Week 5 Lecture 1

NWEN 241
Systems Programming

Alvin C. Valera

alvin.valera@ecs.vuw.ac.nz

Announcements

- Assignment 2 is out. Visit https://ecs.wgtn.ac.nz/Courses/NWEN241_2022T1/Assignments to download handout and sample test files

- Exercise 2 (2.5% of course marks) is due on Friday, 1 April 2022, 23:59

- Teaching feedback available @Blackboard until 6 April 2021
  - Please let me know what you think of my teaching
  - Things that you like and areas for improvement

Content

User-Defined Types

- Enumeration
- Union

Introduction to Linked Lists

Background

- Basic data types
  - int : integer ✓
  - char : character ✓
  - float : floating point number ✓
  - double : double-precision floating point number ✓
- Derived data types
  - Arrays ✓
  - Strings ✓
  - Structures ✓
  - Unions
- User defined data types
  - *Enumeration* types

Motivation for Enumeration Type

- Oftentimes, a variable can only take a few possible discrete values
- Macro is often used to define symbolic constants that will represent possible values of the variable
- **Enumeration is a better alternative**

```
#define COLOR_RED     0
#define COLOR_YELLOW  1
#define COLOR_GREEN   2

int main(void)
{
    int color;
    // can either be 0, 1 or 2
    …
    color = COLOR_GREEN;
}
```

Enumeration

- Enumeration is a user-defined data type that is used to assign identifiers to **integral constants**

- Declaration syntax:

```
enum enum_tag {name_0, name_1, ..., name_n} variable_list;
```

  - Defines a new enumerated type
  - Defines symbolic constants that take on integer values from `0` through `n`
    - → `name_0` has value `0`, `name_1` has value `1`, and so on

Enumeration

- Enumeration is a user-defined data type that is used to assign identifiers to **integral constants**

- Declaration syntax:

```
enum enum_tag {name_0, name_1, ..., name_n} variable_list;
```

  - *enum_tag* and *variable_list* are optional

Enumeration

As an example, the statement:

```
enum colors { red, yellow, green };
```

- Defines a new enumerated type `enum colors`
- Defines three integer constants: `red` is assigned the value 0, `yellow` is assigned 1 and `green` is assigned 2
- Any variable of `enum colors` type or basic data type can be assigned either `red`, `yellow` or `green`

Enumeration

Unnamed enumeration example:

```
enum { red, yellow, green };
```

- Defines three integer constants: `red` is assigned the value 0, `yellow` is assigned 1 and `green` is assigned 2
- Any variable of basic data type can be assigned either `red`, `yellow` or `green`

Enumeration

- It is possible to override the integer assignment, e.g.

```
enum colors {red = 3, yellow = 2, green = 1};
```

- `typedef` can be used to create an alias for the new type, e.g.

```
typedef enum colors {red = 3, yellow = 2, green = 1} color_t;
```

  - `color_t` is a new type which can be used for declaring variables

Enumeration

- If an identifier is assigned a value and subsequent identifiers are not assigned, the subsequent identifiers continue the progression from the assigned value

```
enum colors { red, yellow = 3, green, blue };
```

`red` is assigned the value 0, `yellow` is assigned 3, `green` is assigned 4, and `blue` is assigned 5.

enum Example (1)

```
#include <stdio.h>

/* Declaration defines new enumerated type and integer constants */
enum colors { red, yellow = 3, green, blue };

int main(void)
{
   /* Declaration defines variables of type enum colors */
   /* Can take values of red, yellow, green or blue */
   enum colors fgcolor = blue, bgcolor = yellow;

   printf ("%d %d\n", fgcolor, bgcolor);
   /* Will print 5 3 */

   return 0;
}
```

enum Example (2)

```
#include <stdio.h>

/* Declaration defines integer constants */
enum { red, yellow = 3, green, blue };

int main(void)
{
   /* Declaration defines variables of type int */
   /* Can be assigned red, yellow, green or blue */
   int fgcolor = blue, bgcolor = yellow;

   printf ("%d %d\n", fgcolor, bgcolor);
   /* Will print 5 3 */

   return 0;
}
```

Repeated Identifiers

- An identifier in an enumerated type cannot be re-used to declare a new variable or enumeration in the same scope

```
void func(void)
{
   enum colors { red, yellow, black };
   enum rgb { red, green, blue };
    …
}
```

```
void func(void)
{
   enum colors { red, yellow, black };
   int red;
    …
}
```

Will not compile due to re-use of identifier red in the same scope

Unions

- A union is like a struct, but the different fields take up the **same** space within memory

- Declaration syntax:

```
union union_tag {
        type1 member1;
        type2 member2;
        ...
} variable_list;
```

- ***union_tag*** specifies the name of the union
- ***union_tag*** and ***variable_list*** are optional
- If ***union_tag*** is not specified, ***variable_list*** should be specified; otherwise, there is no way to declare variables using the unnamed union type

Unions

- A union is like a struct, but the different fields take up the **same** space within memory

- Declaration syntax:

```
union union_tag {
        type1 member1;
        type2 member2;
        ...
} variable_list;
```

- Union members can be
  - Basic data types
  - Derived and user-defined types
  - Pointers to basic, derived and user-defined data types
  - Function pointers

Union vs Structure

	Structure	Union
Declaration syntax	Same	
Storage allocation	Allocates storage for all members separately	• Allocates common storage for all its members • Space is allocated to hold the biggest member
Access	All members can be accessed at the same time	Only one member can be "active" at any given time

Union vs Structure: Storage Allocation

```c
struct space {
    int i;
    float f;
    char c[4];
};
```

```c
union space {
    int i;
    float f;
    char c[4];
};
```

`sizeof(struct space) = sizeof(i) + sizeof(f) + sizeof(c)`

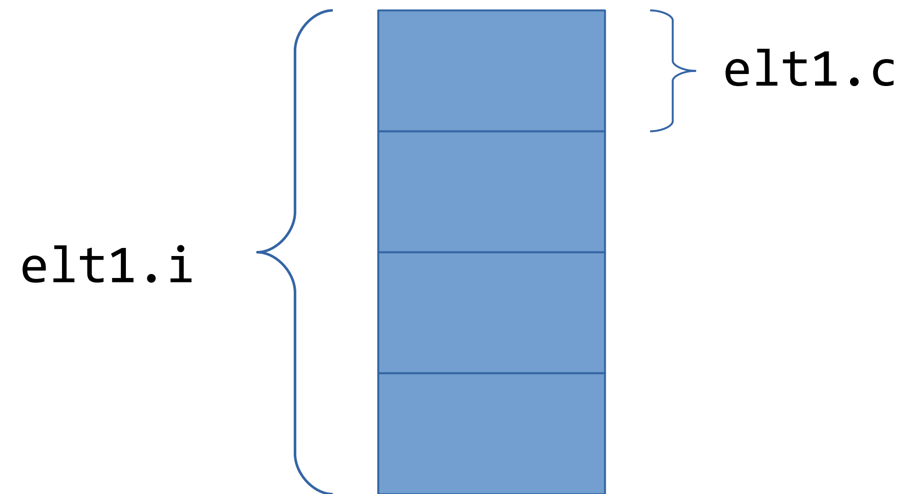
`sizeof(union space) =` *max*`(sizeof(i), sizeof(f), sizeof(c))`

`union` Example

```
union elt {
    int   i;
    char  c;
} elt1;

elt1.c = 'A';
elt1.i = 300;
```

Assuming an int takes up 32 bits (4 bytes):

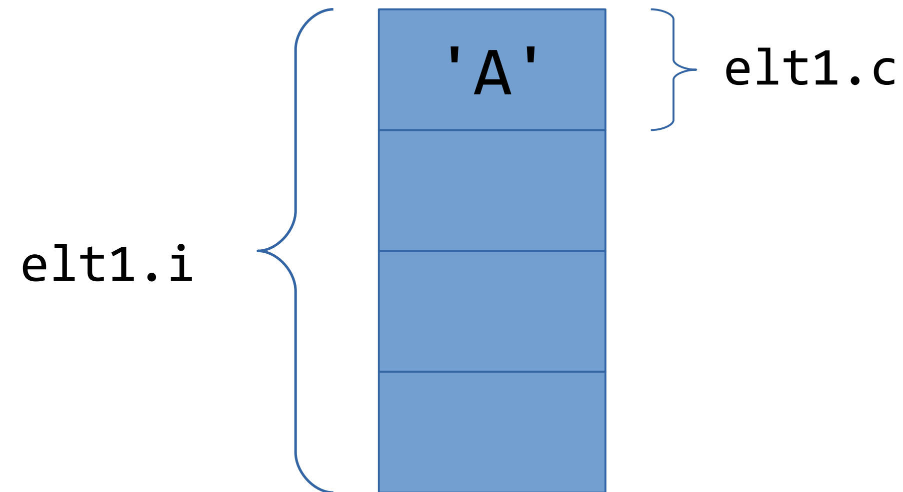


union Example

```
union elt {
    int   i;
    char  c;
} elt1;

elt1.c = 'A';
elt1.i = 300;
```

