Week 5 Lecture 2

NWEN 241
Systems Programming

Jyoti Sahni

Jyoti.sahni@ecs.vuw.ac.nz

Content

- **Dynamic memory allocation**
- **Introduction to Linked Lists**

Recap: Dynamic Memory Management Functions

- `calloc` - allocate *array* of memory
- `malloc` - allocate *a single block* of memory
- `realloc` – extend or reduce the amount of space allocated previously
- `free` - free up a piece of memory that is no longer needed

Memory allocated dynamically does not go away at the end of functions, you **MUST** explicitly **free** it up

`calloc` – Allocate Memory for Array

- Function prototype:

```
void *calloc(size_t num, size_t esize)
```

  - `size_t` – special type used to indicate sizes, unsigned int
  - **`num`** – number of elements to be allocated in the array
  - **`esize`** – size (in bytes) of a single element to be allocated
    - to get the correct value, use **`sizeof(<type>)`**
    - memory of size **`num*esize`** is allocated
- `calloc` returns the address of the 1st byte of this memory
  - Cast the returned address to the appropriate type
- If not enough memory is available, `calloc` returns `NULL`

`free` – Return Memory to Heap

- Function prototype:

```
void free(void *ptr)
```

  - Memory at location pointed by `ptr` is released (so that it could be used again)
  - Program keeps track of each piece of memory allocated from where that memory starts
  - If we free a piece of memory allocated with `calloc`, the entire array is freed (released)
  - **Undefined behaviour** if we pass as address to free an address of something that was not allocated dynamically (or has already been freed)

free Example

```
float *nums;
int a_size;

printf("Read how many numbers:");
scanf("%d",&a_size);
nums = (float *) calloc(a_size, sizeof(float));

/* Use array nums */
…

/* When done with nums: */
free(nums);

/* Would be an error to do it again - free(nums) */
```

free Example

```
float *nums;
…
nums = (float *) calloc(a_size, sizeof(float));
…
free(nums);
```

`malloc` – Allocate Memory

- Function prototype:

```
void *malloc(size_t esize)
```

  - Similar to `calloc`, except we use it to allocate a single block of the given size `esize`
  - `NULL` returned if not enough memory available
  - Memory must be released using free if no longer needed
  - Following are equivalent:

```
malloc(a_size*sizeof(float))
```

```
calloc(a_size, sizeof(float))
```

malloc Example

```
float *nums;
int a_size;
int idx;

printf("Read how many numbers:");
scanf("%d",&a_size);
nums = (float *) malloc(a_size * sizeof(float));

if(nums == NULL) {
      /* exit or do some other stuff */
}
…
```

`realloc` – increase/decrease memory allocation

- Function prototype:

```
void *realloc(void *ptr, size_t esize)
```

- `ptr` is a pointer to a piece of memory previously dynamically allocated
- `esize` is new size to allocate
- `NULL` returned if reallocation fails
- Function performs following action:
  1) allocates memory of size `esize`,
  2) copies the contents of the memory at `ptr` to the first part of the new piece of memory, and lastly,
  3) old block of memory is freed up
  4) Address to new piece of memory is returned

realloc Example

```
float *nums;
int a_size;

nums = (float *)calloc(5, sizeof(float));
/* nums is an array of 5 floating point values */

for (a_size = 0; a_size < 5; a_size++)
  nums[a_size] = 2.0 * a_size;
/* nums[0]=0.0, nums[1]=2.0, nums[2]=4.0, etc. */

nums = (float *)realloc(nums, 10*sizeof(float));
/* An array of 10 floating point values is allocated, the
first 5 floats from the old nums are copied as the first 5
floats of the new nums, then the old nums is released */
```

realloc Example

```
float *nums;
...
nums = (float *)calloc(5, sizeof(float));
…
nums = (float *)realloc(nums, 10*sizeof(float));
```

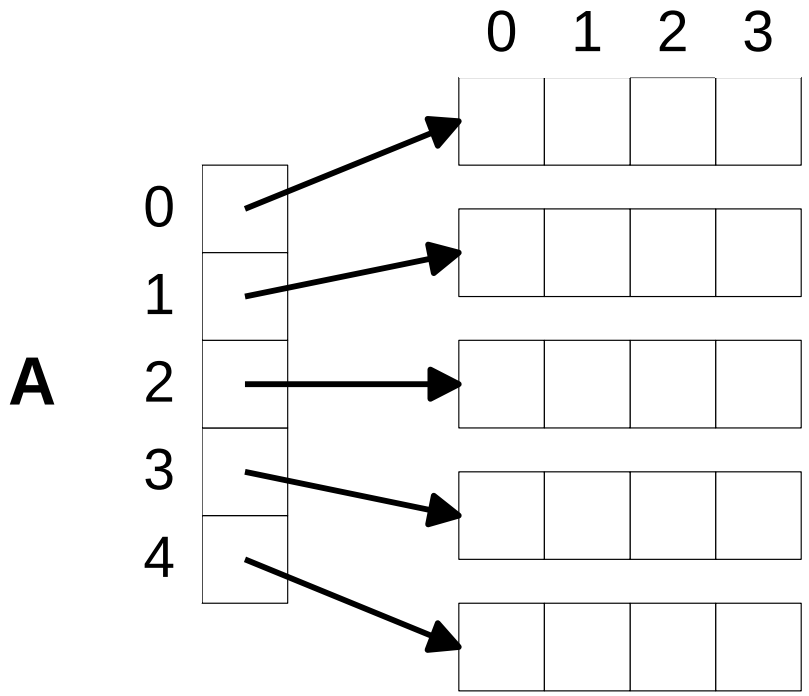
realloc Example: Will this work?

```
float *nums;
…
nums = (float *)calloc(5, sizeof(float));
…
realloc(nums, 10*sizeof(float));
```

Allocating Memory for 2D array

- Can not simply allocate 2D (or higher) array dynamically

- Solution:
  1) Allocate an array of pointers (1st dimension),
  2) Make each pointer point to a 1D array of the appropriate size

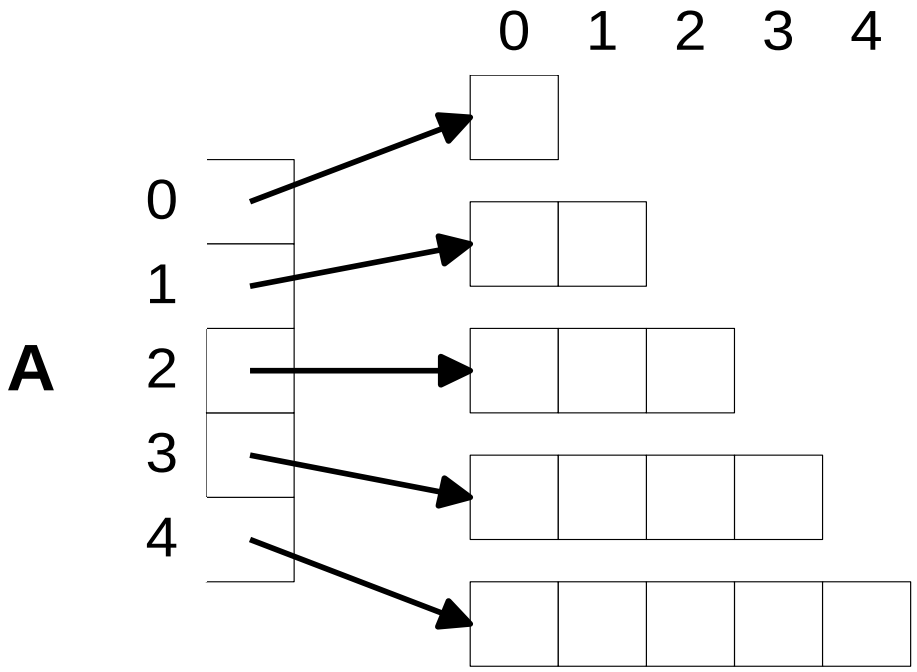


Allocating Memory for 2D array

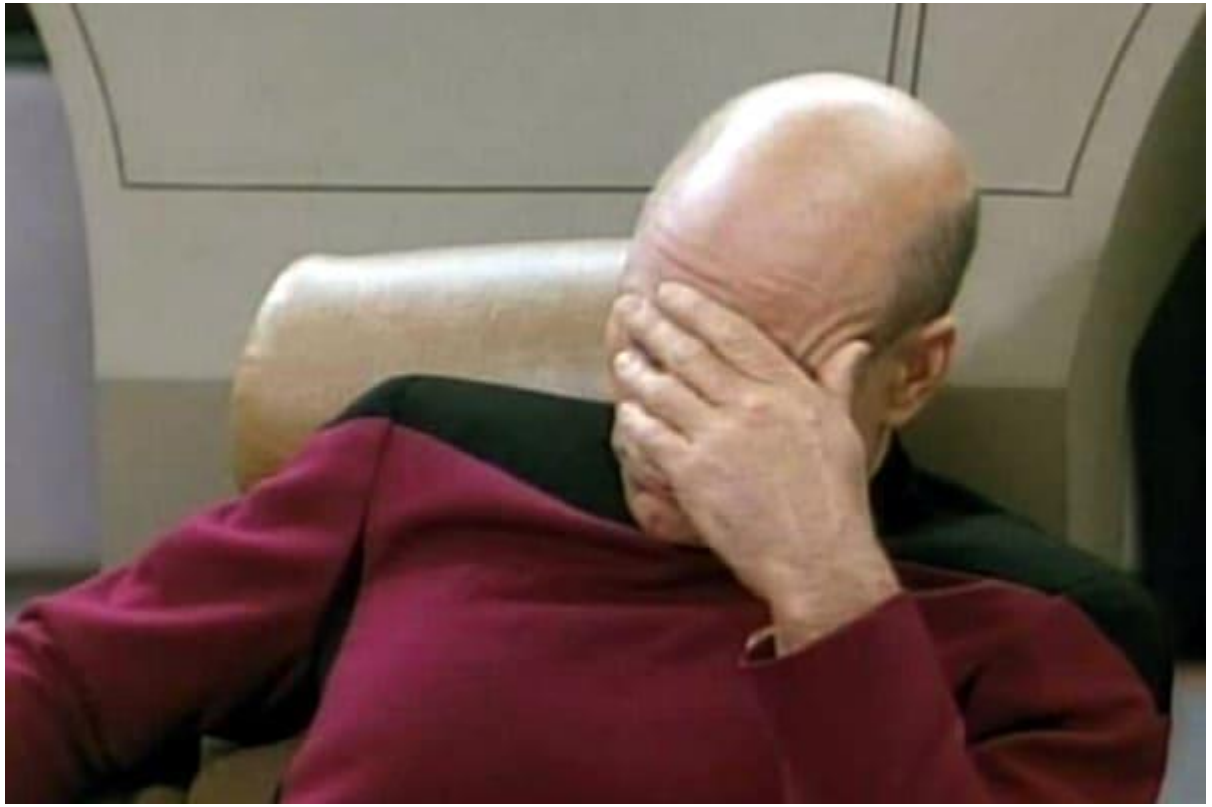
```
float **A;  /* A is an array (pointer) of float pointers */
int X;

A = (float **) calloc(5, sizeof(float *));
/* A is a 1D array (size 5) of float pointers */

for (X = 0; X < 5; X++)
  A[X] = (float *) calloc(4, sizeof(float));
  /* Each element of array points to an array of 4 float
     variables */

/* A[X][Y] is the Yth entry in the array that the Xth member of A
points to */
```

