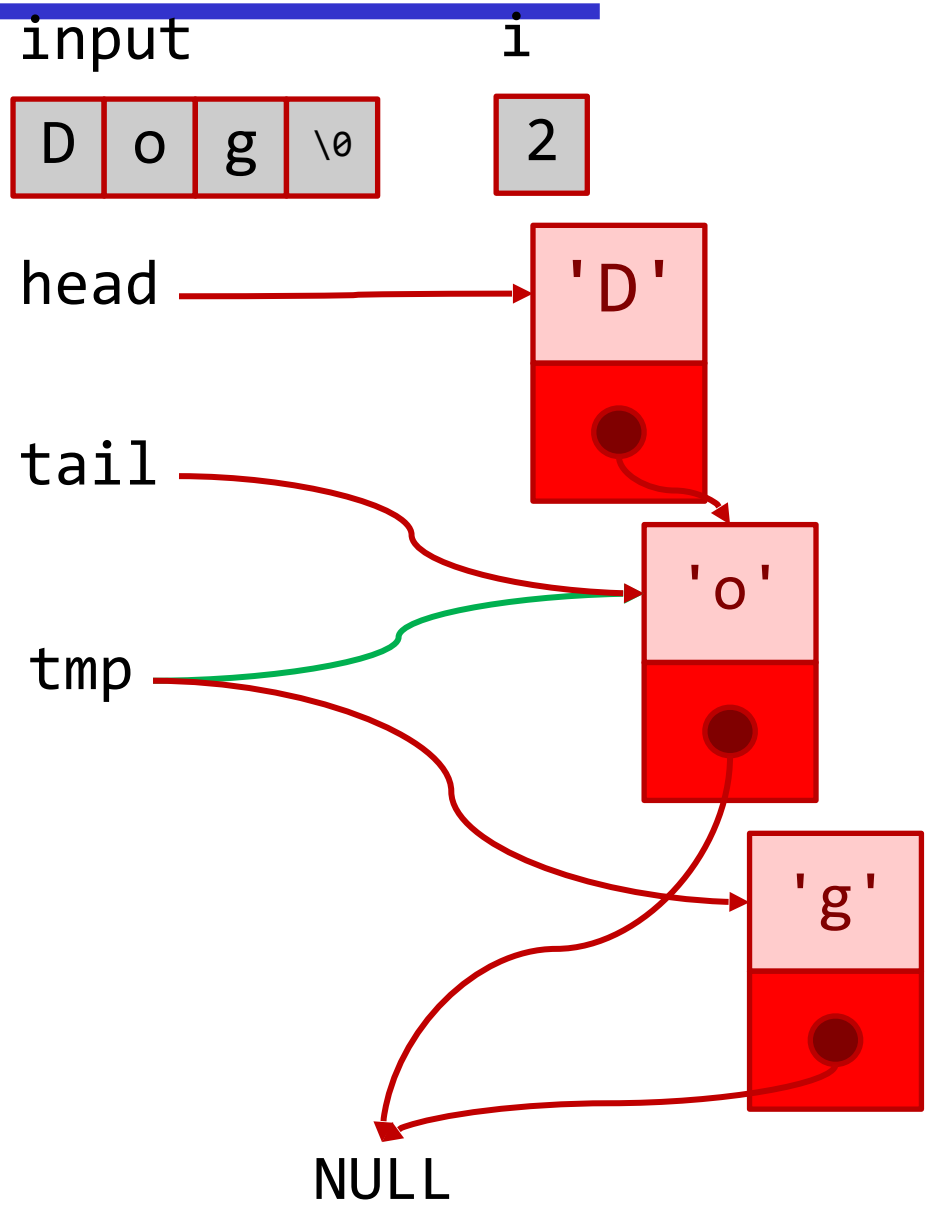
```
while(input[i] != '\0') {
    tmp = (Node *)malloc(sizeof(Node));
    tmp->data = input[i];
    tmp->next = NULL;
    if(head == NULL) {
        head = tmp;
        tail = head;
    } else {
        tail->next = tmp;
        tail = tmp;
    }
    i++;
}
```



Walkthrough

- Suppose user input is Dog

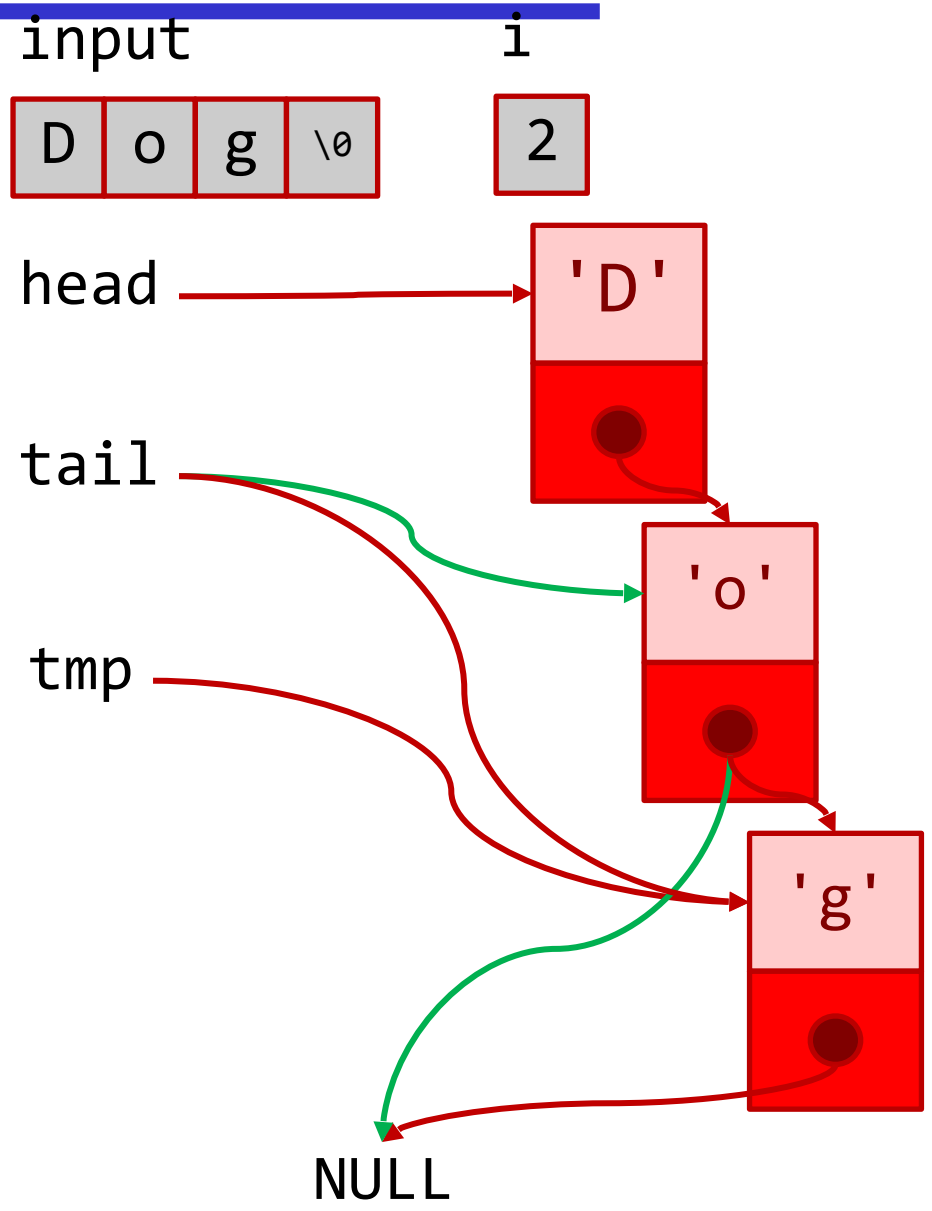
```
while(input[i] != '\0') {
    tmp = (Node *)malloc(sizeof(Node));
    tmp->data = input[i];
    tmp->next = NULL;
    if(head == NULL) {
        head = tmp;
        tail = head;
    } else {
        tail->next = tmp;
        tail = tmp;
    }
    i++;
}
```