Assuming an int takes up 32 bits (4 bytes):

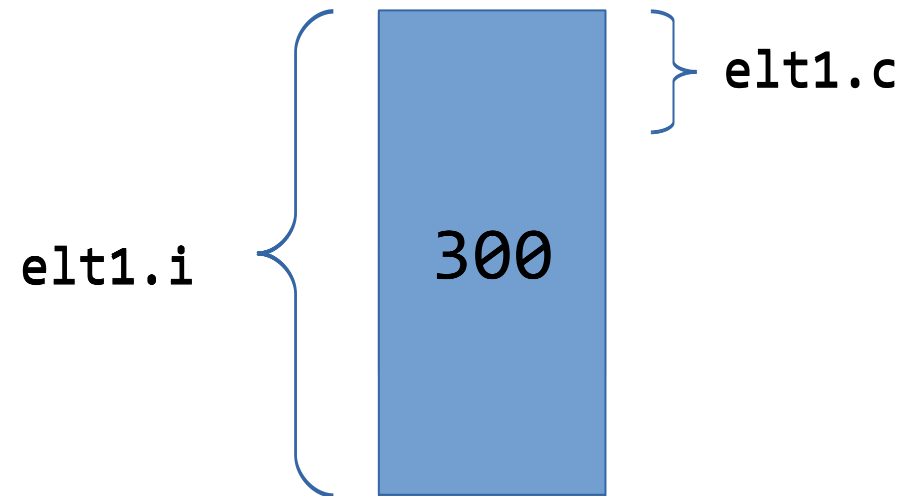


union Example

```
union elt {
    int   i;
    char  c;
} elt1;

elt1.c = 'A';
elt1.i = 300;
```

Assuming an int takes up 32 bits (4 bytes):



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

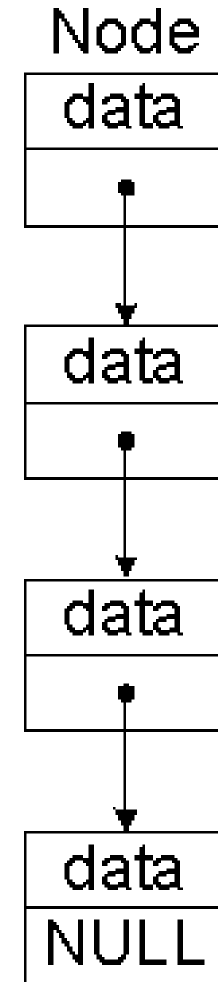
Linked Data Structures

- 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;
```

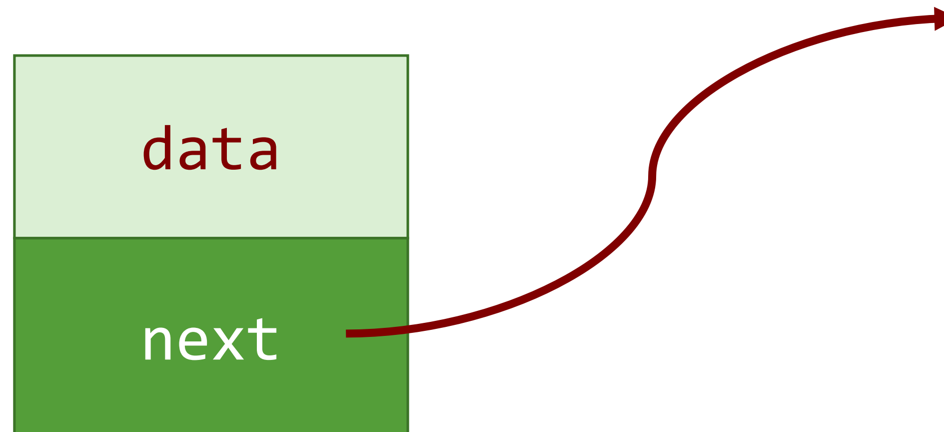
- A singly-linked list is the simplest example



Singly-Linked List (1)

- Node type definition

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



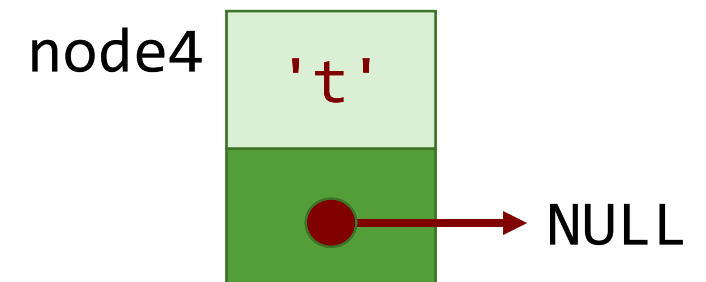
Singly-Linked List (2)

- 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;
```



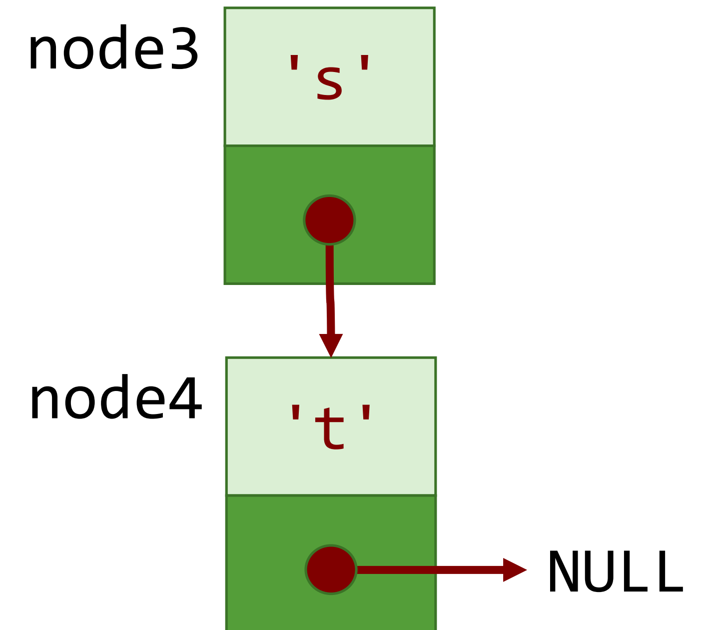
Singly-Linked List (3)

- 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;
```



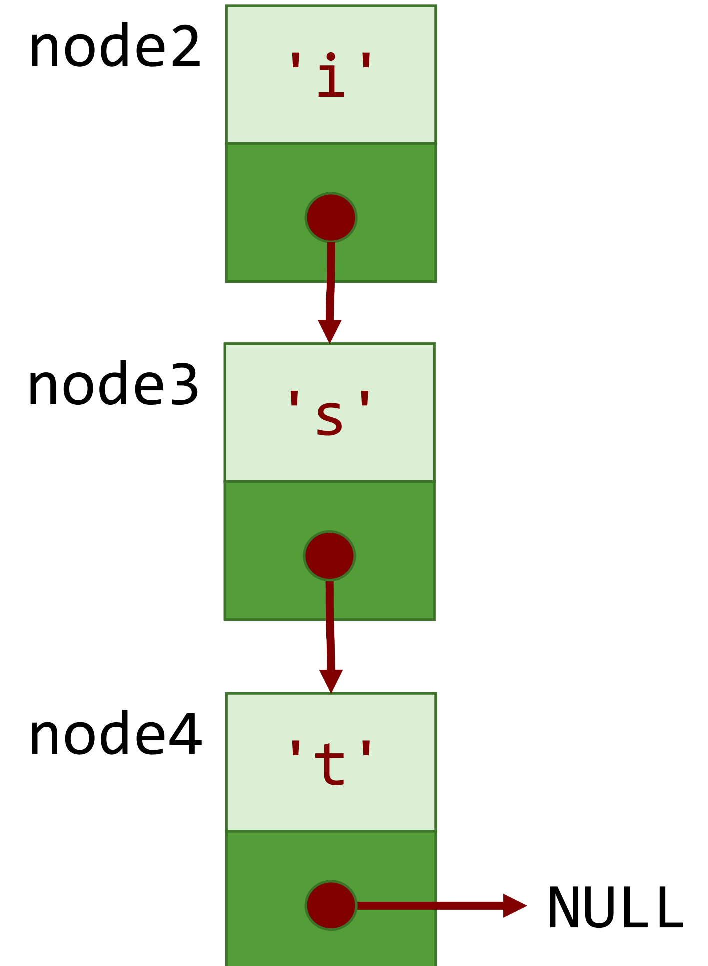
Singly-Linked List (4)

- 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;
```