Irregular-sized 2D array

```
float **A;
int X;

A = (float **)calloc(5,
sizeof(float *));

for (X = 0; X < 5; X++)
  A[X] = (float *)
         calloc(X+1,
         sizeof(float));
```



Common Issues With Dynamic Memory



Issue #1

- Returning a pointer to an automatic variable

```
int *foo(void)
{
      int x;

      …

      return &x;
      /* x does not exist outside the function */
      /* Returning its address will result in unknown behaviour */
}
```

Issue #2

- Heap block overrun: similar to array going out of bounds

```
void foo(void)
{
      int *x = (int *) malloc(10 * sizeof(int));

      x[10] = 10;
      /* Allocated memory is only up to x[9] */

      …

      free(x);
}
```

Issue #3

- Memory leak: loss of pointer to allocated memory

```
int *pi;

void foo(void)
{
        pi = (int*) malloc(8*sizeof(int));
        /* Leaked the old memory pointed to by pi */
        …
        free(pi); /* foo() is done with pi, so free it */
}

int main(void)
{
        pi = (int*) malloc(4*sizeof(int));
        foo();
}
```

Issue #4

- Potential memory leak
  - Loss of pointer to beginning of memory block
  - May still recover through pointer arithmetic

```
int *ip = NULL;

void foo(void)
{
        ip = (int *) malloc(2 * sizeof(int));
        …
        ip++;
        /* ip is not pointing to the start of the block anymore */
}
```

Issue #5

- Freeing non-heap or unallocated memory

```
void foo(void)
{
   int fnh = 0;
   free(&fnh); /* Freeing stack memory */
}

void bar(void)
{
   int *fum = (int *) malloc(4 * sizeof(int));
   free(fum+1); /* fum+1 points to middle of block */
   free(fum);
   free(fum); /* Freeing already freed memory */
}
```

Detecting Memory Leaks and Other Issues

- Valgrind is an open-source tool for detecting memory management and threading bugs
- For more information: http://valgrind.org/

Valgrind

- Valgrind is an open-source tool for detecting memory management and threading bugs
- Available in CO246 lab computers and ECS servers

- Steps to using Valgrind:
  - Compile program with –g option
    - Example: `gcc -g buggy.c –o buggy`
  - Run with valgrind:
    - Example: `valgrind --leak-check=yes ./buggy`

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

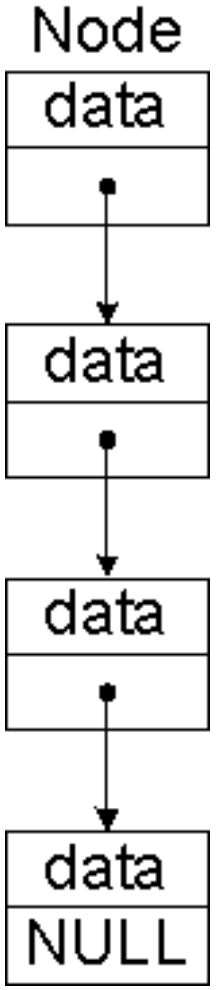
Linked Data Structures

- 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;
```

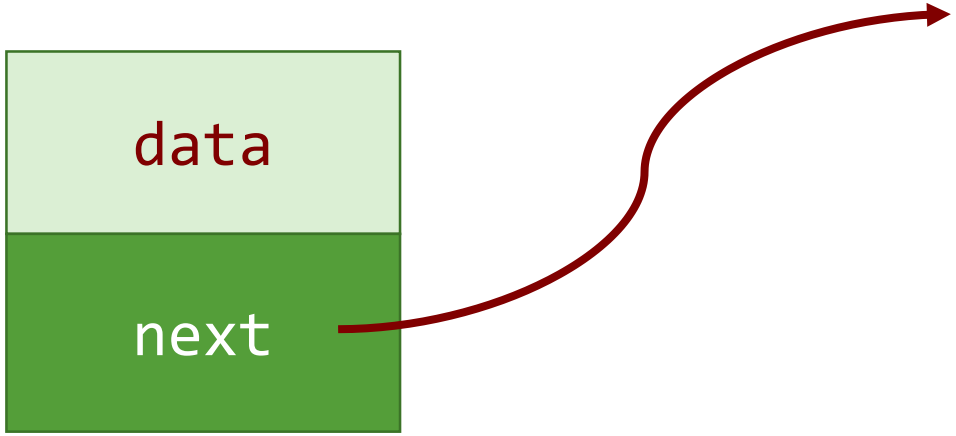
- A singly-linked list is the simplest example



Singly-Linked List (1)

- Node type definition

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



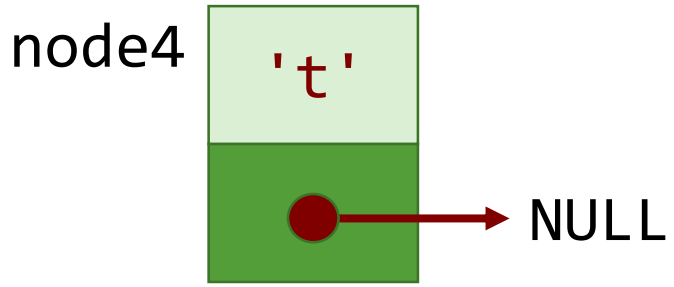
Singly-Linked List (2)

- 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;
```



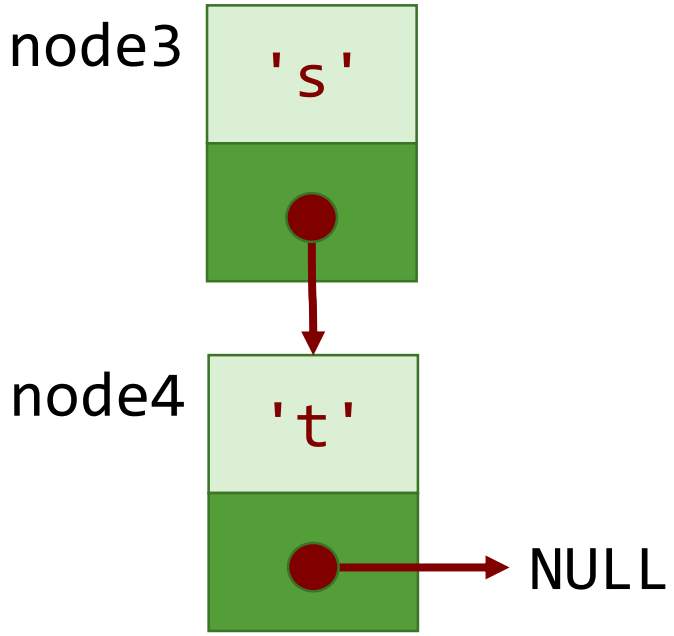
Singly-Linked List (3)

- 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;
```



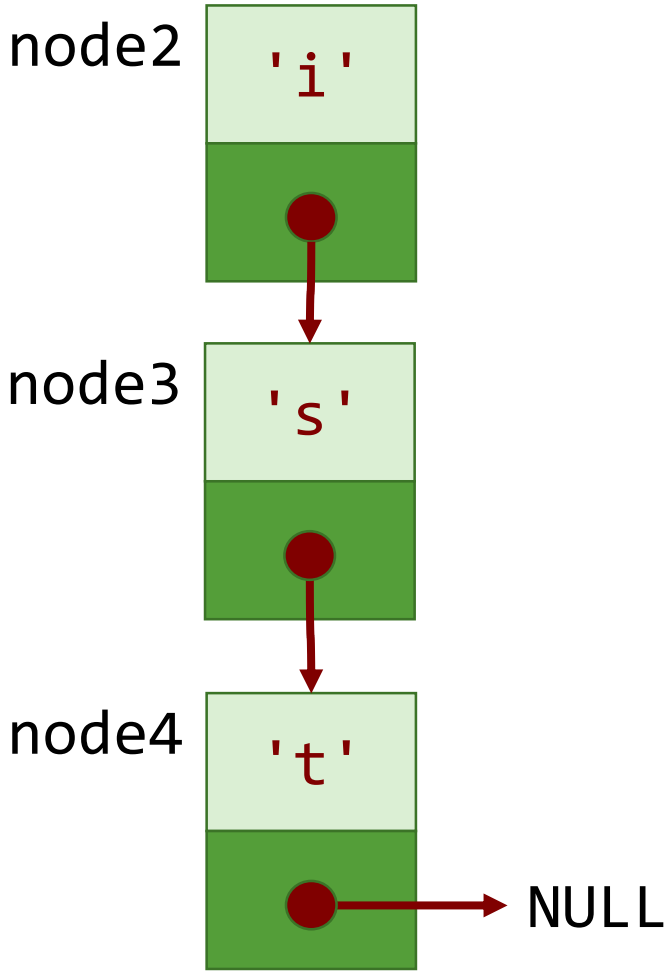
Singly-Linked List (4)

- 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;
```