Walkthrough

- Suppose user input is Dog

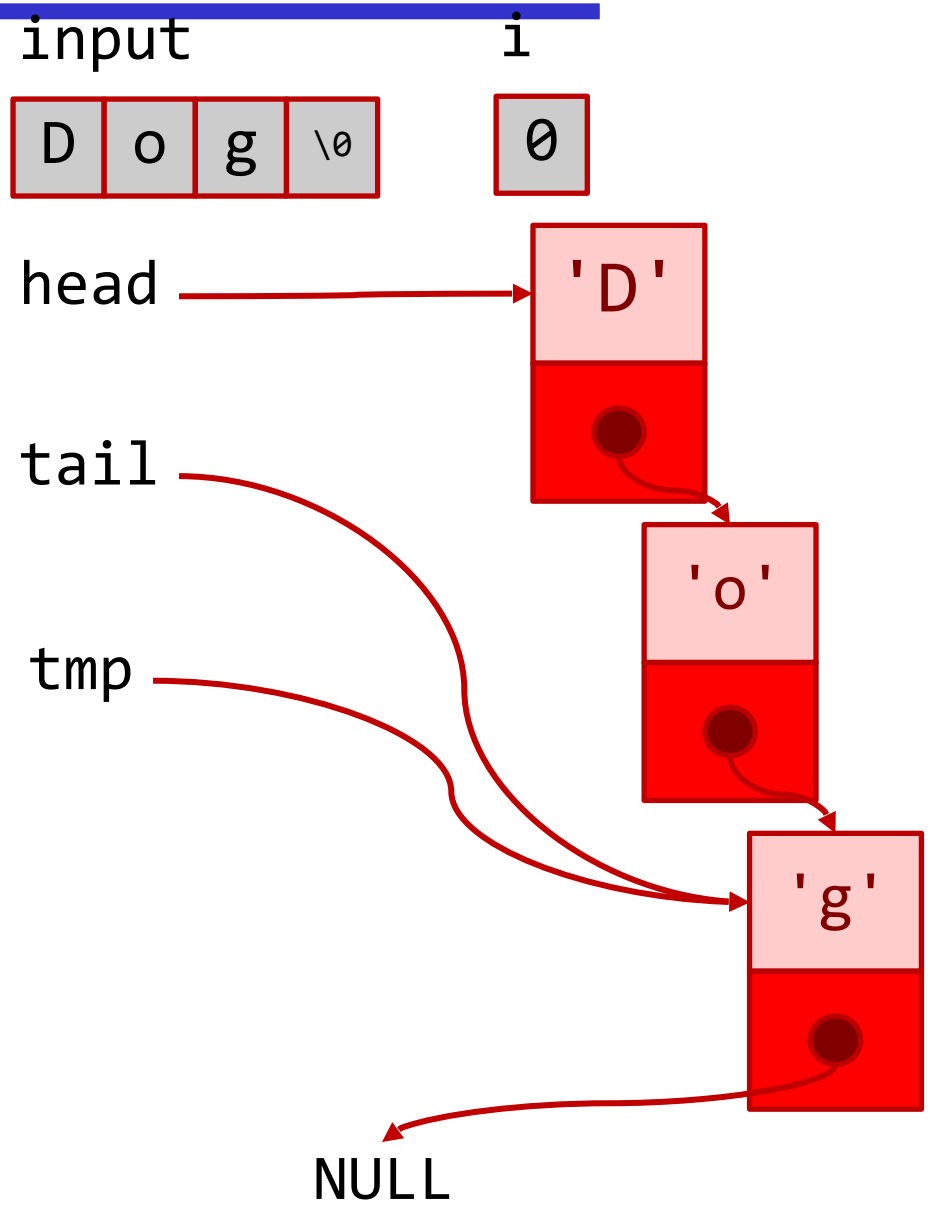
```
while(input[i] != '\0') {
    tmp = (Node *)malloc(sizeof(Node));
    tmp->data = input[i];
    tmp->next = NULL;
    if(head == NULL) {
        head = tmp;
        tail = head;
    } else {
        tail->next = tmp;
        tail = tmp;
    }
    i++;
}
```