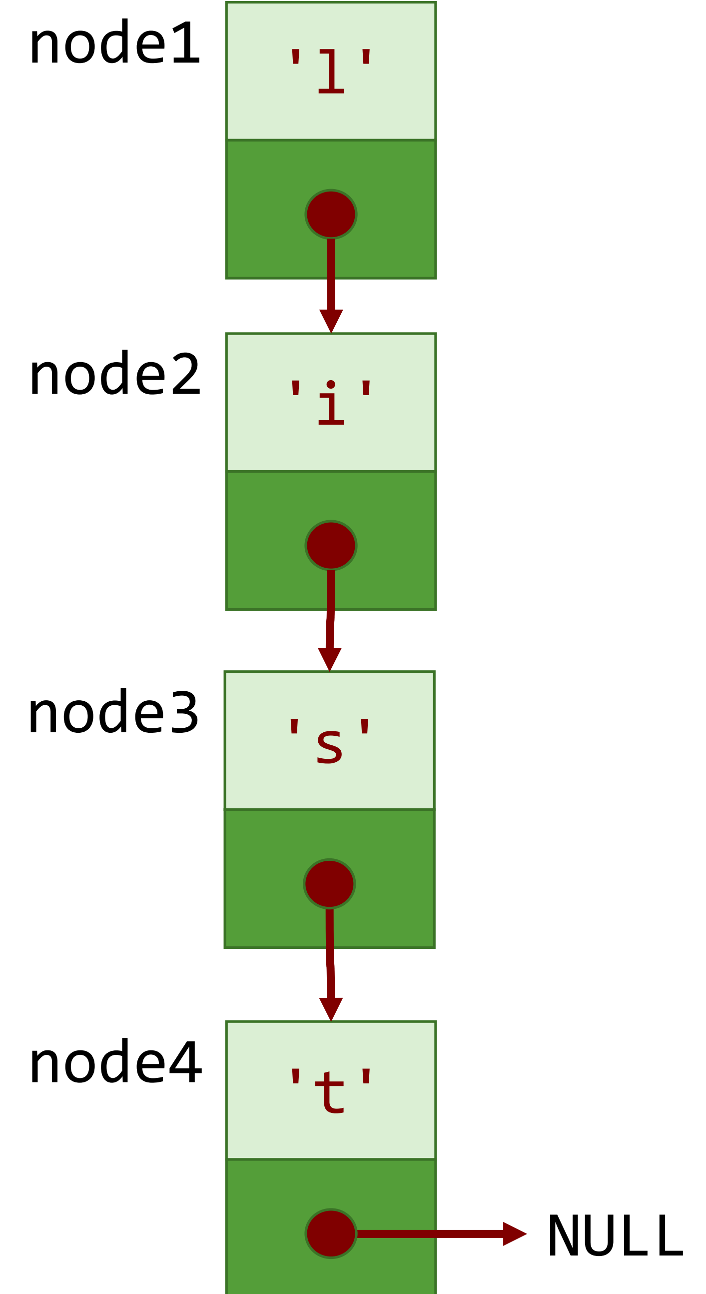
Singly-Linked List (5)

- 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;
```



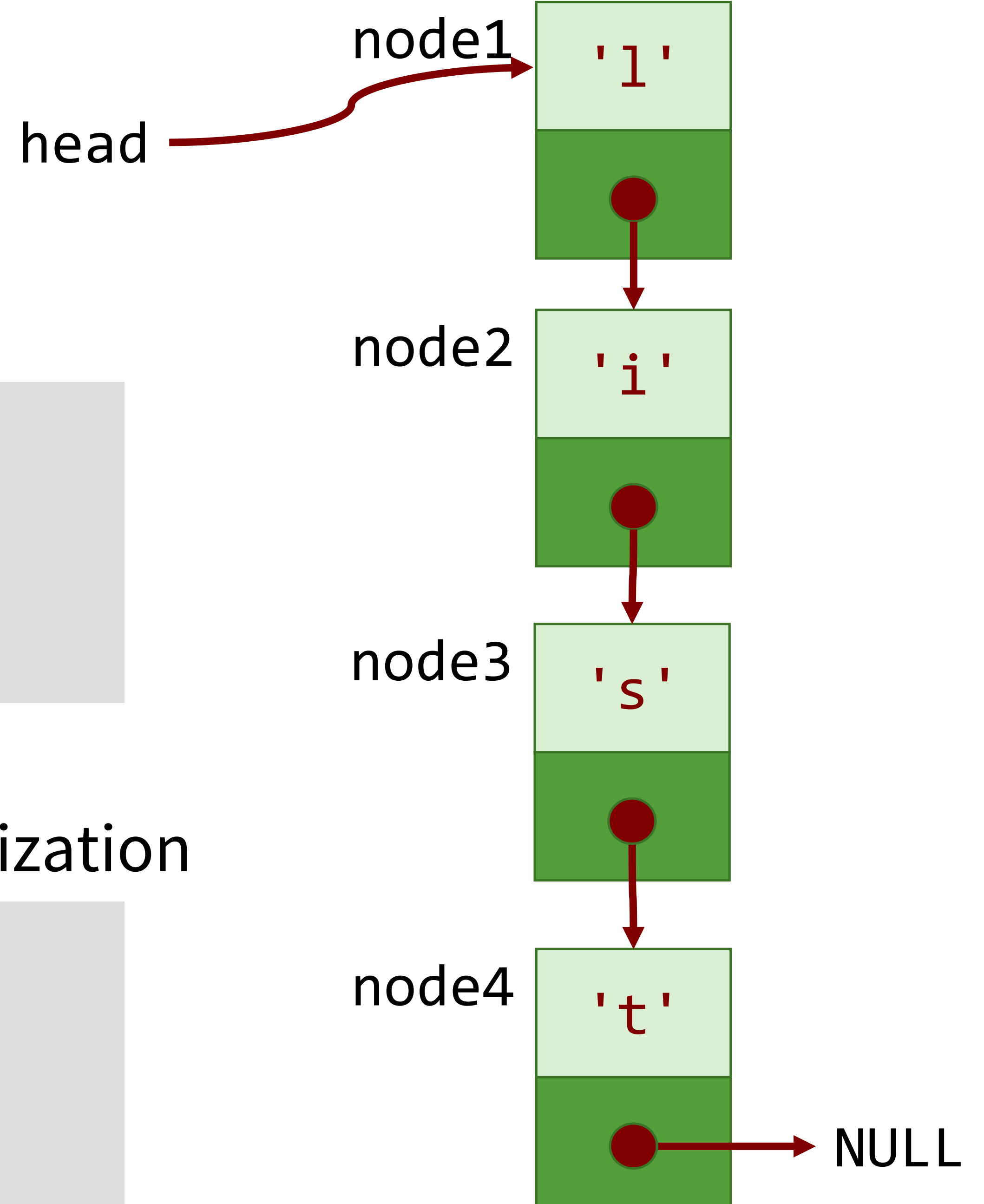
Singly-Linked List (6)

- 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);
```

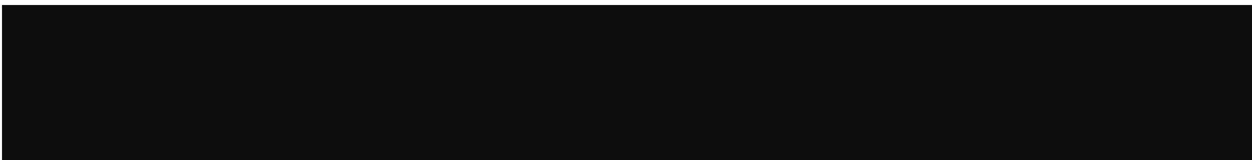
Singly-Linked List Traversal (2)

- 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:



head
p
node1 'l'
node2 'i'
node3 's'
node4 't'
NULL

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:

head
p
node1 'l'
node2 'i'
node3 's'
node4 't'
NULL

Singly-Linked List Traversal (4)