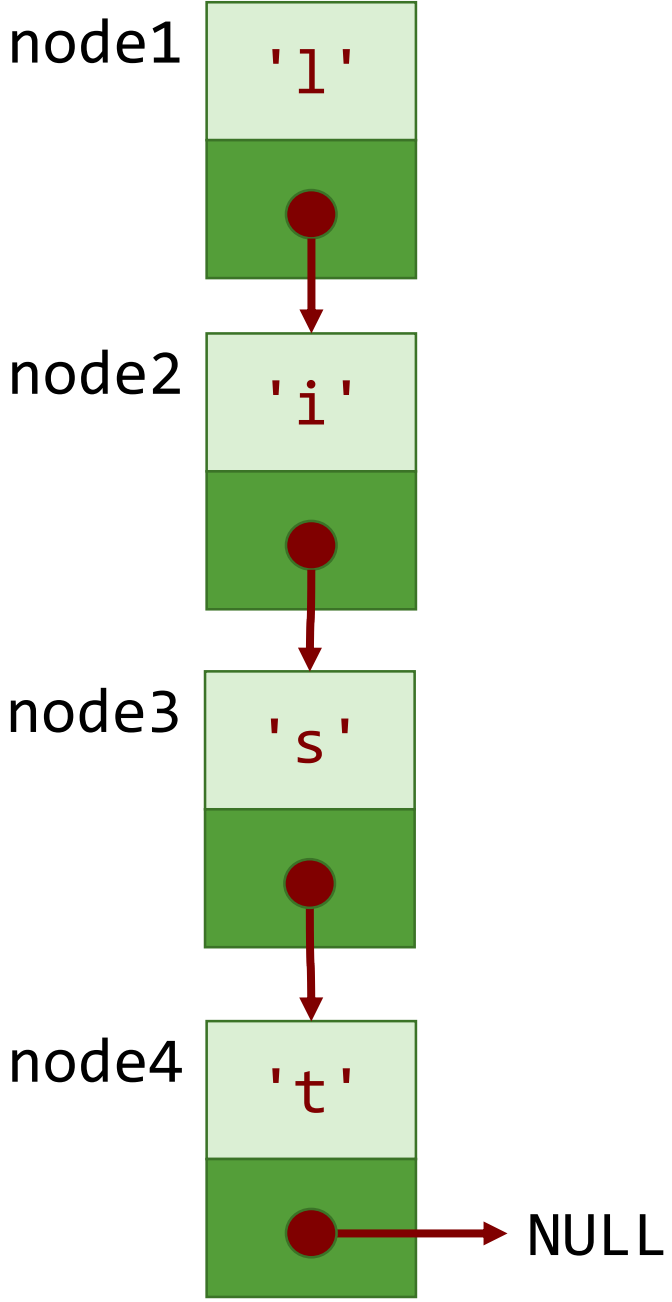
Singly-Linked List (5)

- 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;
```



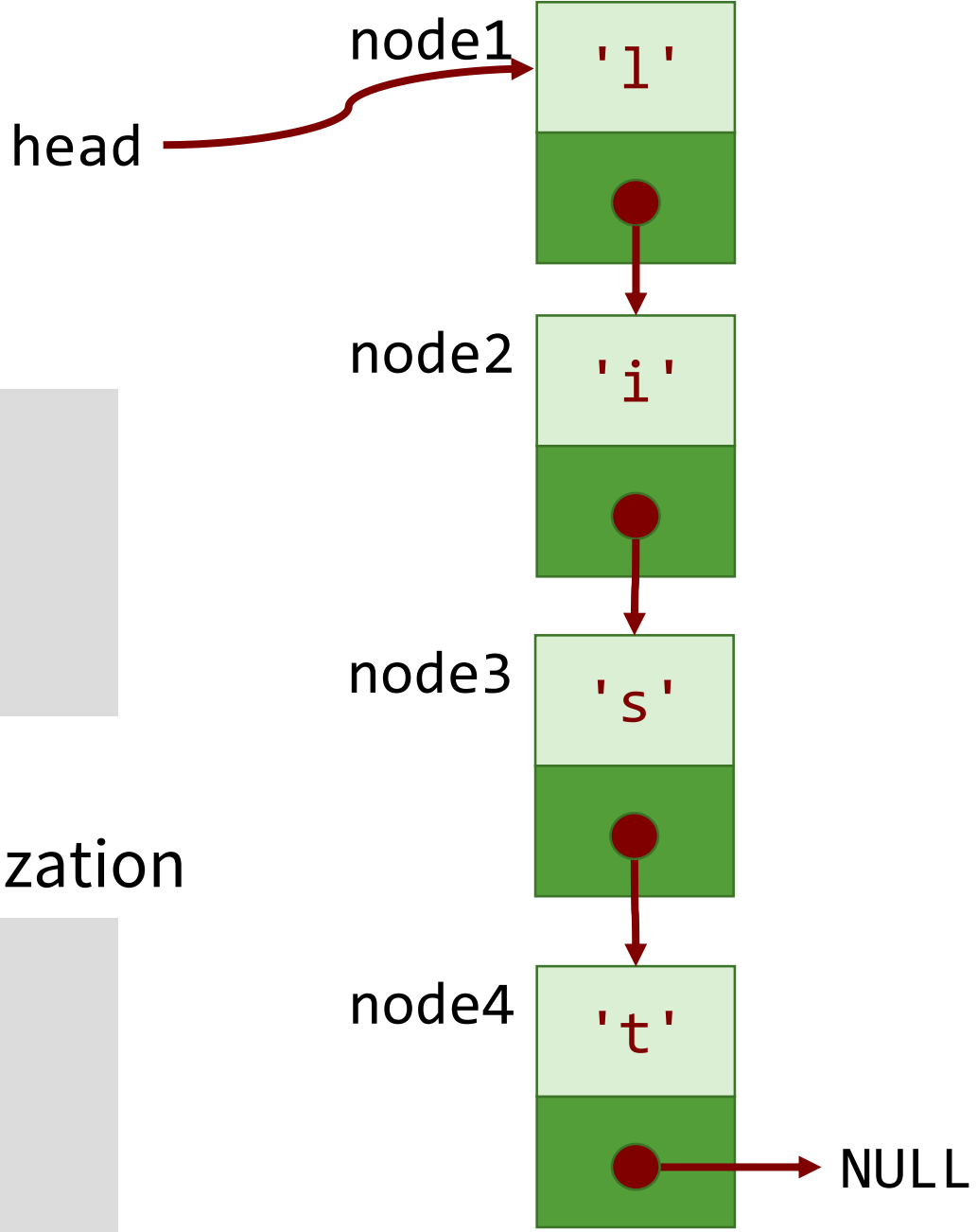
Singly-Linked List (6)

- 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);
```

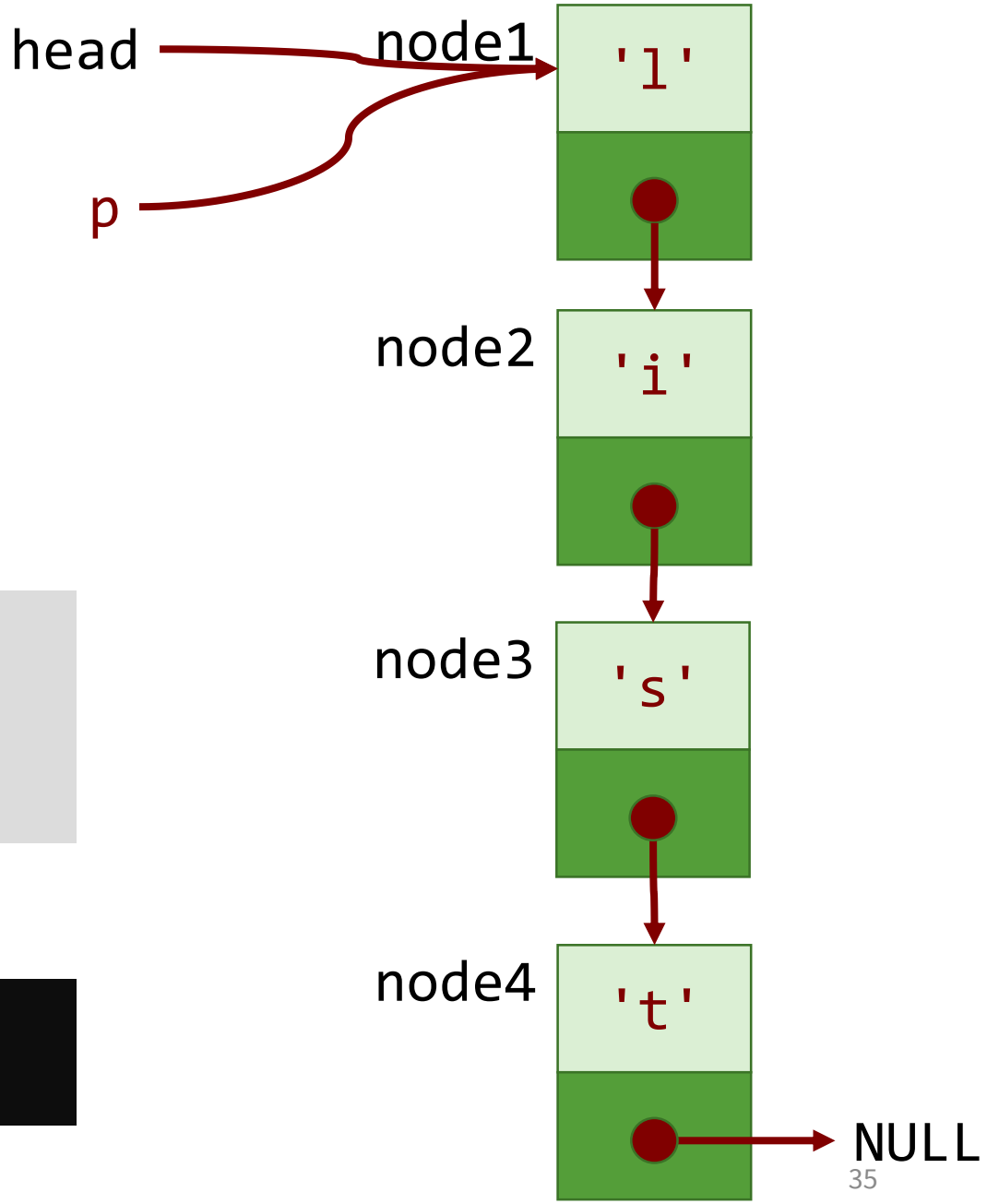
Singly-Linked List Traversal (2)