Walkthrough

- Suppose user input is Dog

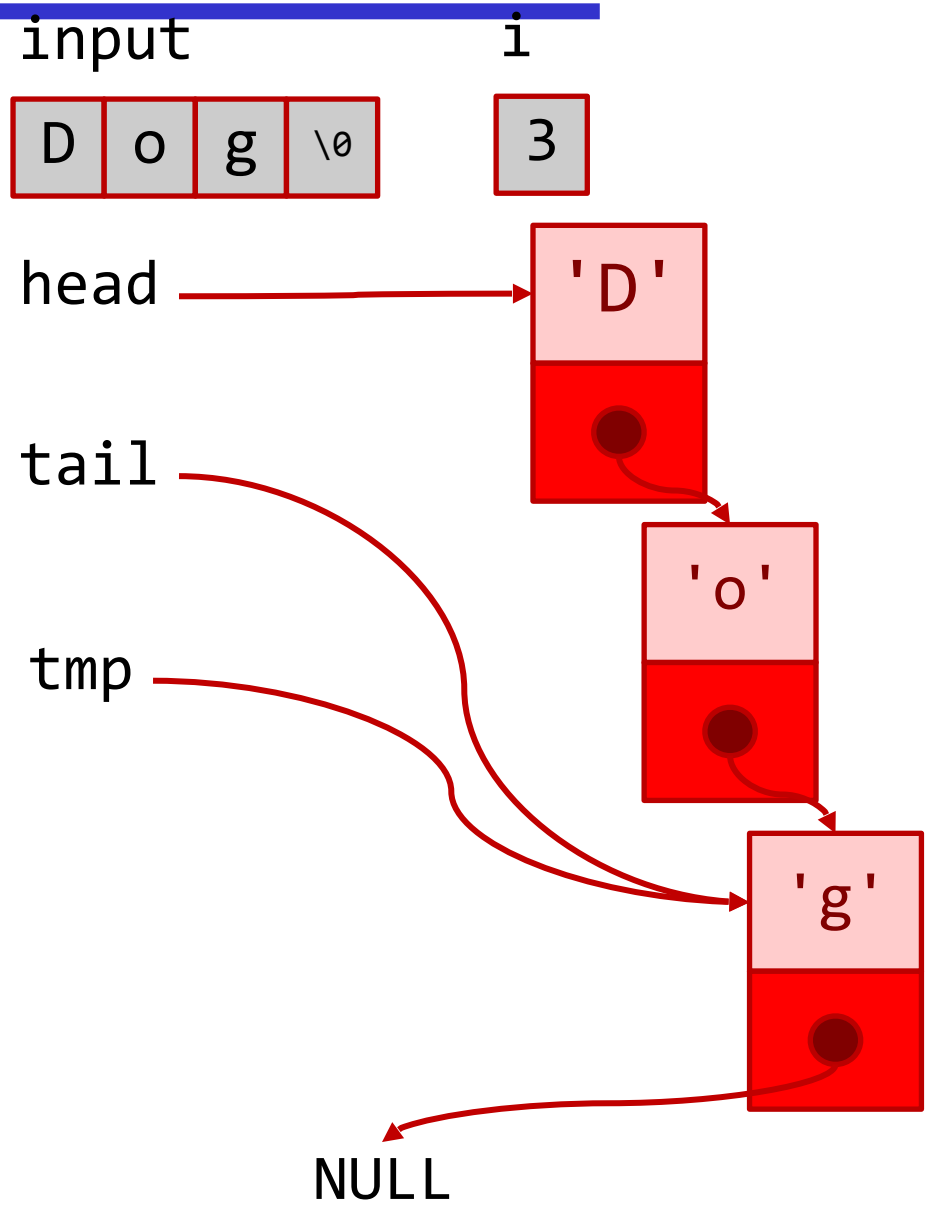
```
while(input[i] != '\0') {
    tmp = (Node *)malloc(sizeof(Node));
    tmp->data = input[i];
    tmp->next = NULL;
    if(head == NULL) {
        head = tmp;
        tail = head;
    } else {
        tail->next = tmp;
        tail = tmp;
    }
    i++;
}
```



Walkthrough

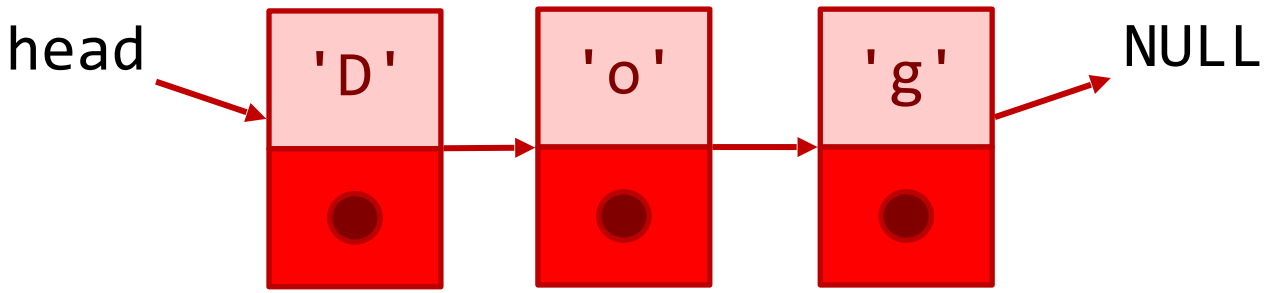
- Suppose user input is Dog

```
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++;
}
```



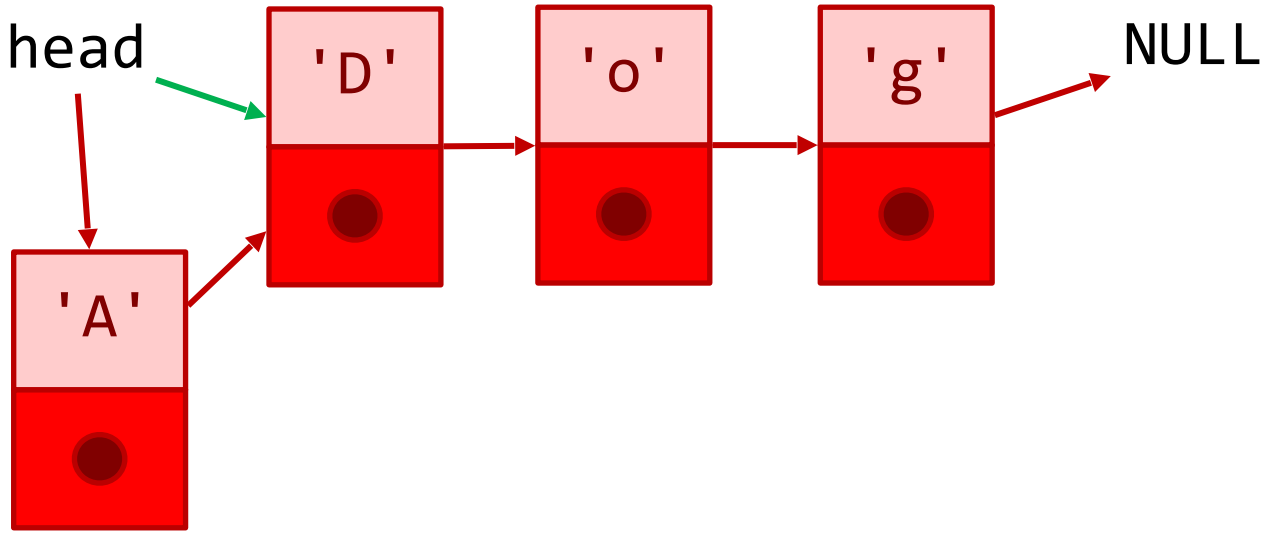
Inserting a Node (At Head)

- Easy if insertion is before first node



Inserting a Node (At Head)