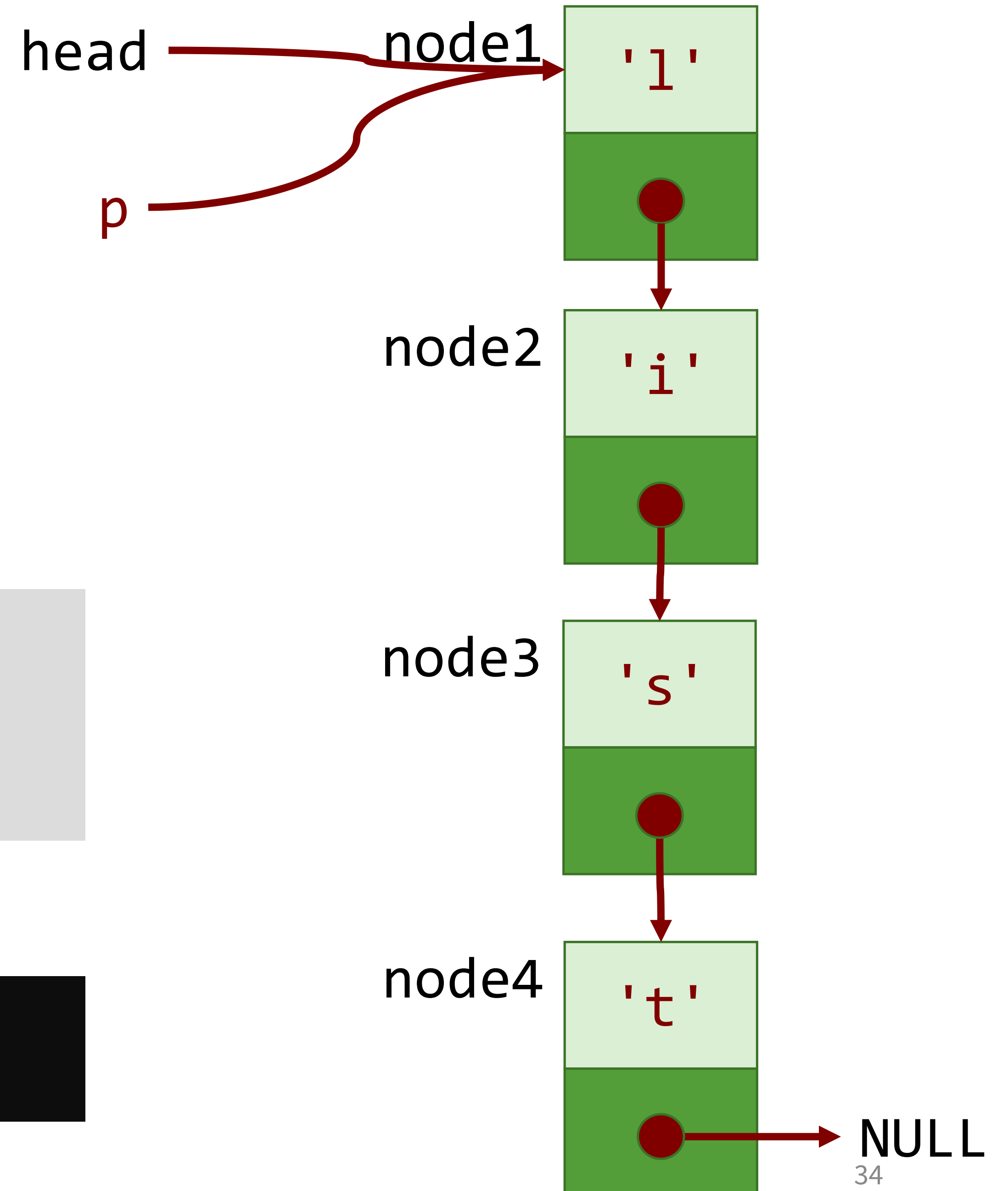
- Will always begin from head
- Visit every node until last node

- Example:

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

- Output:

```
l
```



Singly-Linked List Traversal (5)

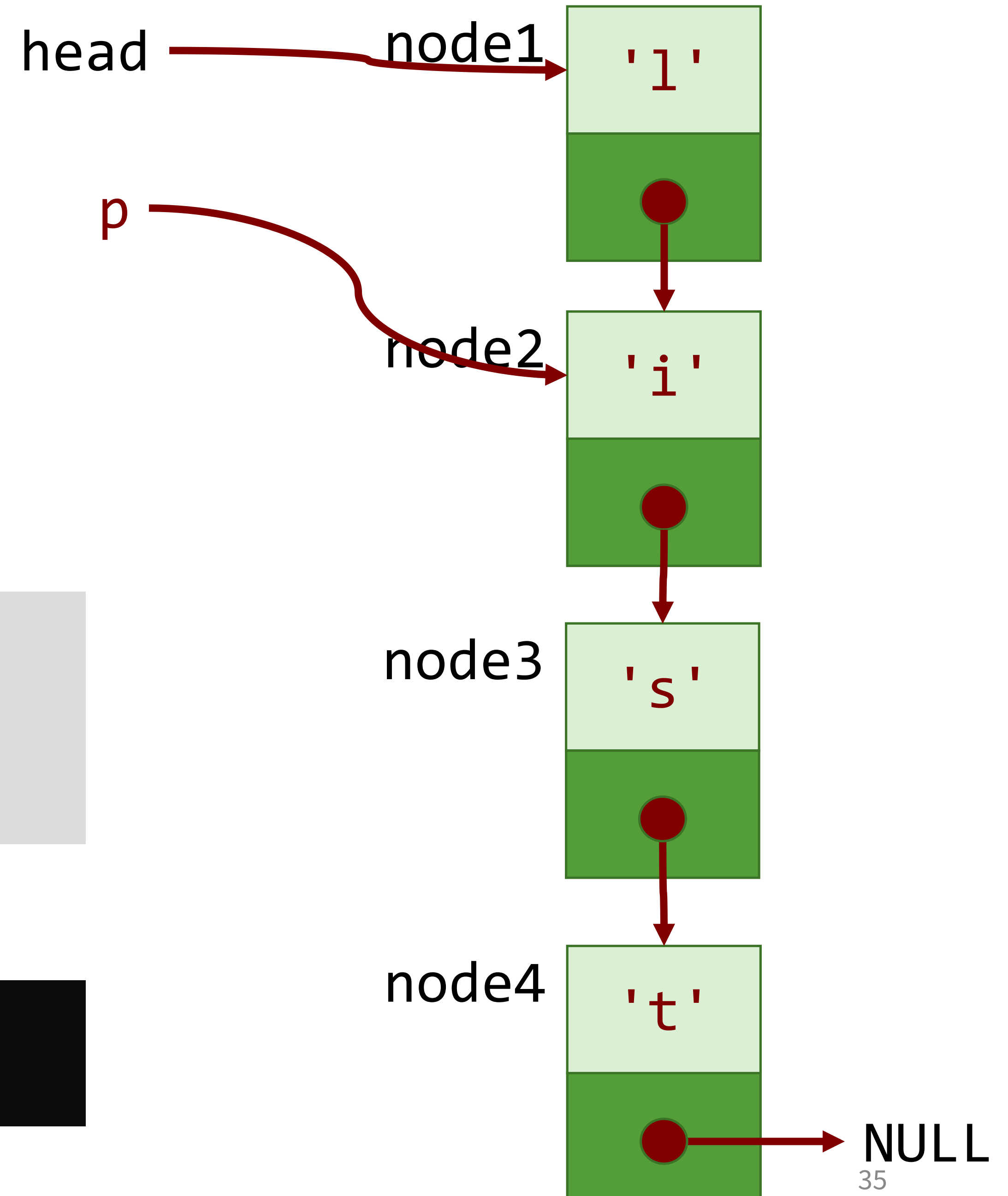
- Will always begin from head
- Visit every node until last node

- Example:

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

- Output:

```
l
```



Singly-Linked List Traversal (6)