- 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:



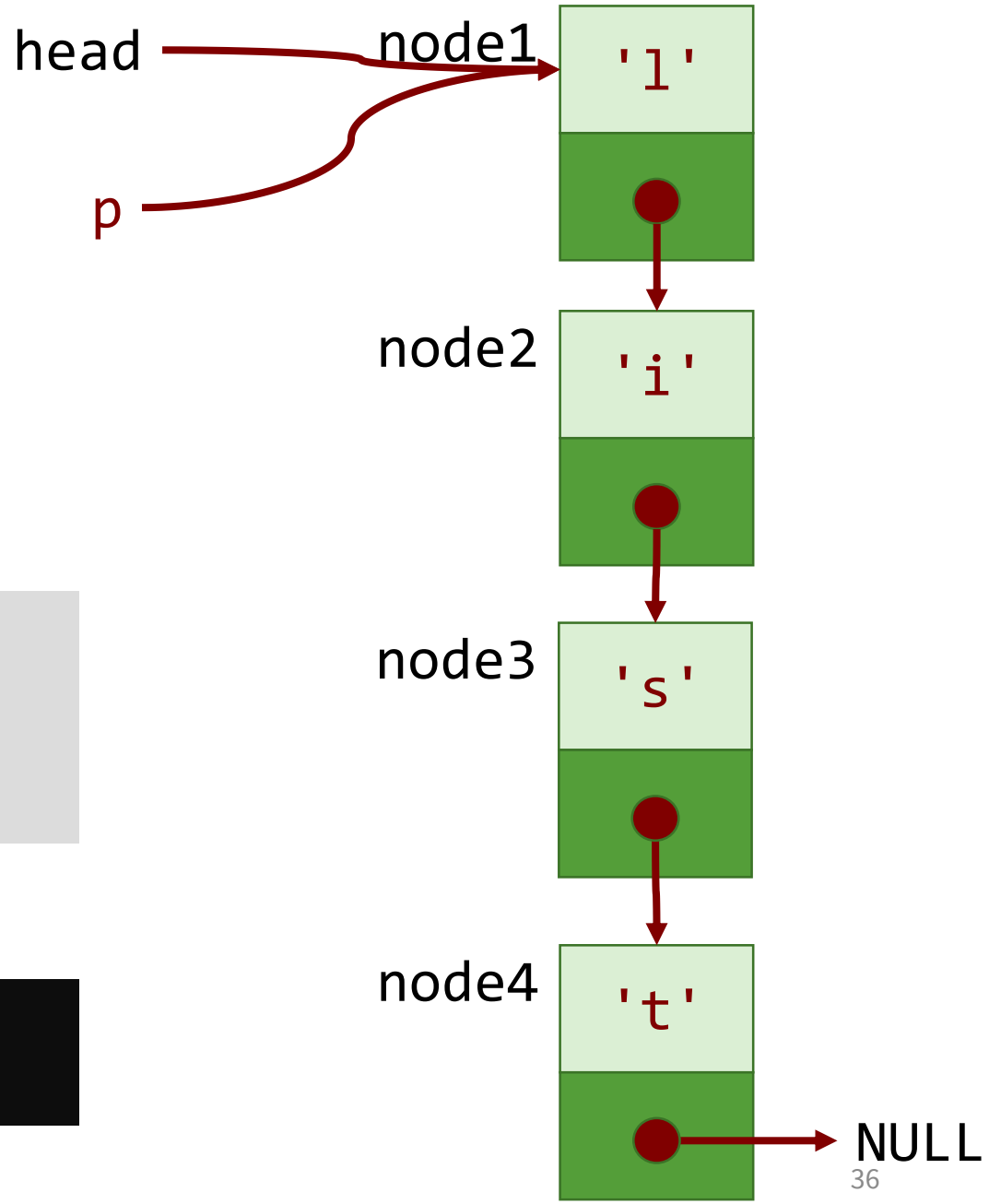
Singly-Linked List Traversal (3)

- 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)

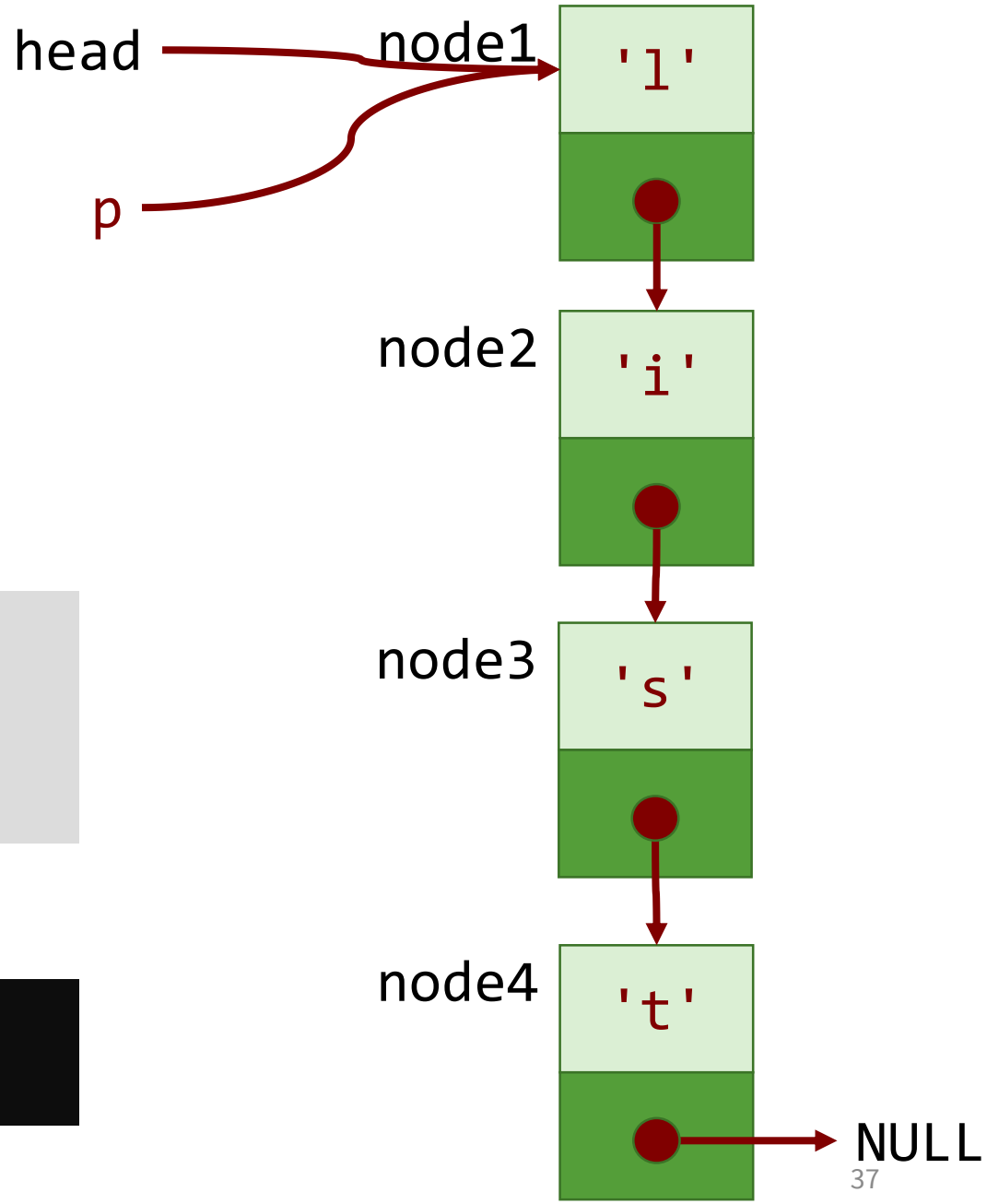
- Will always begin from head
- Visit every node until last node

- Example:

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

- Output:

```
l
```



Singly-Linked List Traversal (5)

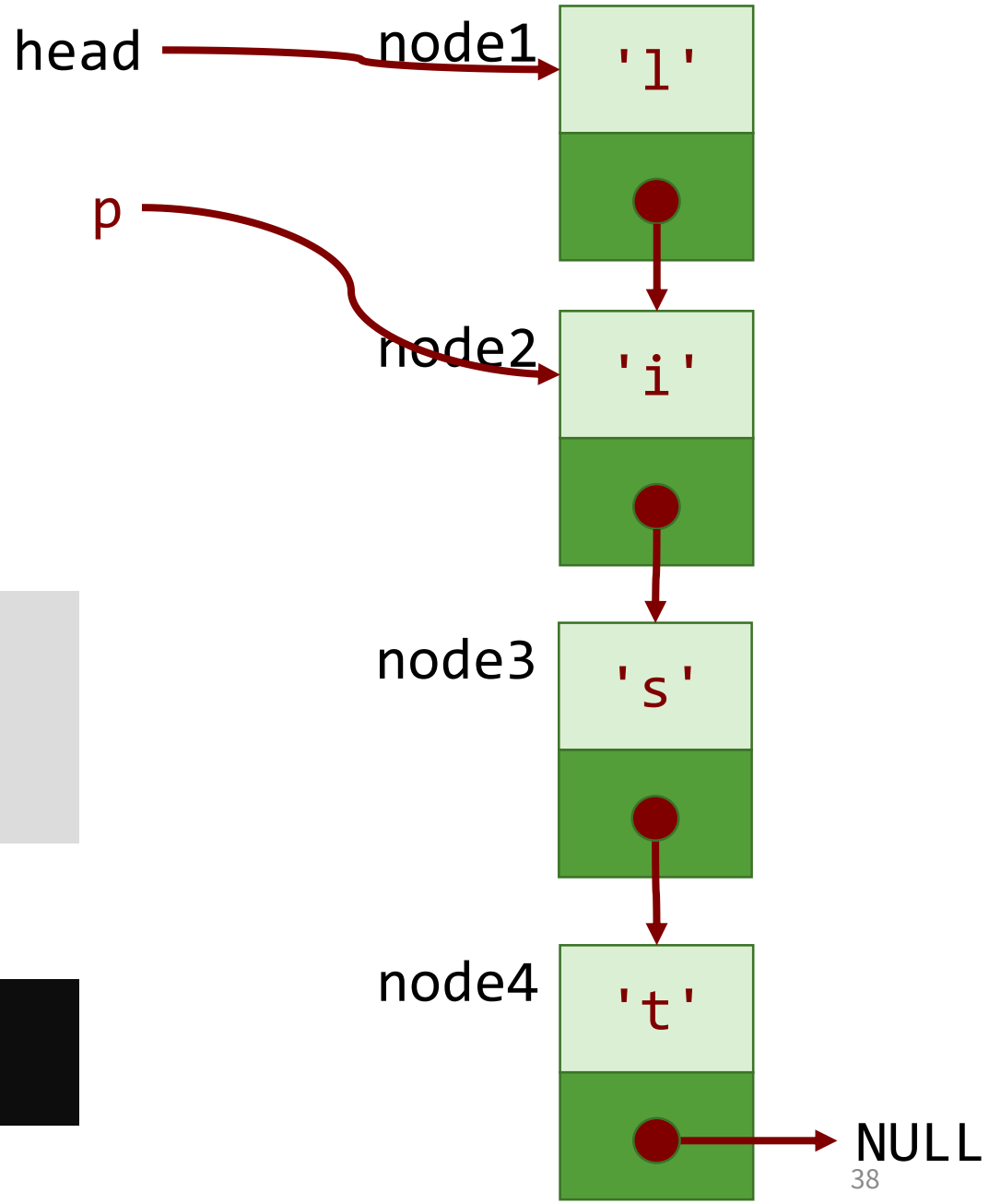
- Will always begin from head
- Visit every node until last node