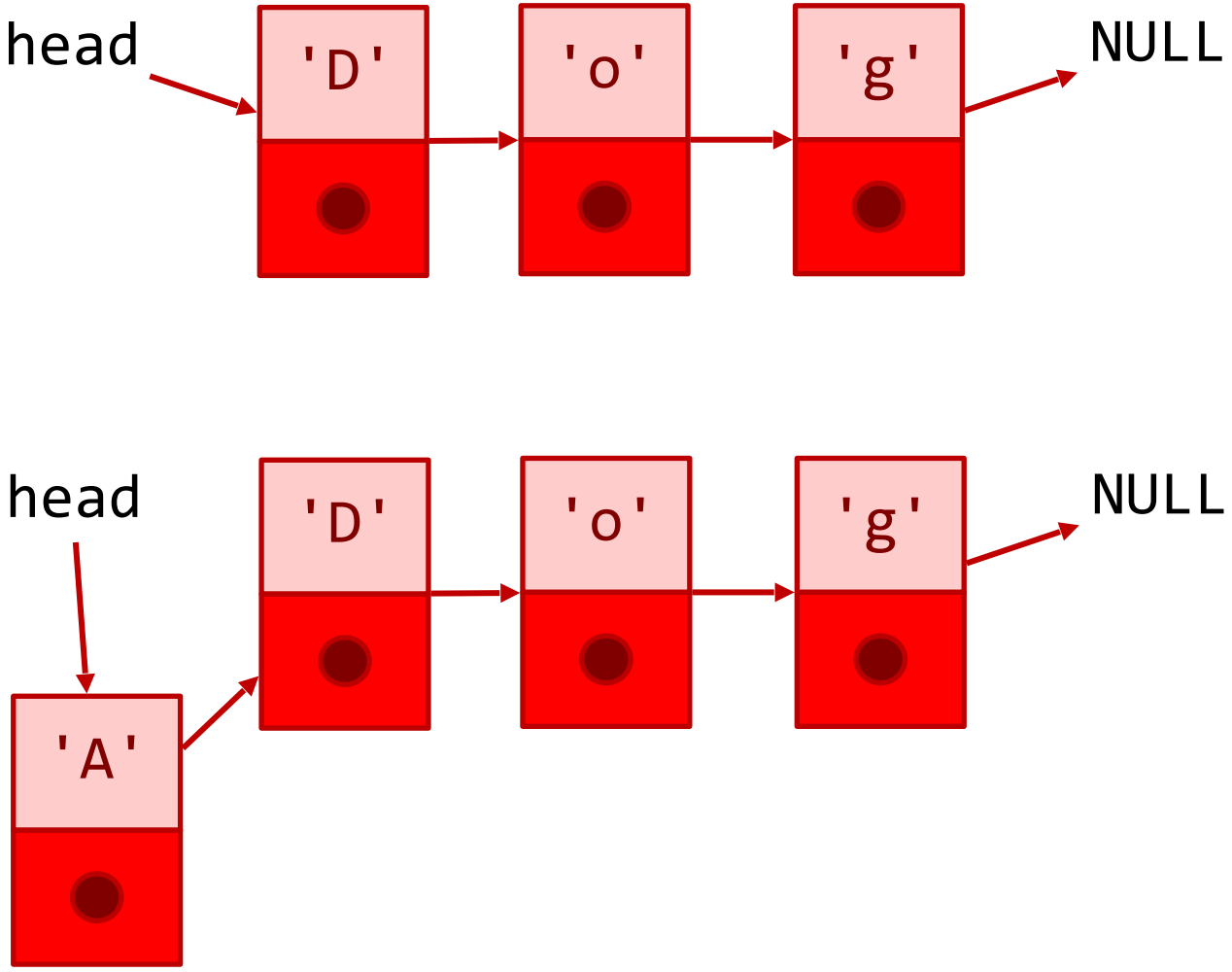
- Easy if insertion is before first node




```
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

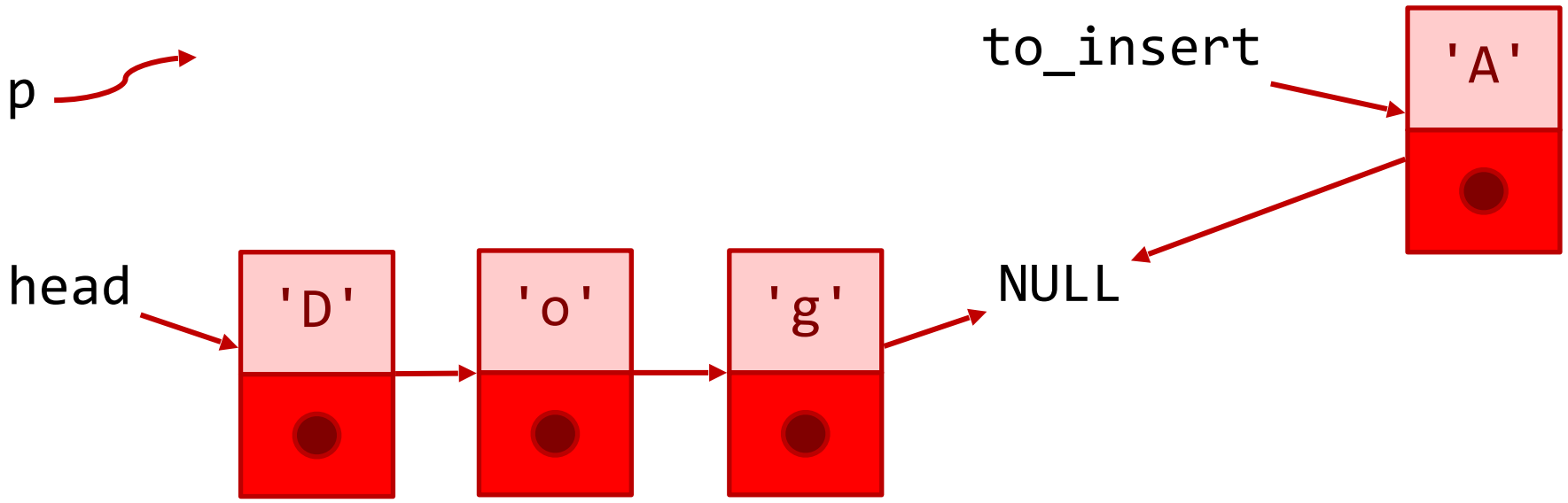



```
Node *to_insert;

/* Allocate space for
   to_insert and
   initialize */
…

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

Inserting a Node (Not at Head)

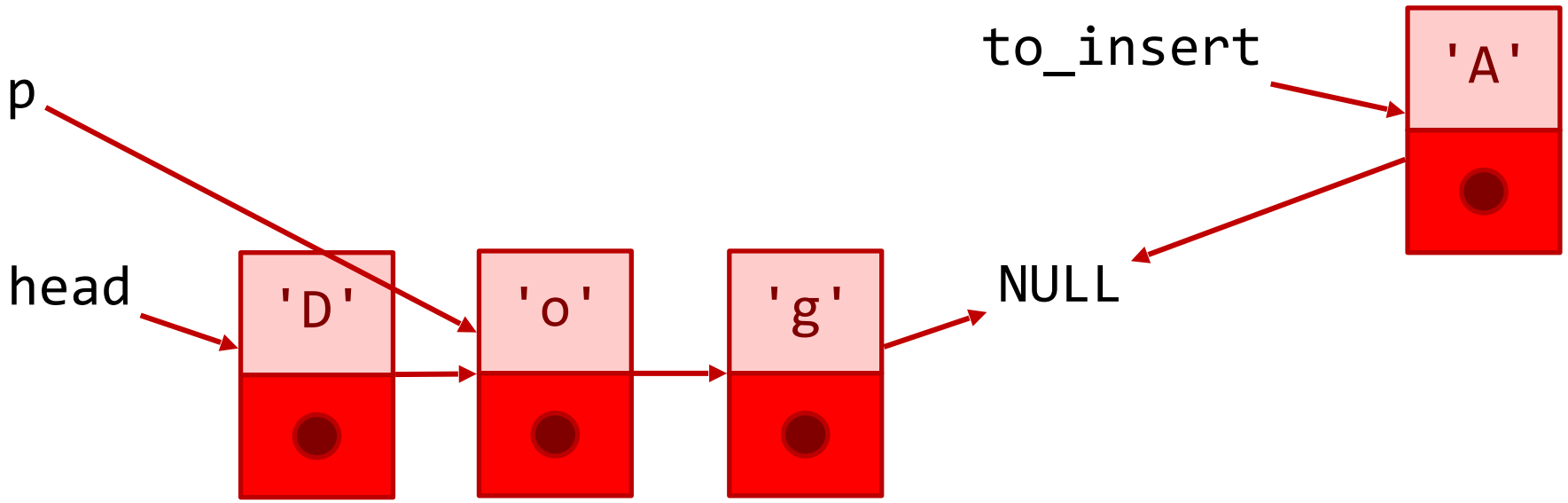
- Need to traverse list until insertion point

- Illustration: Insert `'A'` in between `'o'` and `'g'`.