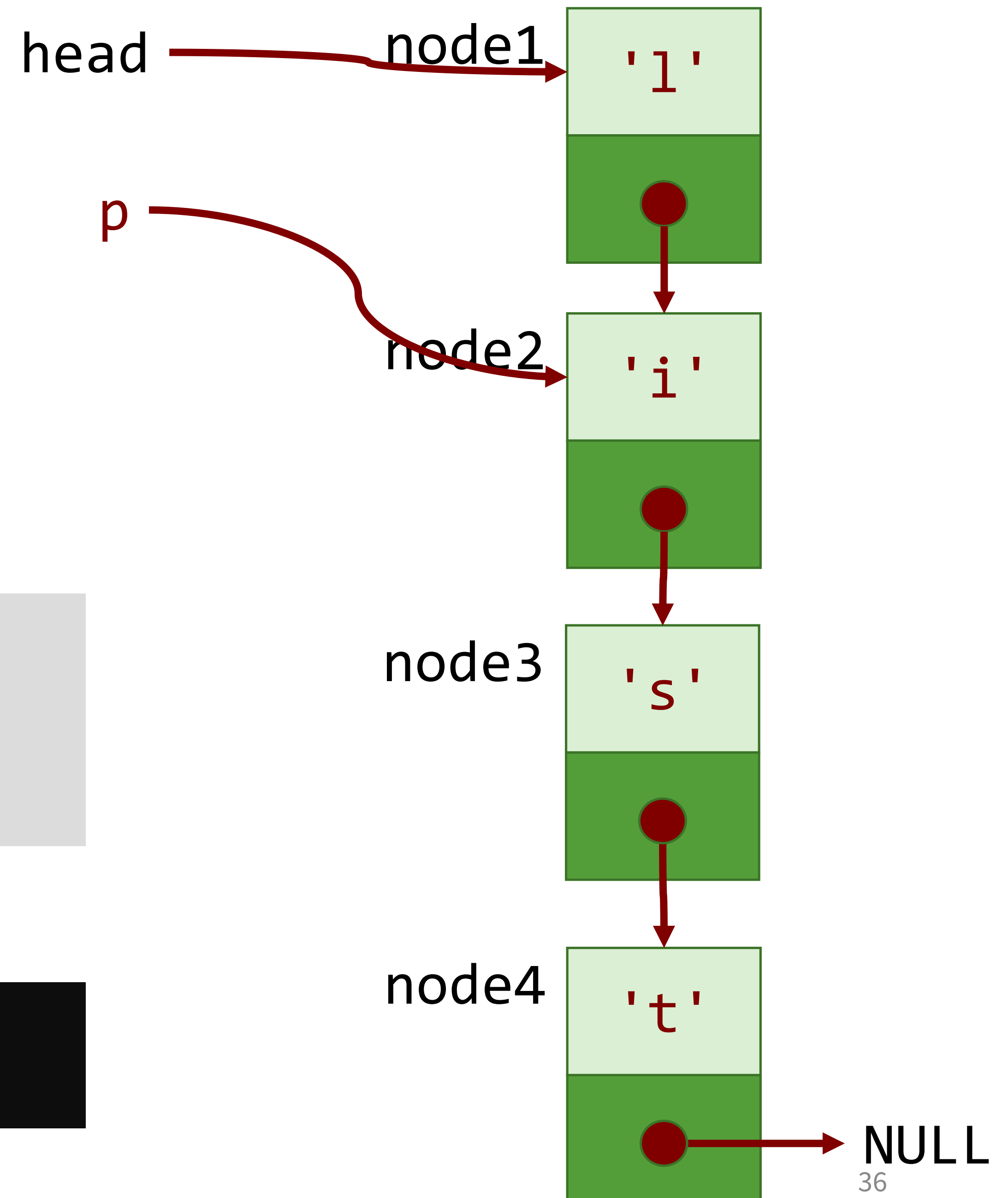
- Will always begin from head
- Visit every node until last node

- Example:

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

- Output:

```
l
```



Singly-Linked List Traversal (7)

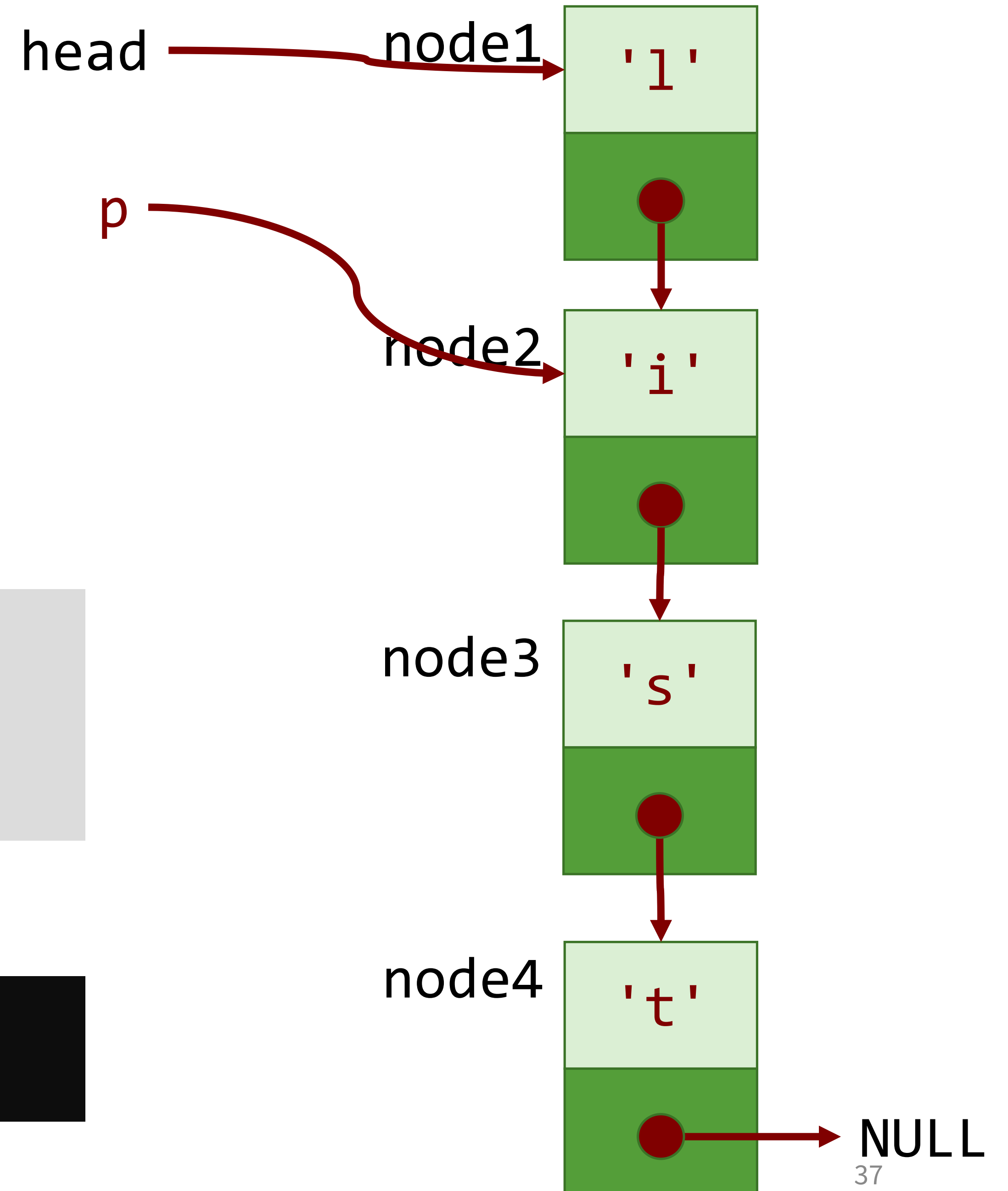
- Will always begin from head
- Visit every node until last node

- Example:

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

- Output:

```
li
```



Singly-Linked List Traversal (8)