- Example:

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

- Output:

```
l
```



Singly-Linked List Traversal (6)

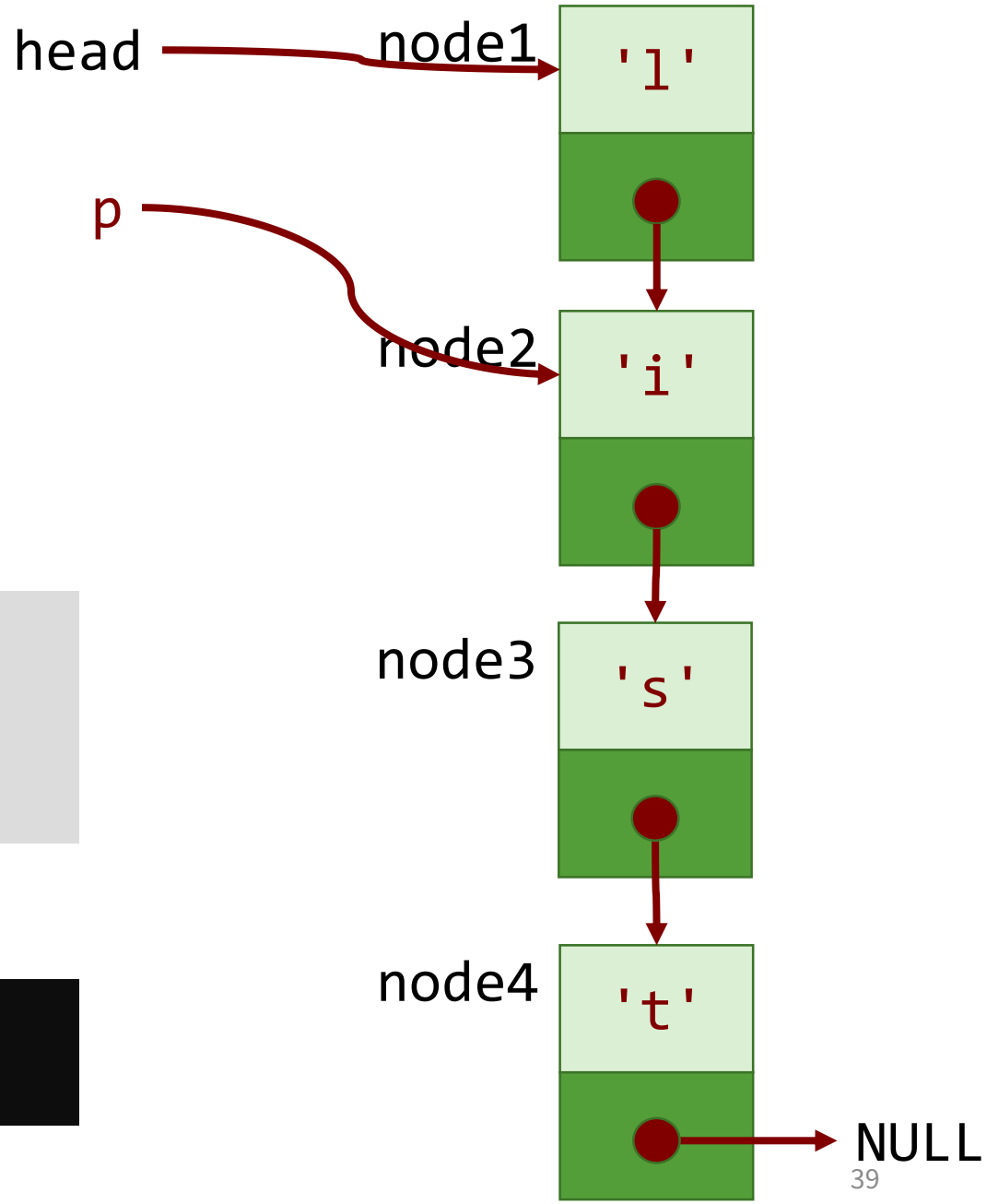
- Will always begin from head
- Visit every node until last node

- Example:

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

- Output:

```
l
```



Singly-Linked List Traversal (7)

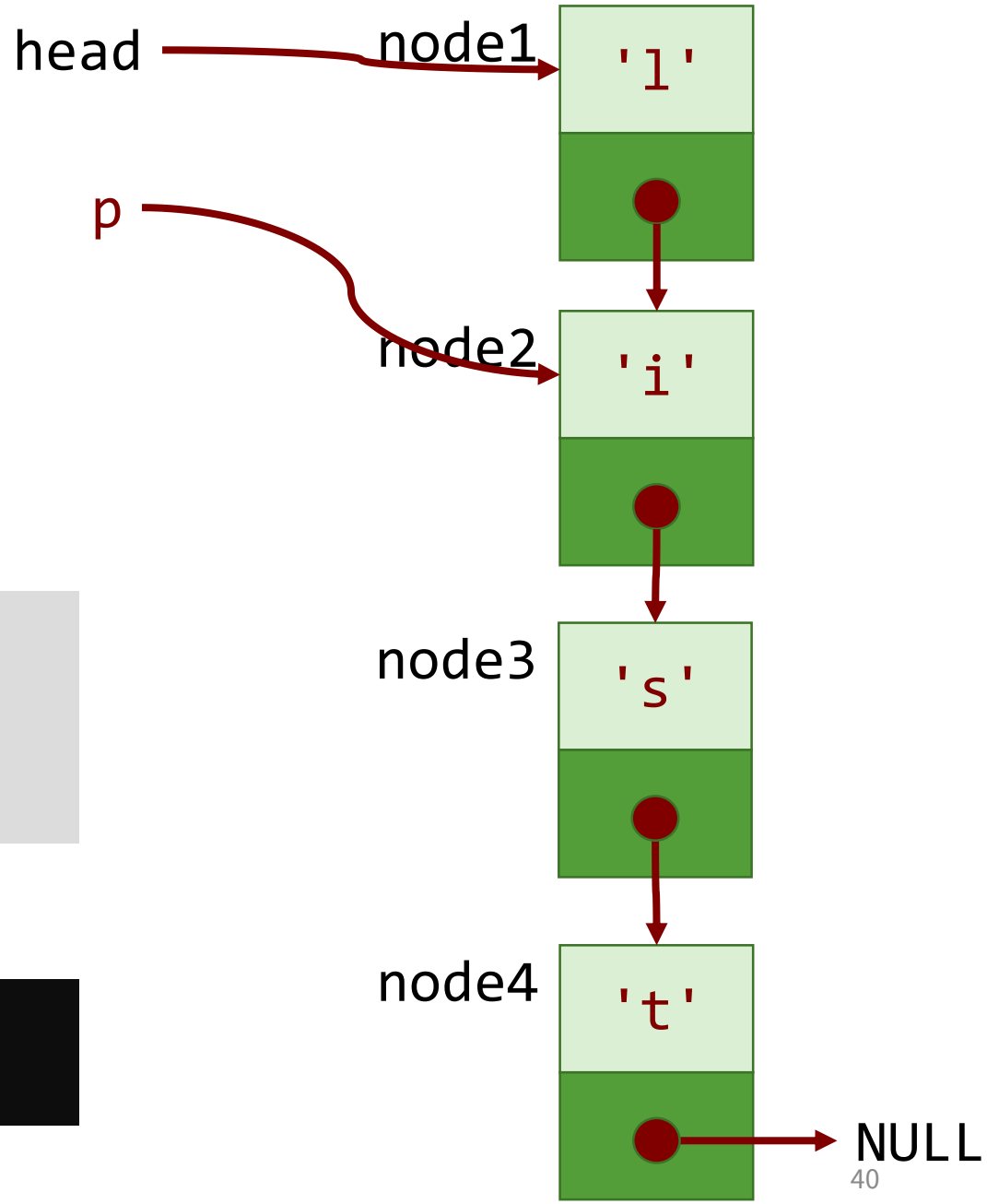
- Will always begin from head
- Visit every node until last node

- Example:

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

- Output:

```
li
```



Singly-Linked List Traversal (8)