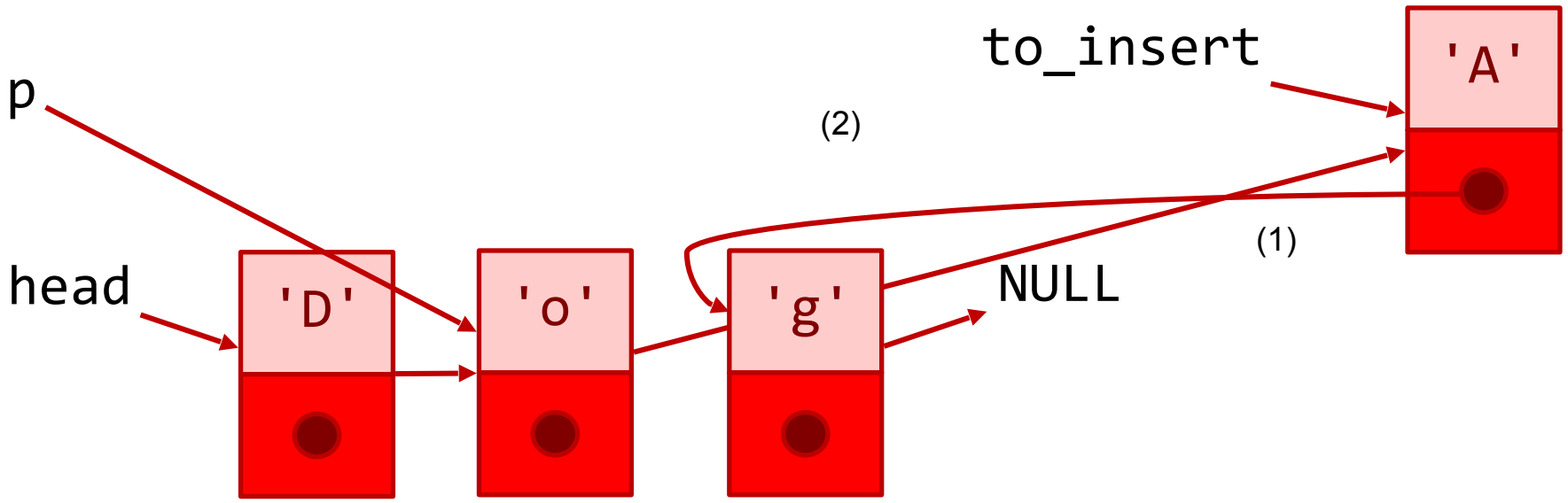
Inserting a Node (Not at Head)

- 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

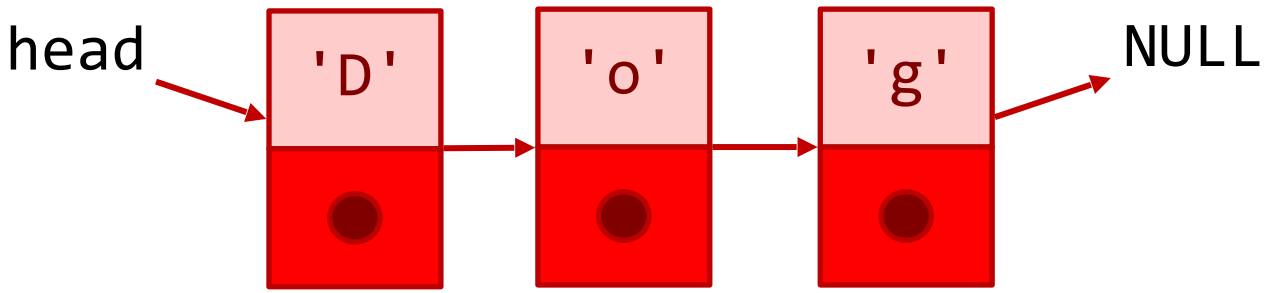


Inserting a Node (Not at Head)

```
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;
```

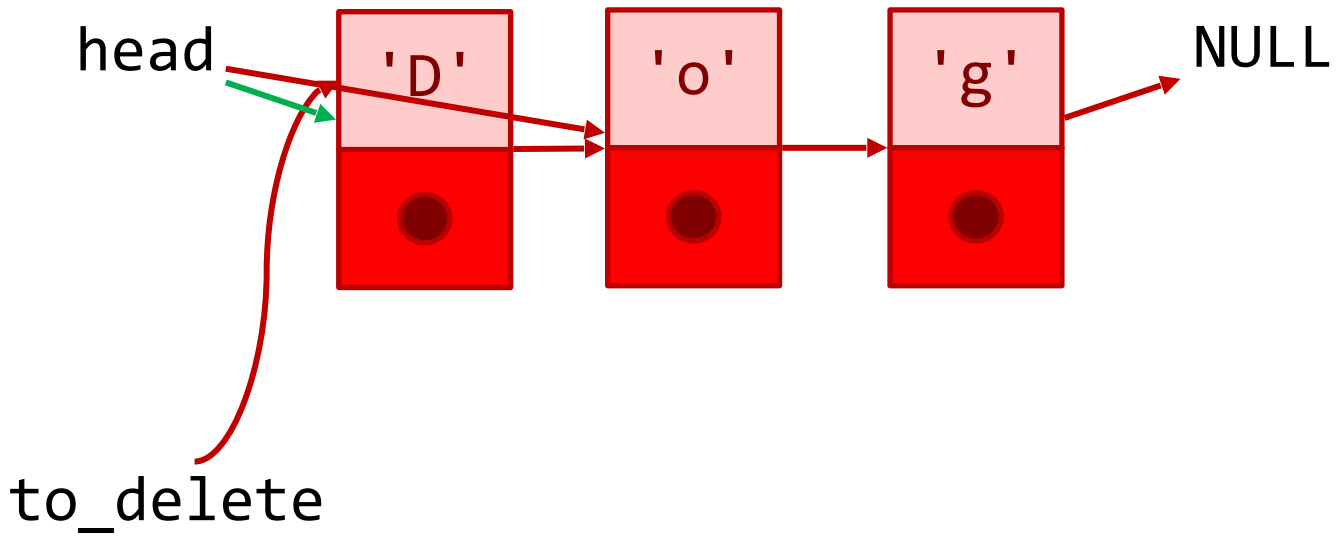
Deleting a Node (At Head)