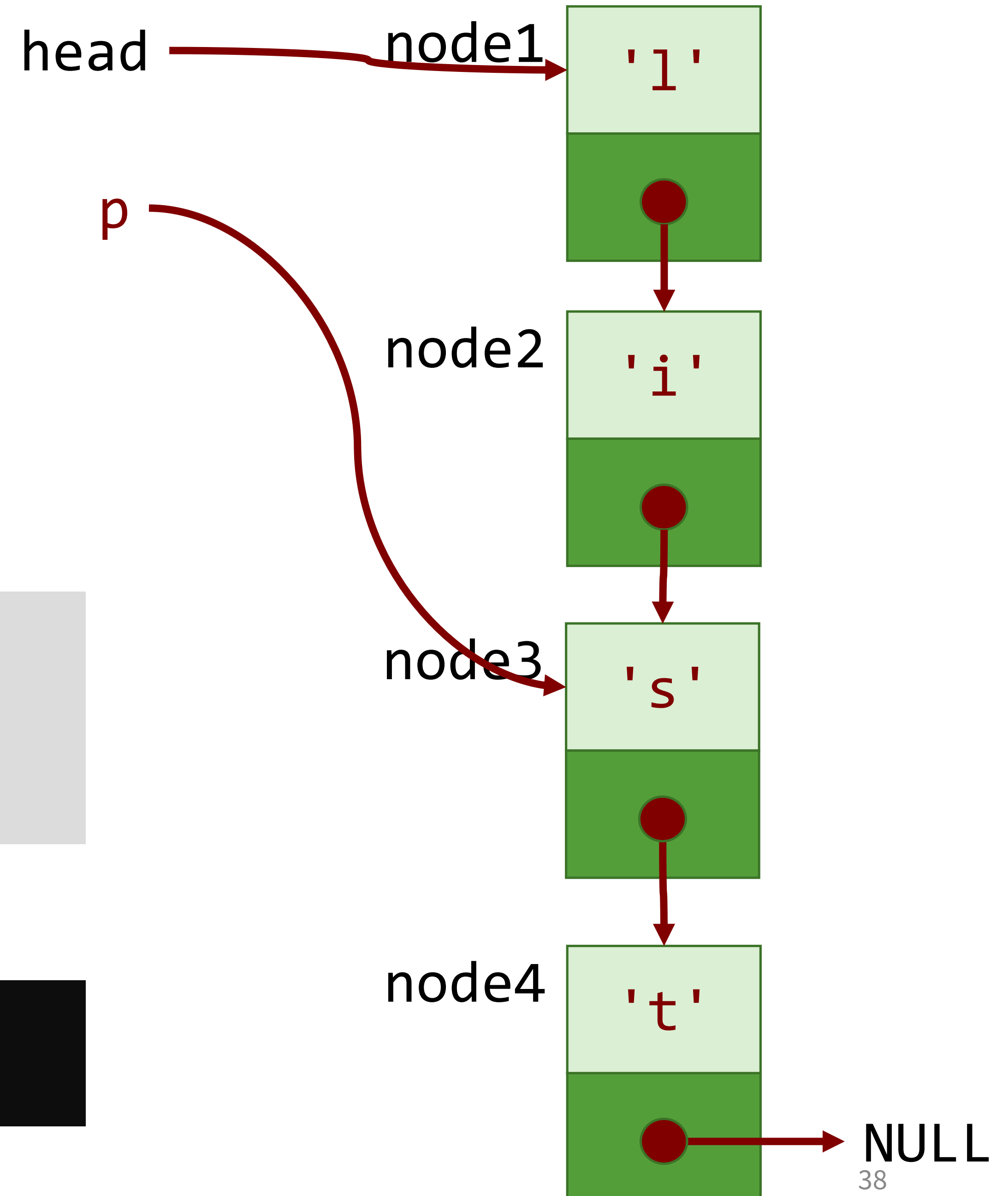
- Will always begin from head
- Visit every node until last node

- Example:

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

- Output:

```
li
```



Singly-Linked List Traversal (9)

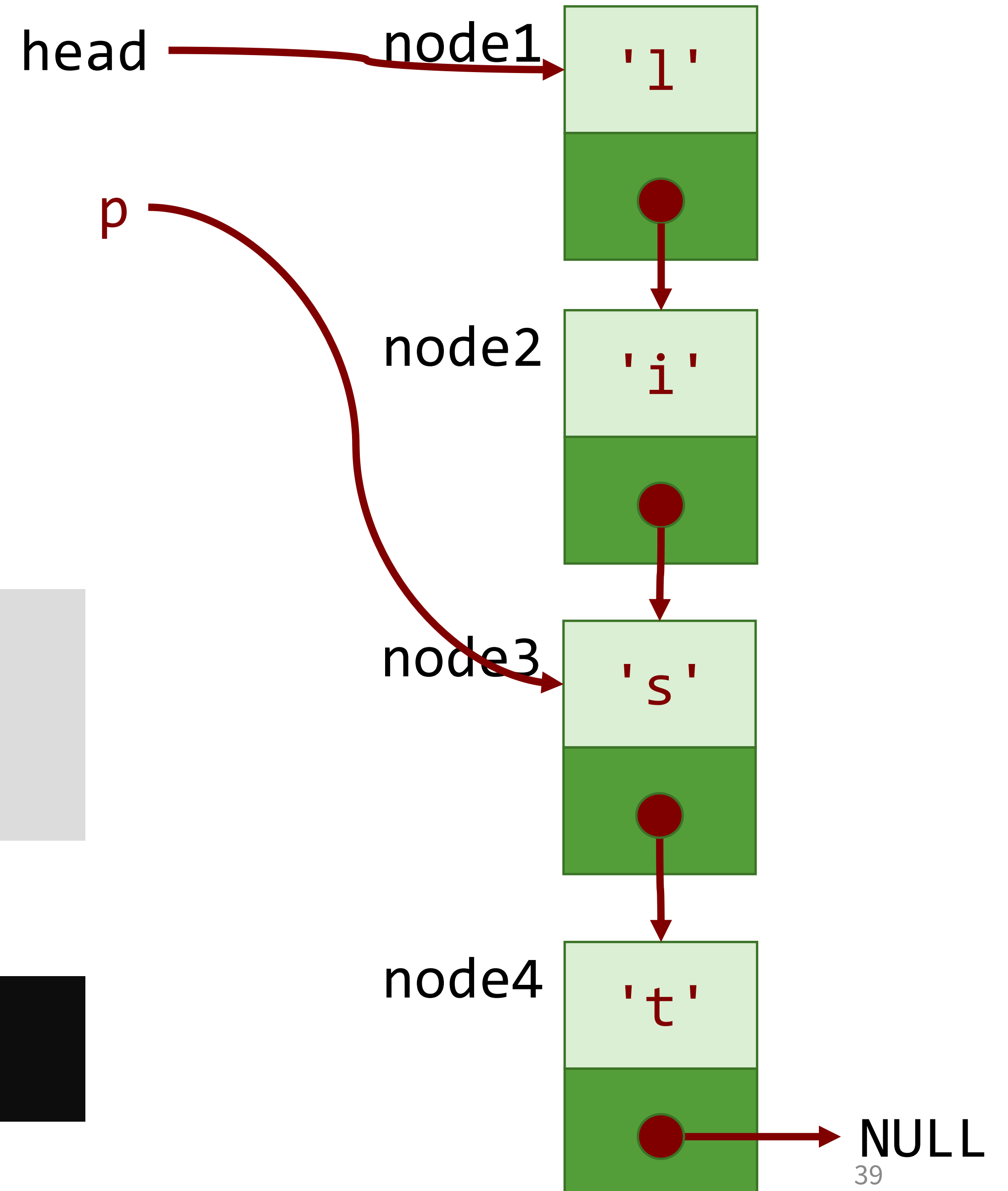
- Will always begin from head
- Visit every node until last node

- Example:

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

- Output:

```
li
```



Singly-Linked List Traversal (10)

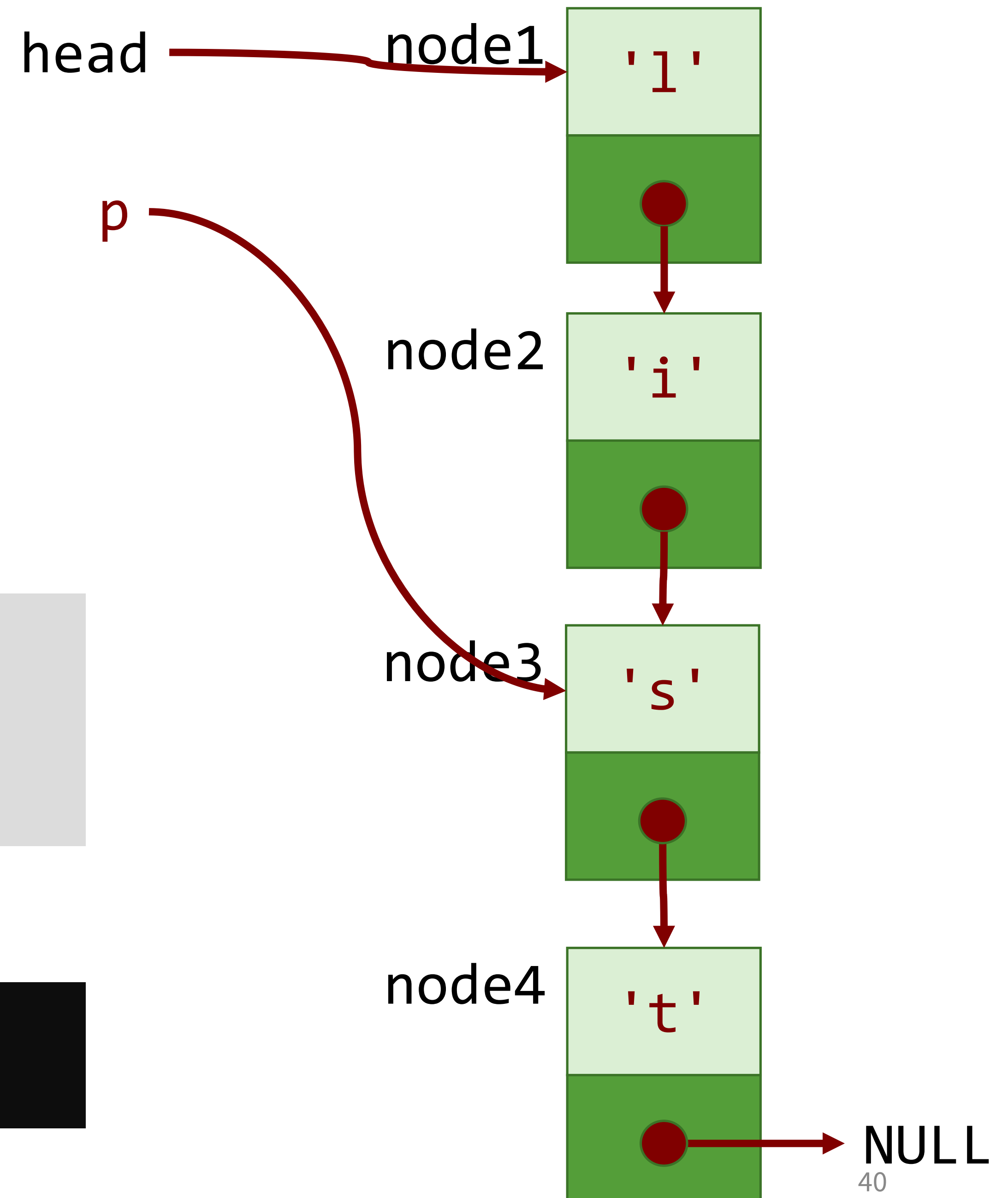
- Will always begin from head
- Visit every node until last node