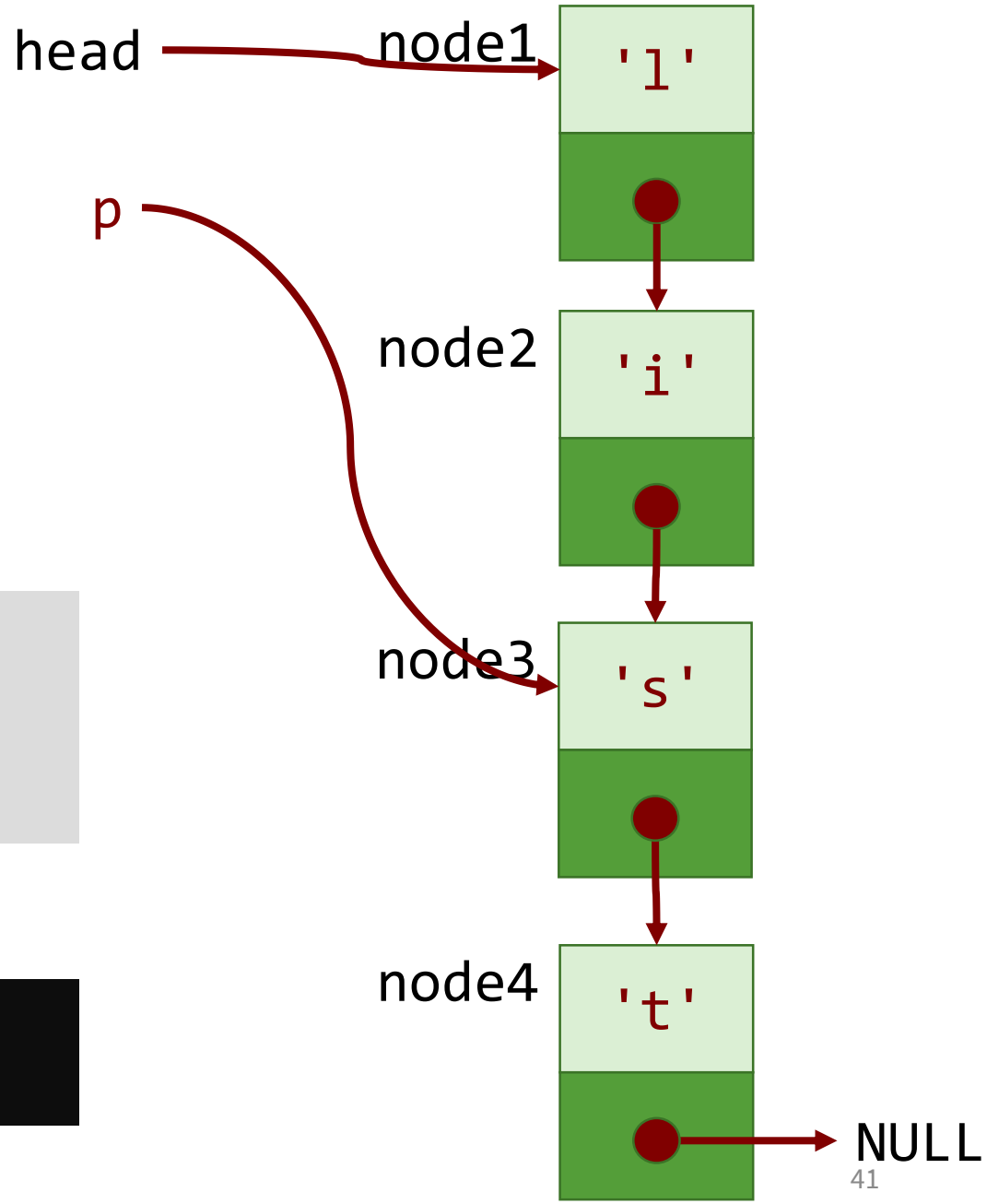
- Will always begin from head
- Visit every node until last node

- Example:

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

- Output:

```
li
```



Singly-Linked List Traversal (9)

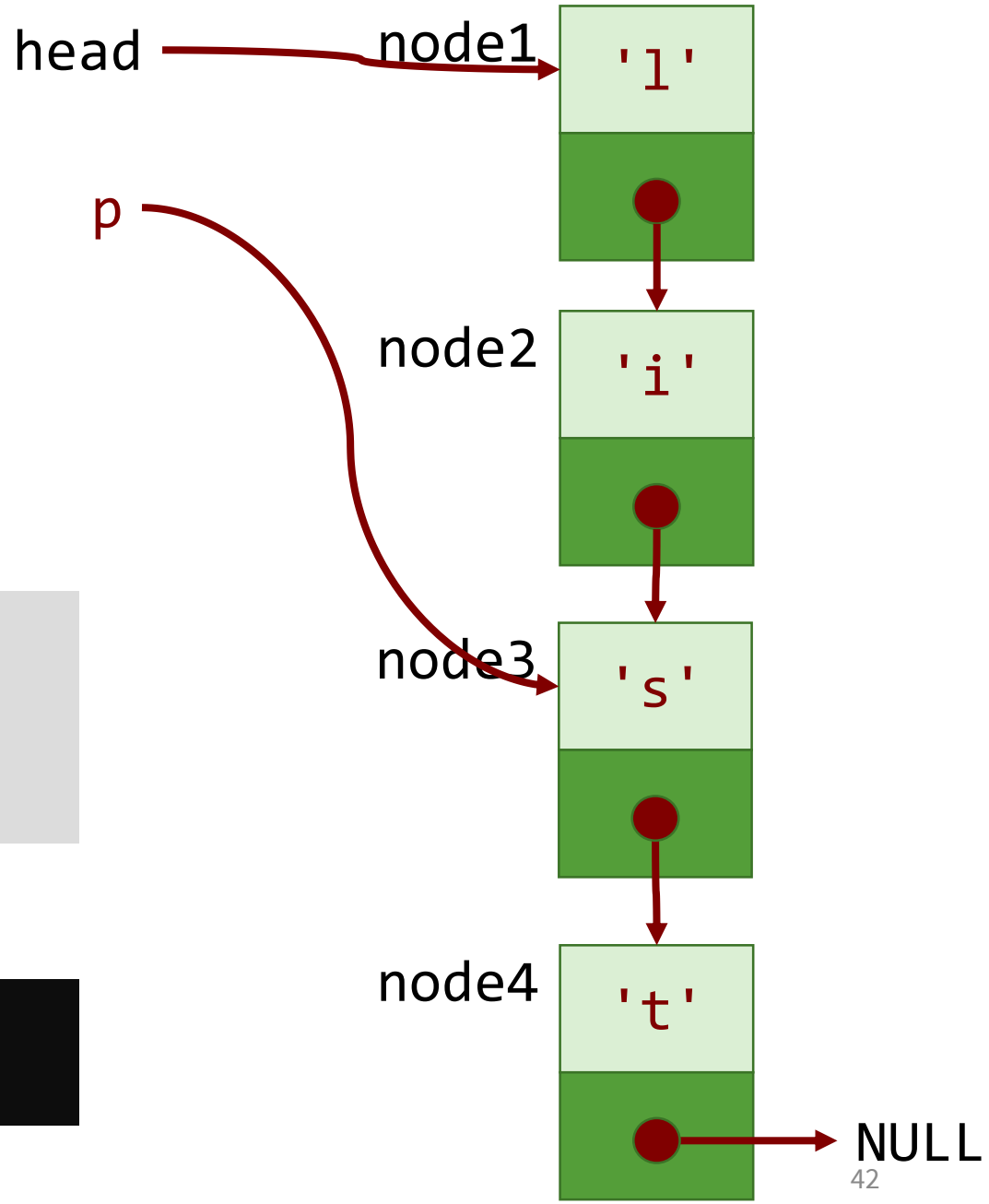
- Will always begin from head
- Visit every node until last node

- Example:

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

- Output:

```
li
```



Singly-Linked List Traversal (10)

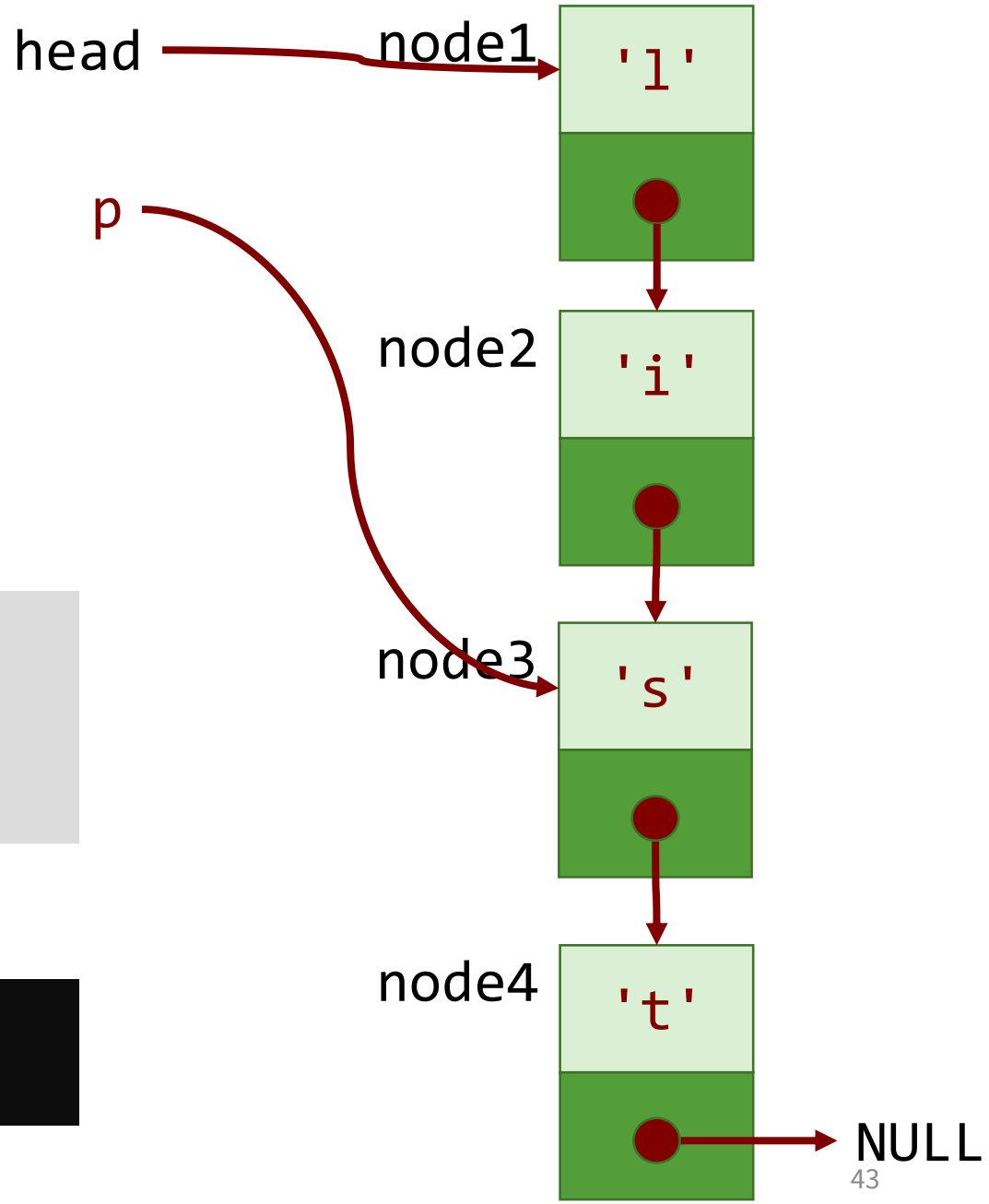
- Will always begin from head
- Visit every node until last node