- Easy if node to delete is first node



Deleting a Node (At Head)

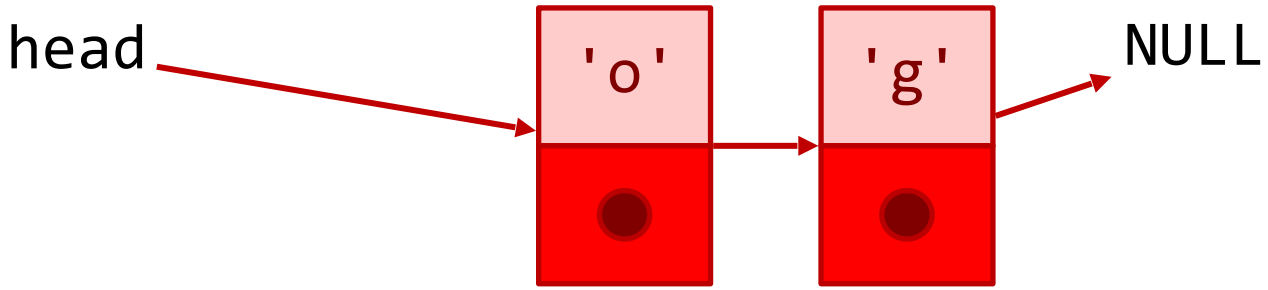
- Easy if node to delete is first node




```
Node *to_delete;

to_delete = head;
head = to_delete->next;

free(to_delete);
```

Deleting a Node (At Head)

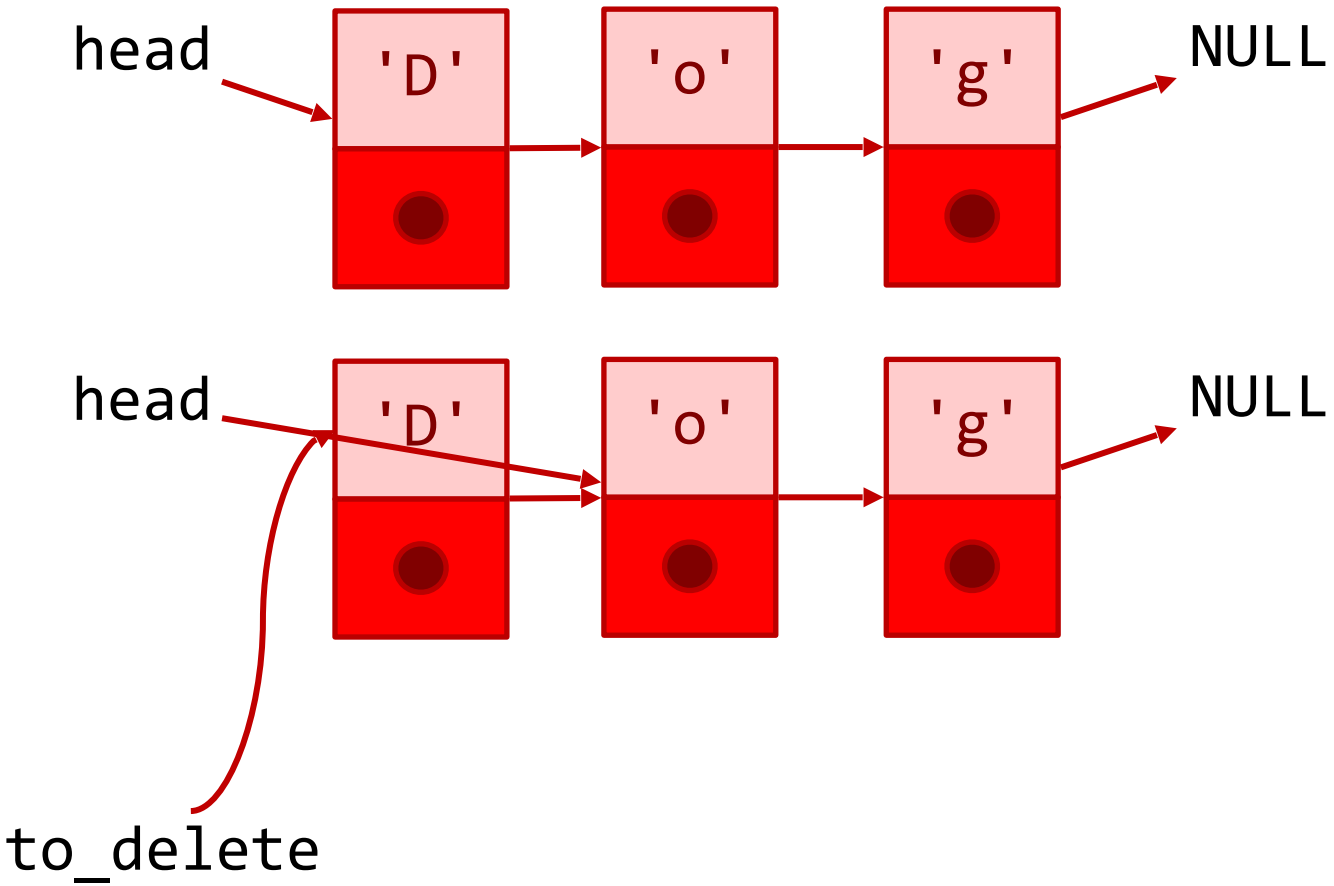
- Easy if node to delete is first node




```
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

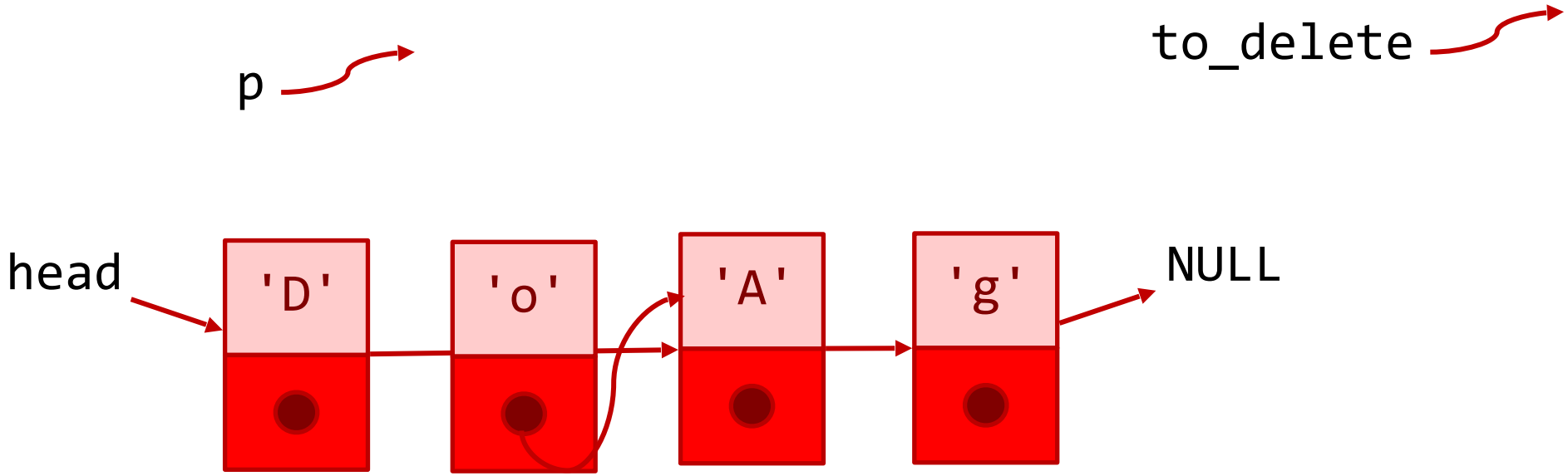