- Example:

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

- Output:

```
lis
```



Singly-Linked List Traversal (11)

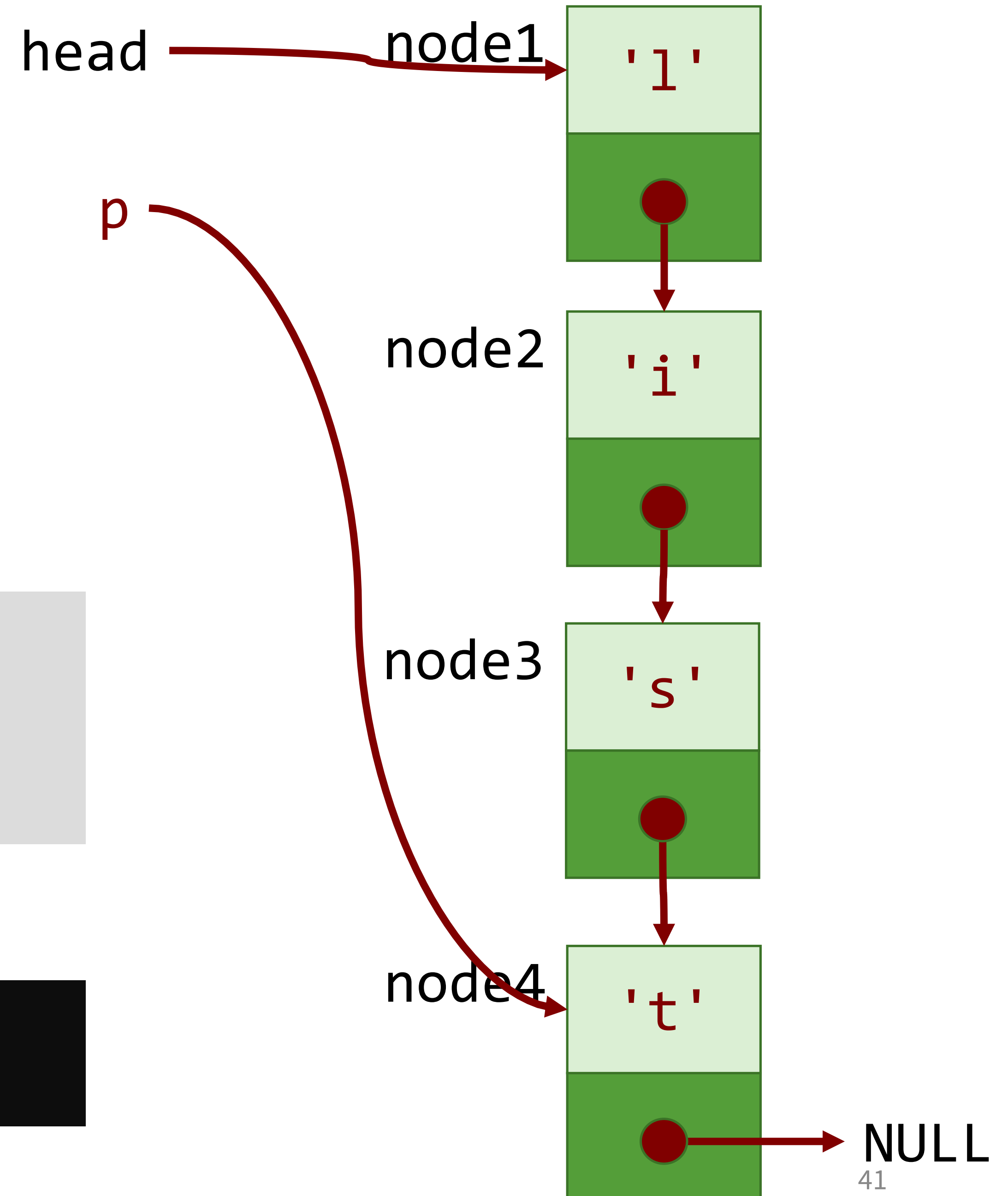
- Will always begin from head
- Visit every node until last node

- Example:

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

- Output:

```
lis
```



Singly-Linked List Traversal (12)

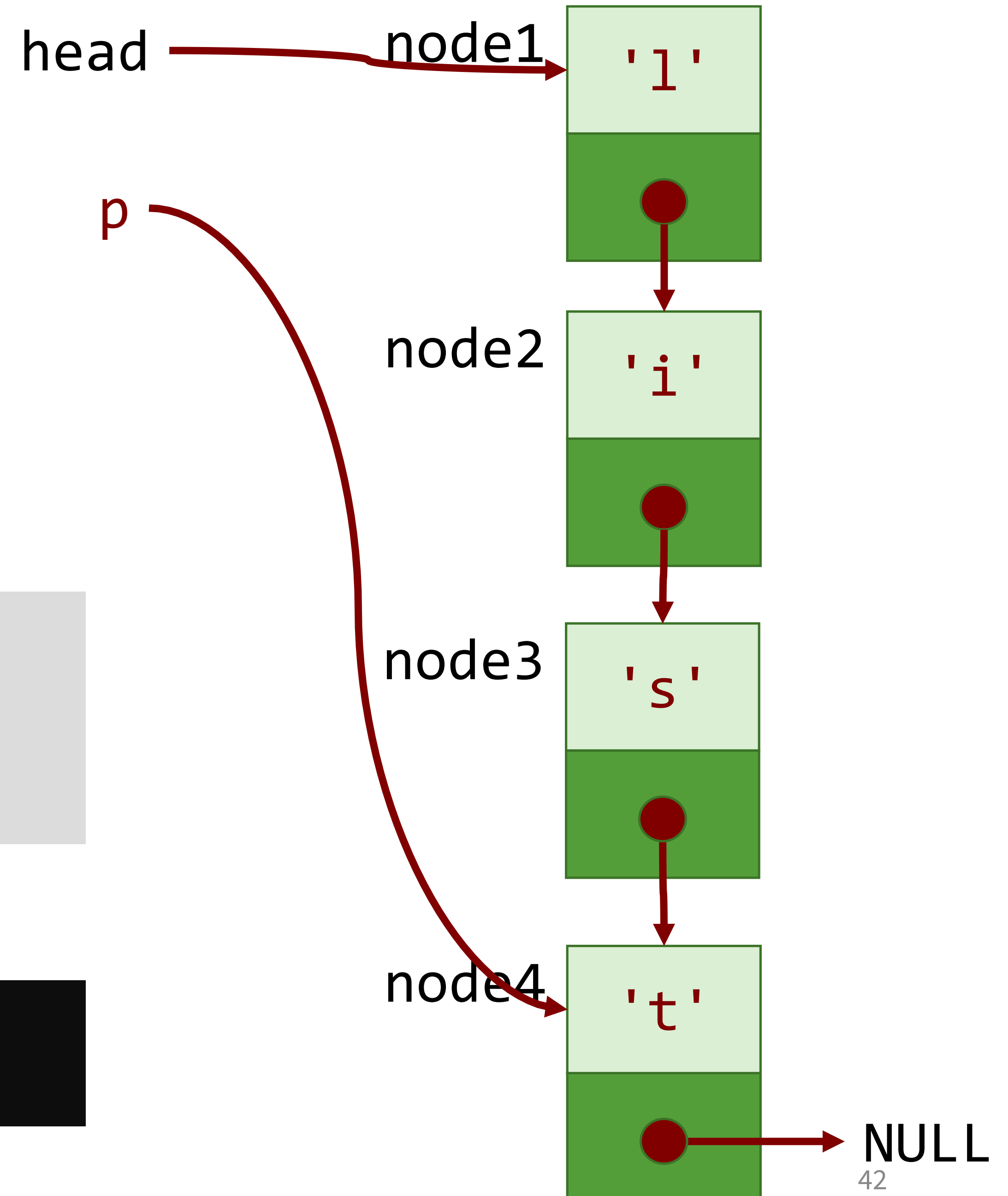
- Will always begin from head
- Visit every node until last node