- Example:

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

- Output:

```
lis
```



Singly-Linked List Traversal (11)

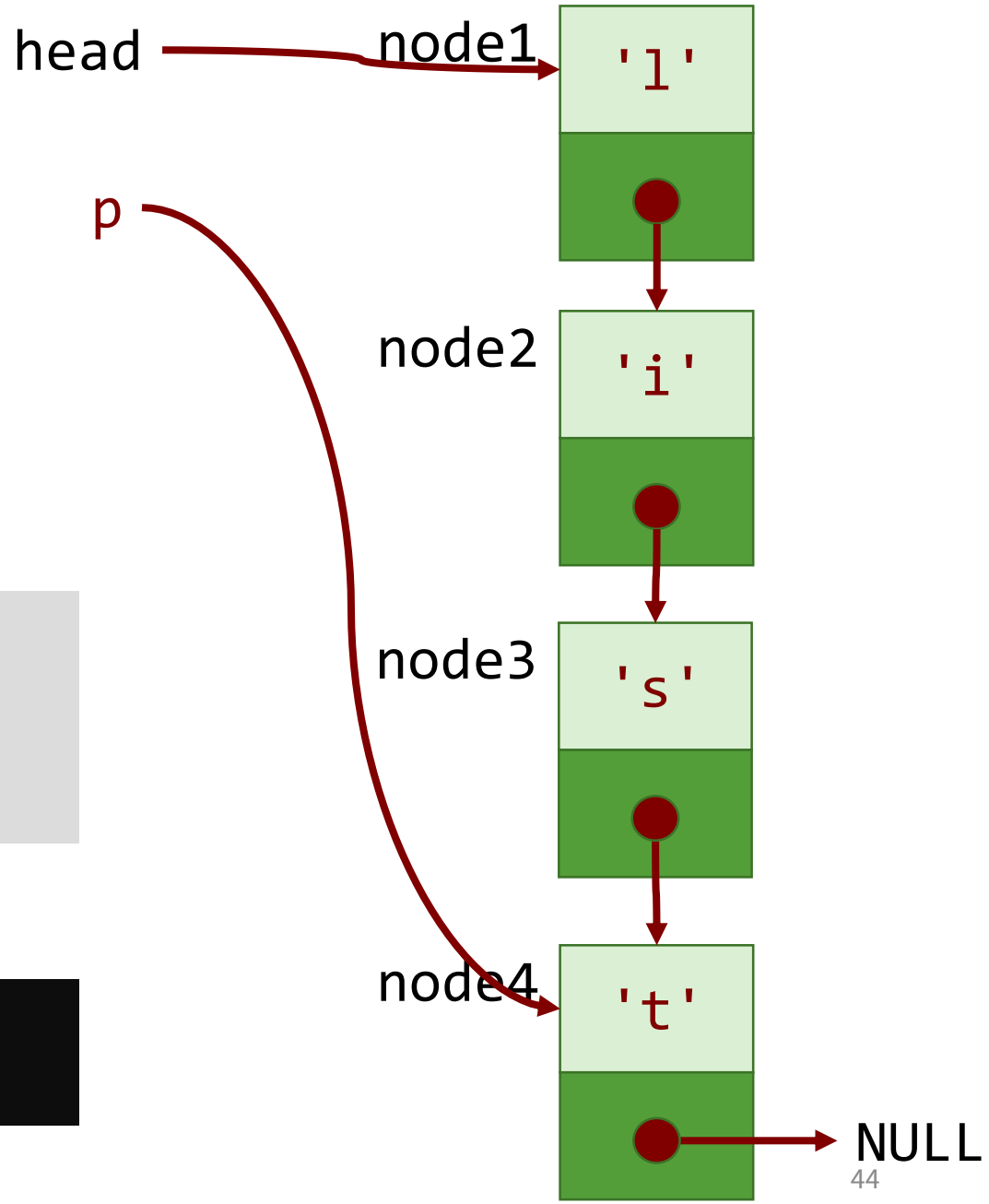
- Will always begin from head
- Visit every node until last node

- Example:

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

- Output:

```
lis
```



Singly-Linked List Traversal (12)

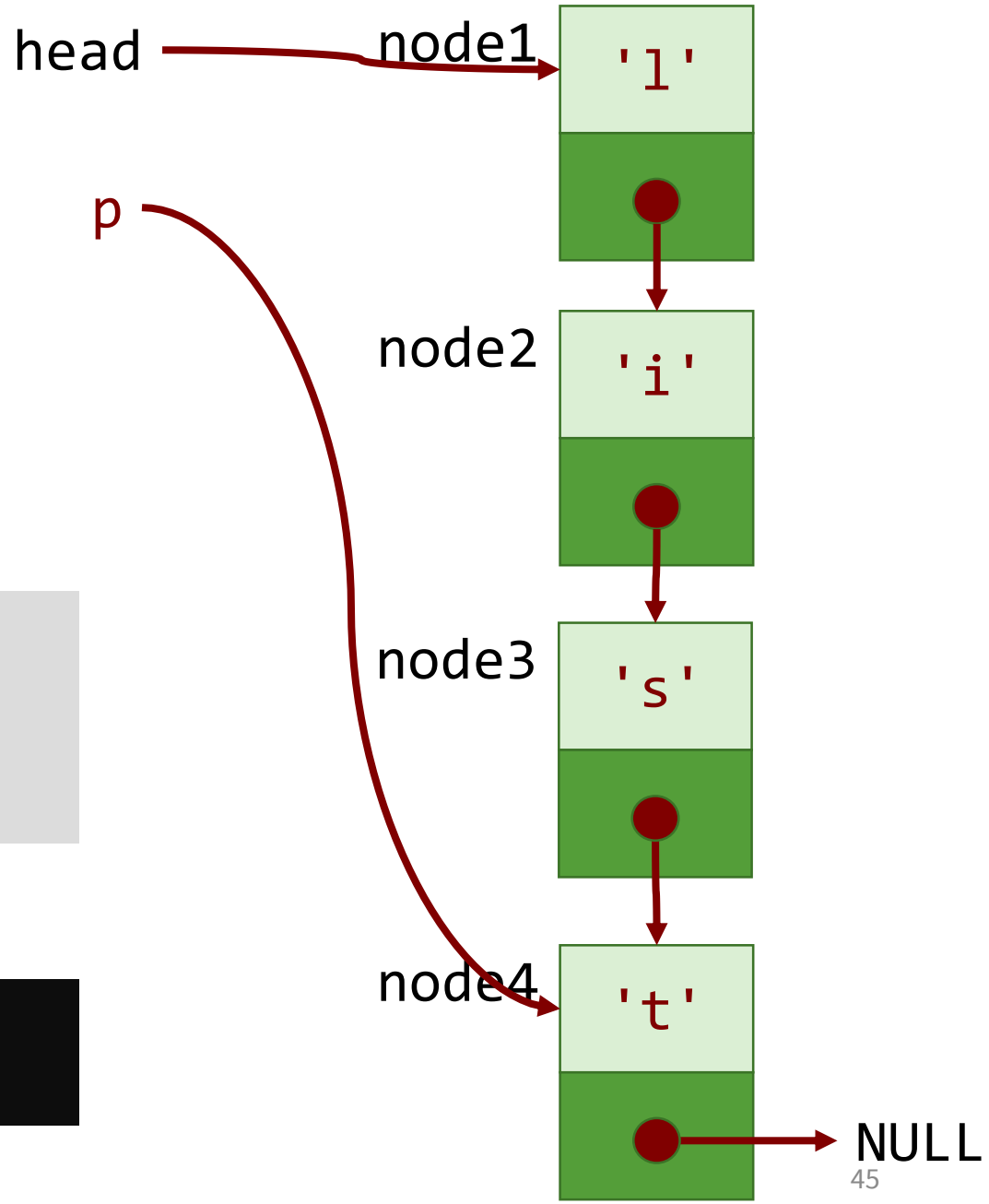
- Will always begin from head
- Visit every node until last node

- Example:

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

- Output:

```
lis
```



Singly-Linked List Traversal (13)