```
Node *to_delete;

to_delete = head;
head = to_delete->next;

free(to_delete);
```

Deleting a Node (Not at Head)

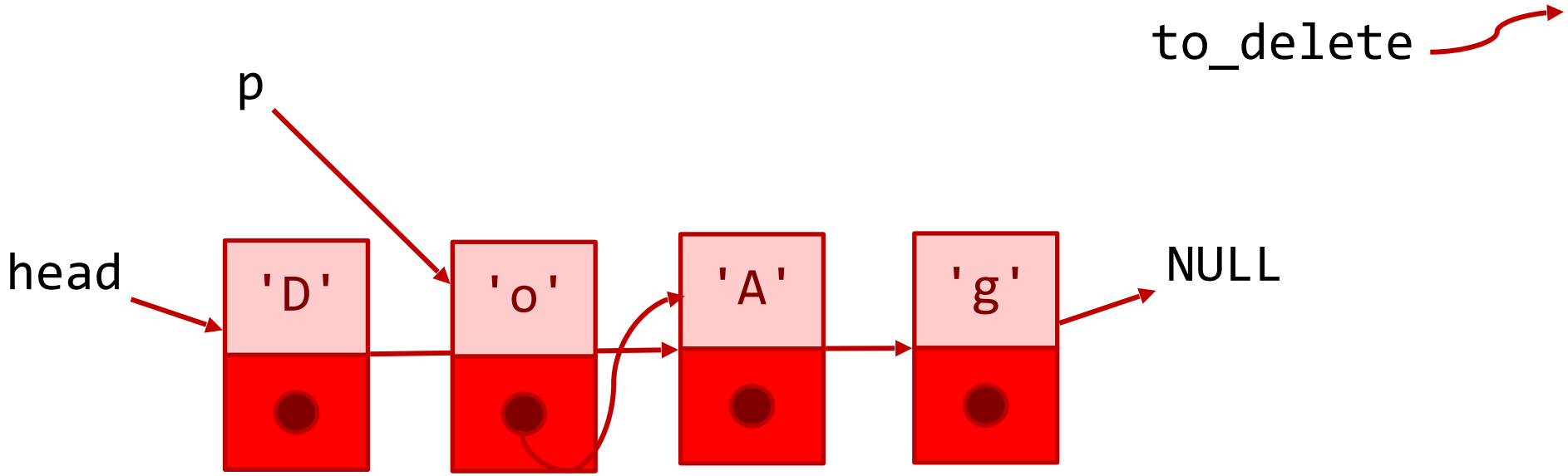
- Need to traverse list until just before the node to be deleted

- Illustration: Delete `'A'` in between `'o'` and `'g'`.



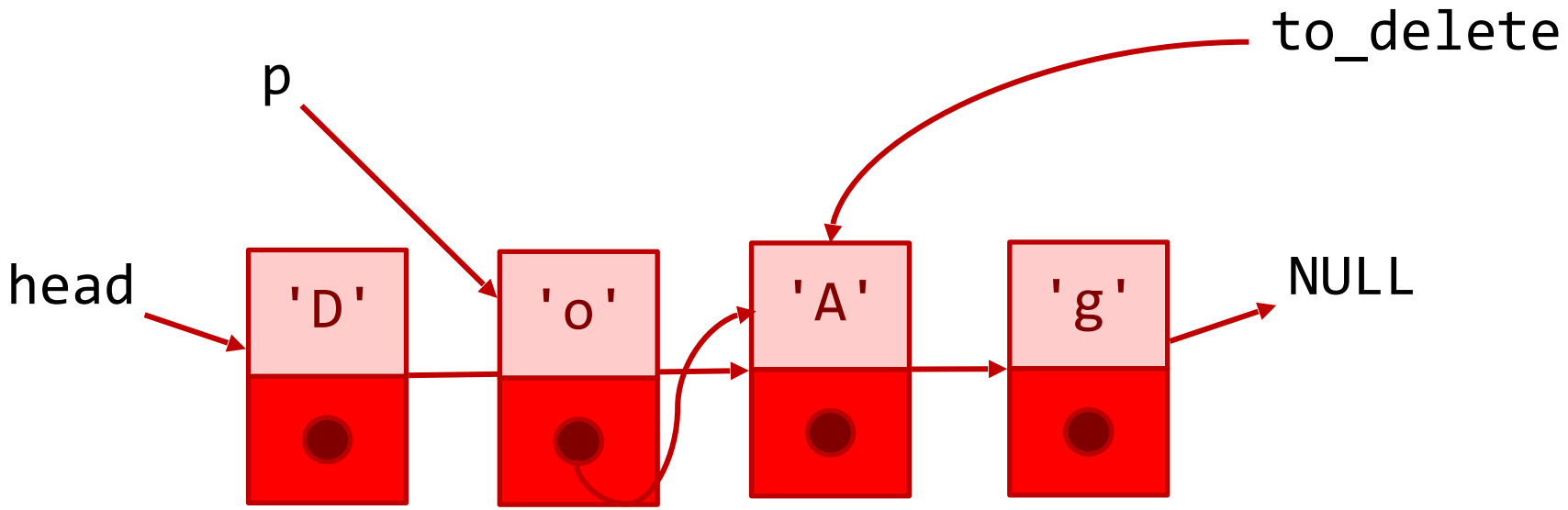
Deleting a Node (Not at Head)

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



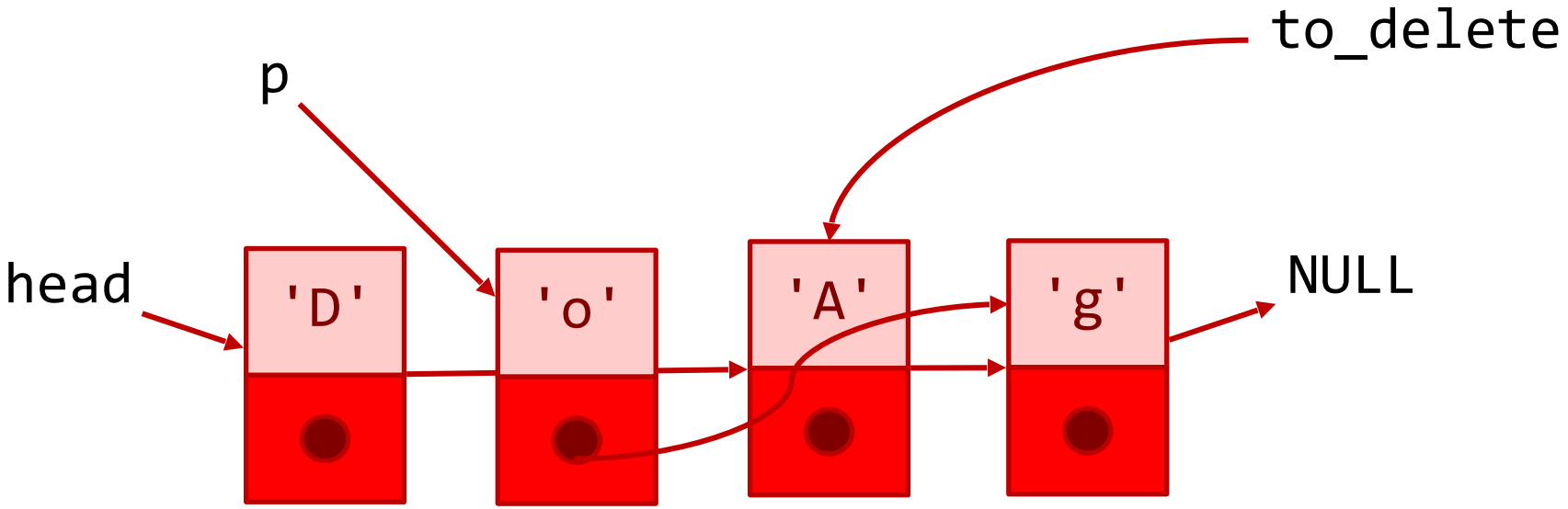
Deleting a Node (Not at Head)

- 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)



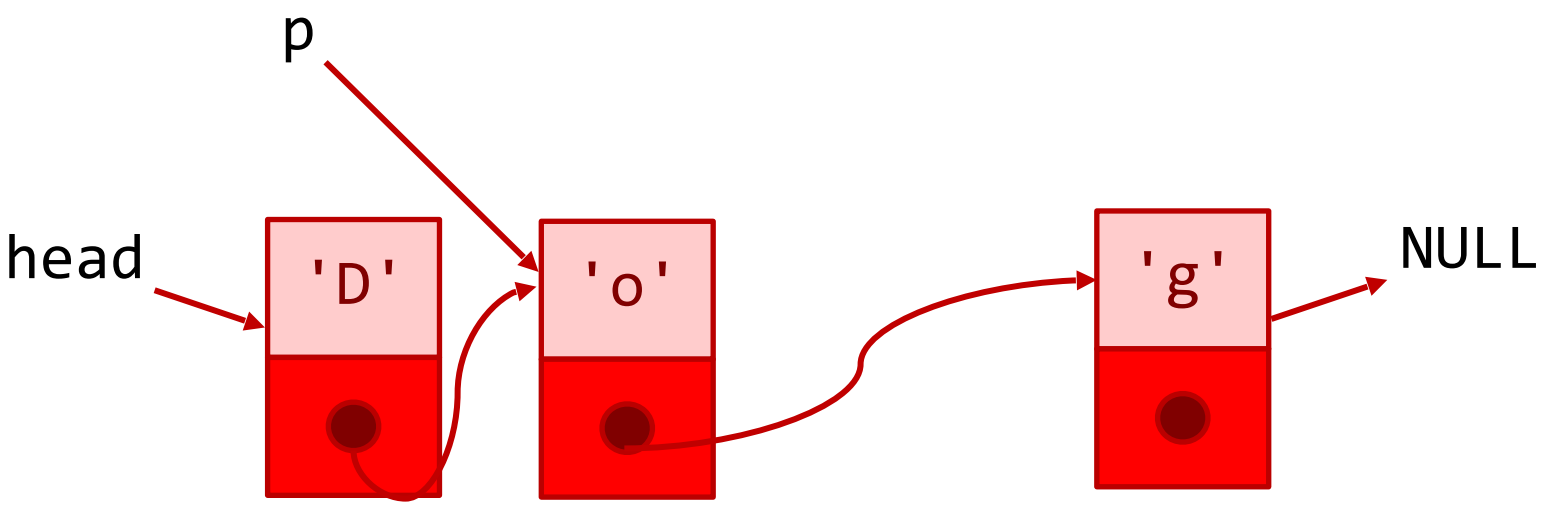
Deleting a Node (Not at Head)

- 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)?

Things to Consider

- Which of these need two pointers in every node: list, doubly-linked list, binary
- What is `a.next->data` here? Would `a.next.data` also work? Why or why not?

```
Node b = {'y', NULL};
Node a = {'x', &b};
```

Things to Consider

- What does a traversal starting from `head` print?

```
Node c = {'c', NULL};
Node b = {'b', &c};
Node a = {'a', &b};
Node *head = &b;
```

- In a list declared as `node4`, `node3`, `node2`, `node1`, why does `node4` come first? Give one limitation of building a list this way.

- Using the code above, what are `head->next->data` and `head->next->next`?

Things to Consider

- For the list `list`, what does this print? What goes wrong for the list `dog`?

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

- Write `int list_length(Node *head)`, which returns the number of nodes.

Things to Consider

```
int list_length(Node *head)
{
    int n = 0;
    Node *p = head;

    for( ; p != NULL; p = p->next)
        n++;
    return n;
}
```

Things to Consider

- For the input `Hi`, how many times is `malloc` called? What do `head` and `tail` point to at the end?

```
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++;
}
```

- The `else` branch is changed to `else { tail->next = tmp; }`, with no `tail = tmp;`. For the input `Dog`, what does a traversal print? What happened to the missing node?

- Why does every new node need `tmp->next = NULL`?

Things to Consider

- `to_insert` is a new node holding `'X'`. Starting with the list `Dog`, what does a traversal print after these lines?

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

- This search has been changed from the one on the slides. If the list is `Dig`, what does the list look like after the insert, with `to_insert` holding `'A'`? What happens on an empty list?

```
p = head;
while(p->next != NULL && p->data != 'o')
    p = p->next;
to_insert->next = p->next;
p->next = to_insert;
```

Things to Consider

- Write `void delete_second(Node *head)`, which deletes the second node, if there is one.
- The list is `aba`. What does the list look like after this code?

```
p = head;
while(p->next != NULL && p->next->data != 'a')
    p = p->next;
if(p->next != NULL) {
    to_delete = p->next;
    p->next = to_delete->next;
    free(to_delete);
}
```

- `p` is on the `'o'` in the list `DoAg`. This is meant to delete `'A'`. What does it actually do?

```
p->next = p->next->next;
free(p->next);
```

Things to Consider

```
void delete_second(Node *head)
{
    Node *to_delete;

    if(head == NULL || head->next == NULL)
        return;
    to_delete = head->next;
    head->next = to_delete->next;
    free(to_delete);
}
```