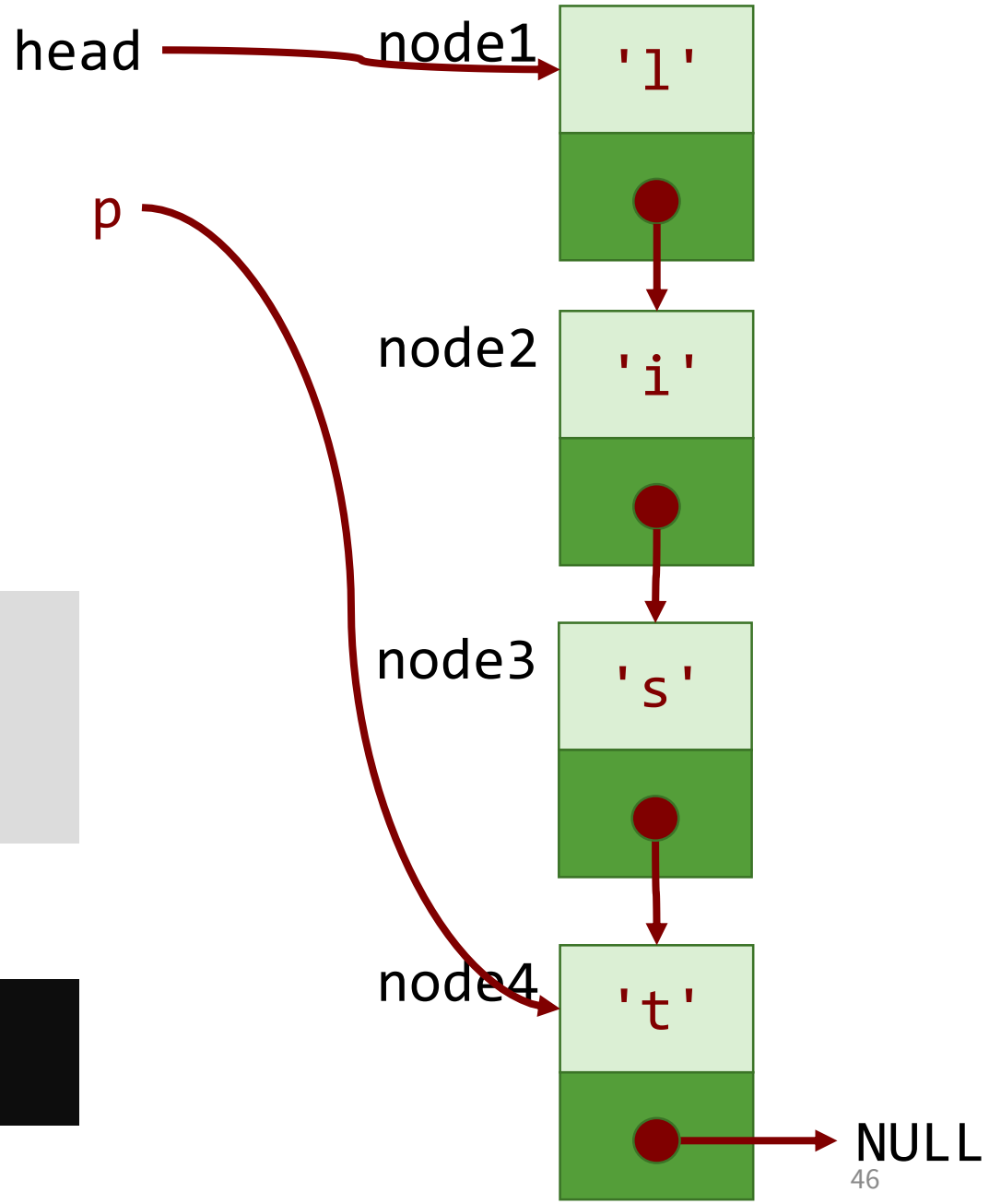
- Will always begin from head
- Visit every node until last node

- Example:

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

- Output:

```
list
```



Singly-Linked List Traversal (14)

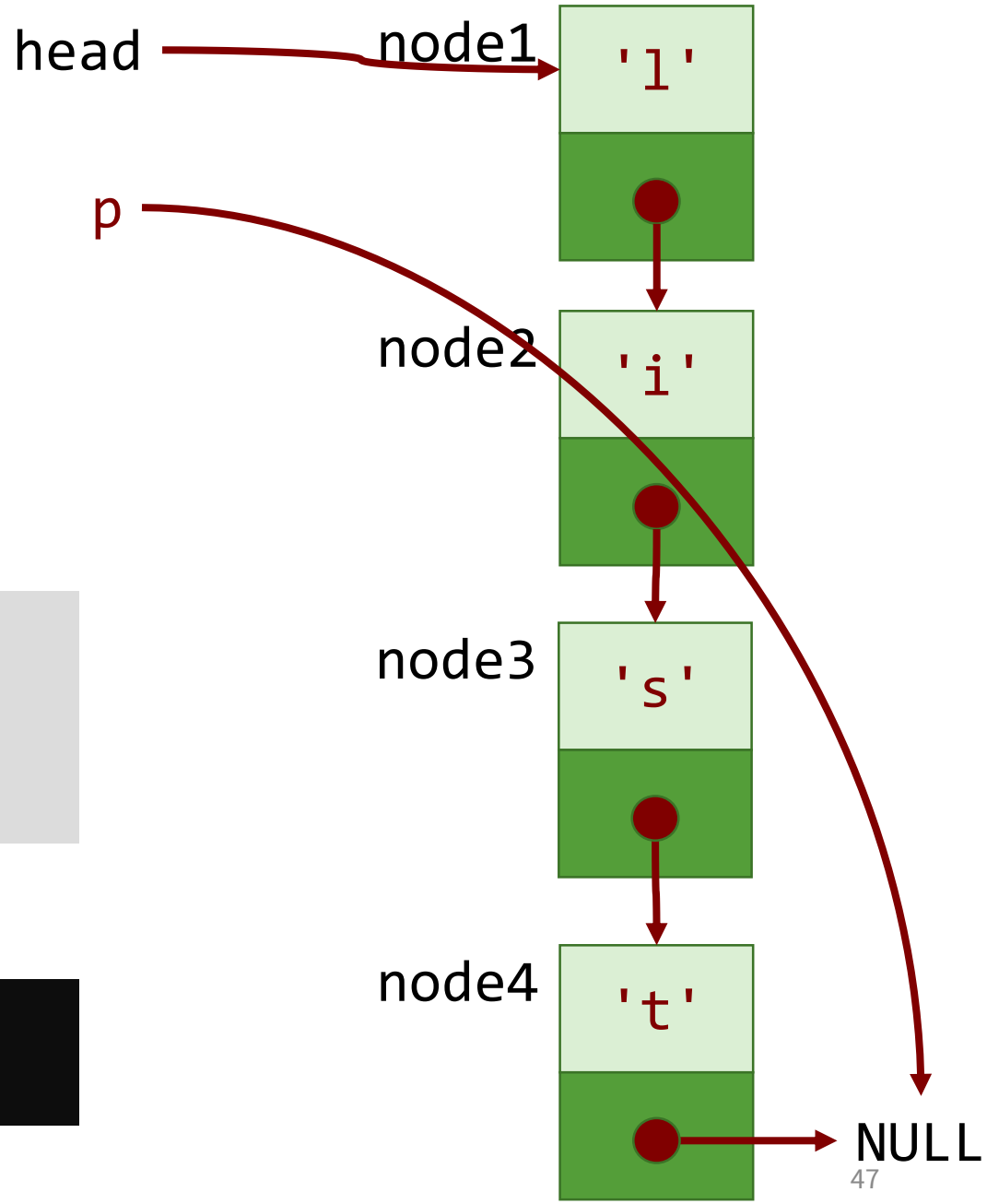
- Will always begin from head
- Visit every node until last node

- Example:

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

- Output:

```
list
```



Singly-Linked List Traversal (15)

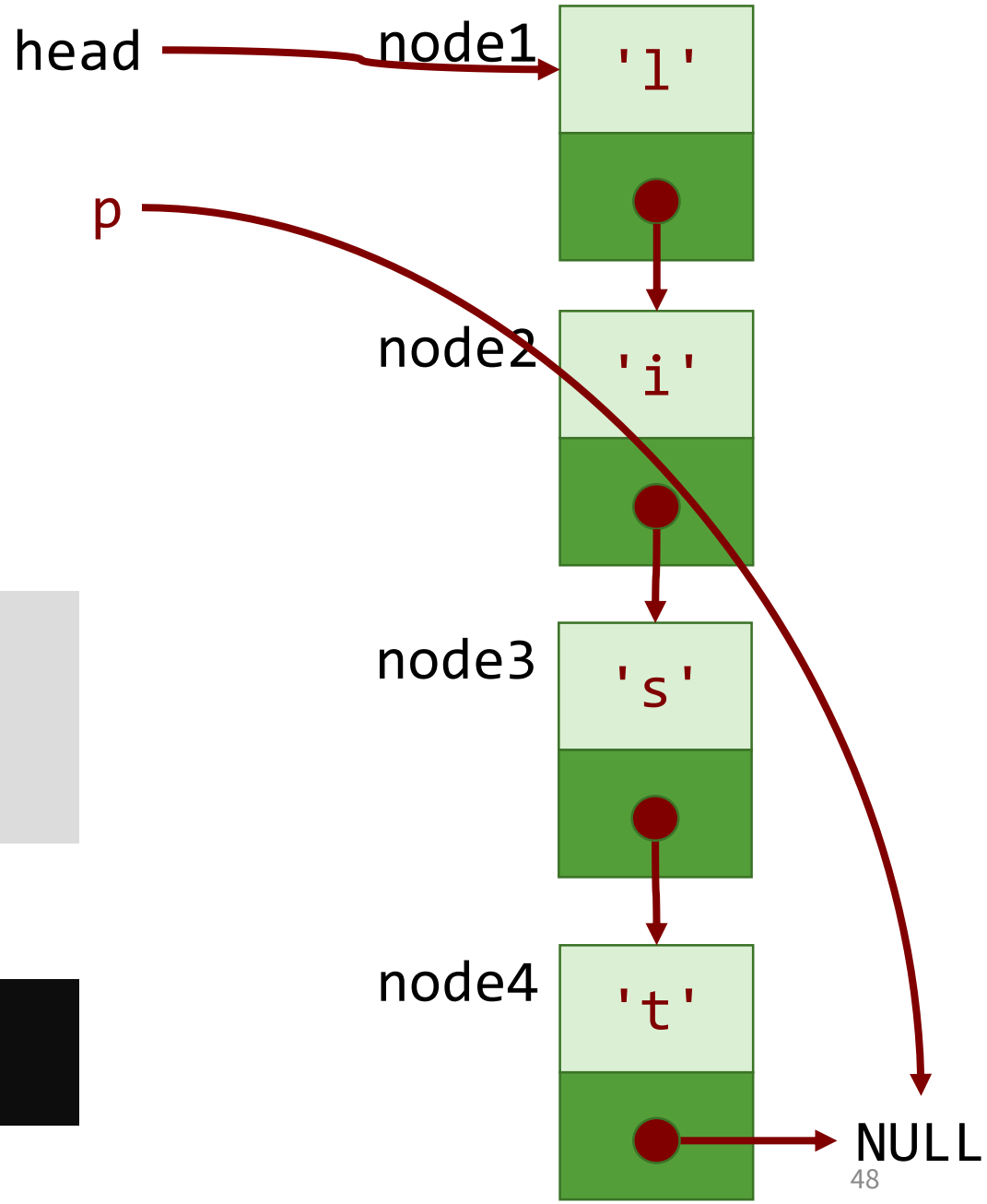
- Will always begin from head
- Visit every node until last node

- Example:

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

- Output:

```
list
```



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 (read additional Lecture Material on Linked Lists (not assessed))

Next Lecture

- User Defined Data types
- FILE Stream I/O