- Example:

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

- Output:

```
lis
```



Singly-Linked List Traversal (13)

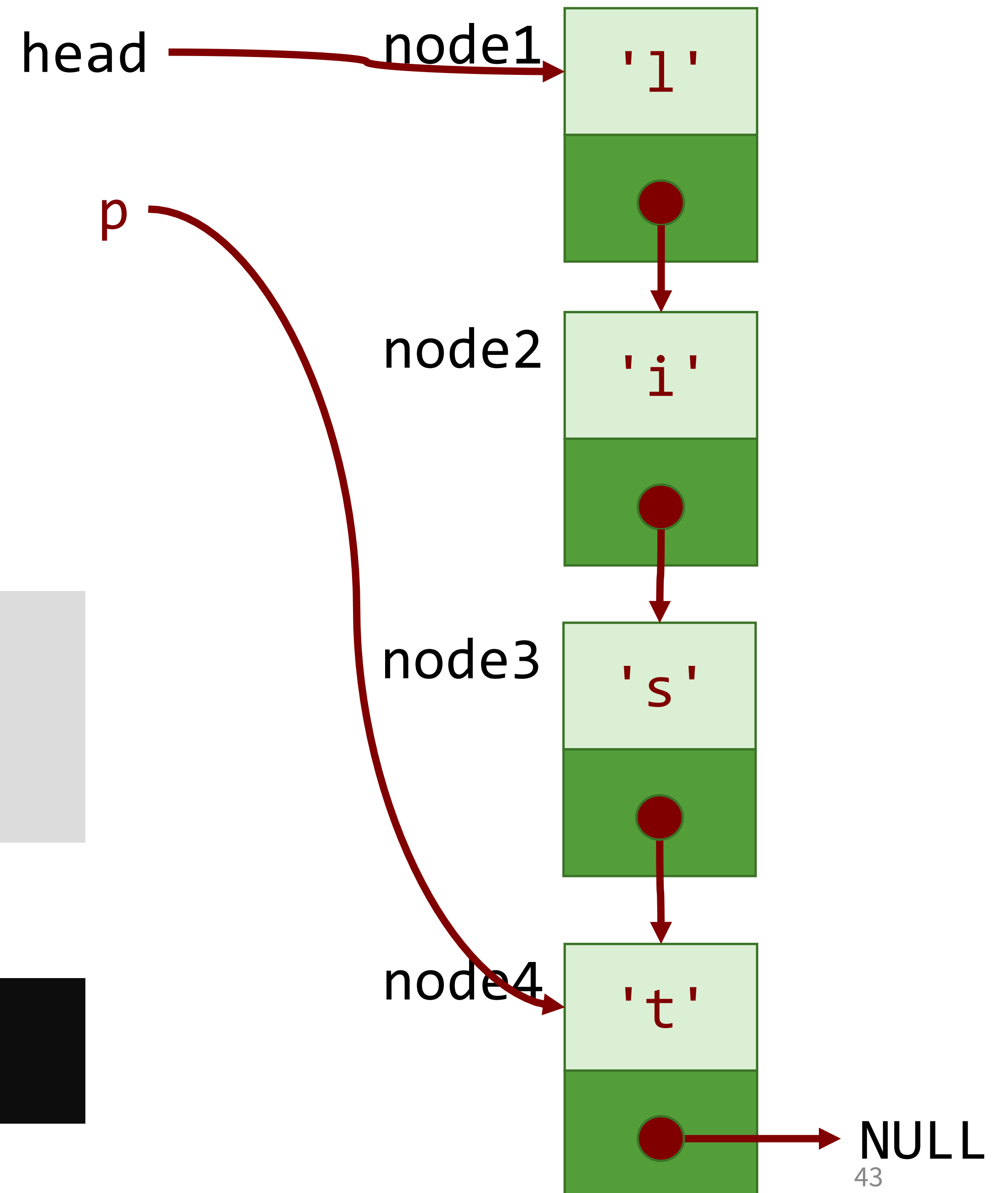
- Will always begin from head
- Visit every node until last node

- Example:

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

- Output:

```
list
```



Singly-Linked List Traversal (14)

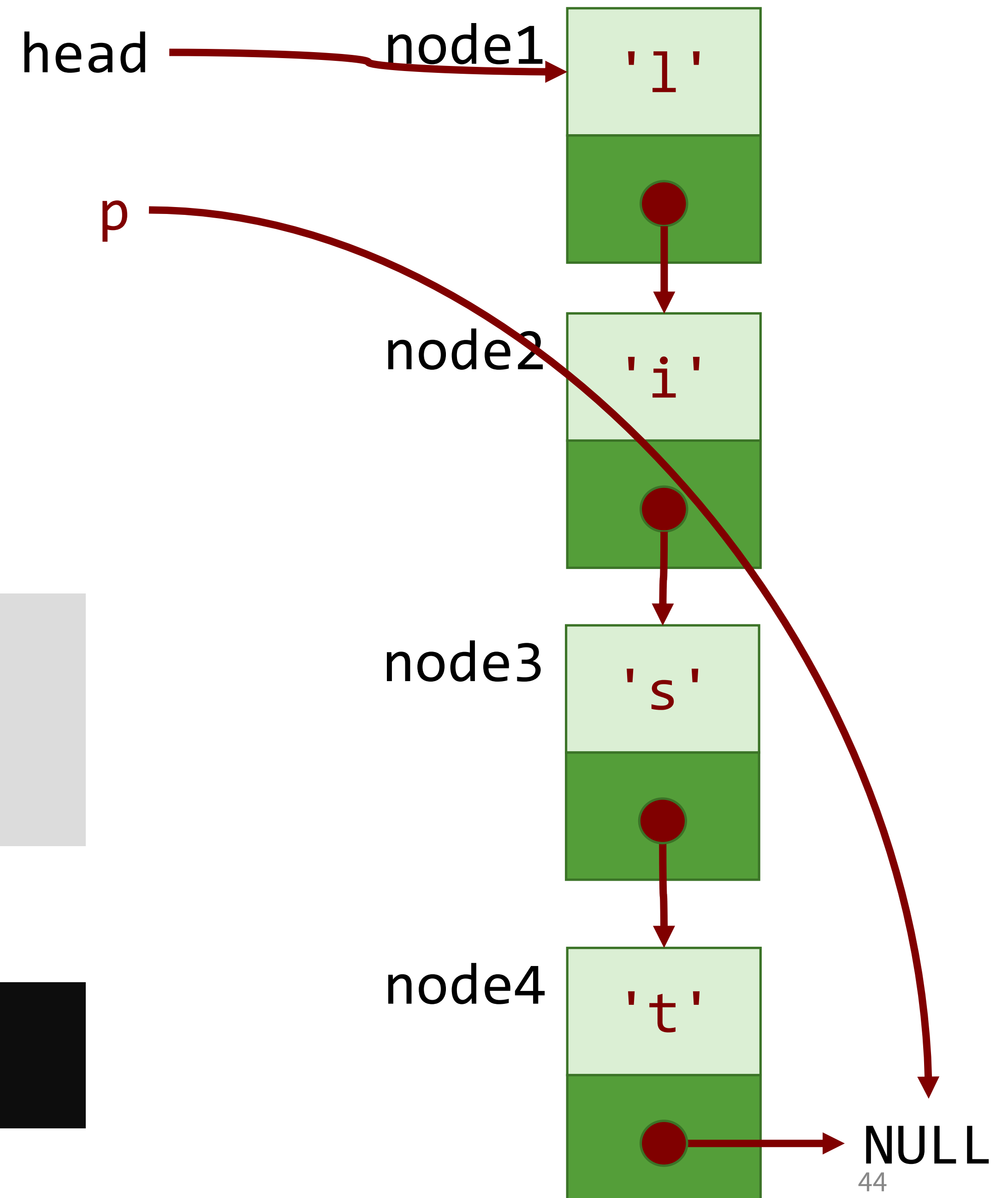
- Will always begin from head
- Visit every node until last node

- Example:

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

- Output:

```
list
```



Singly-Linked List Traversal (15)

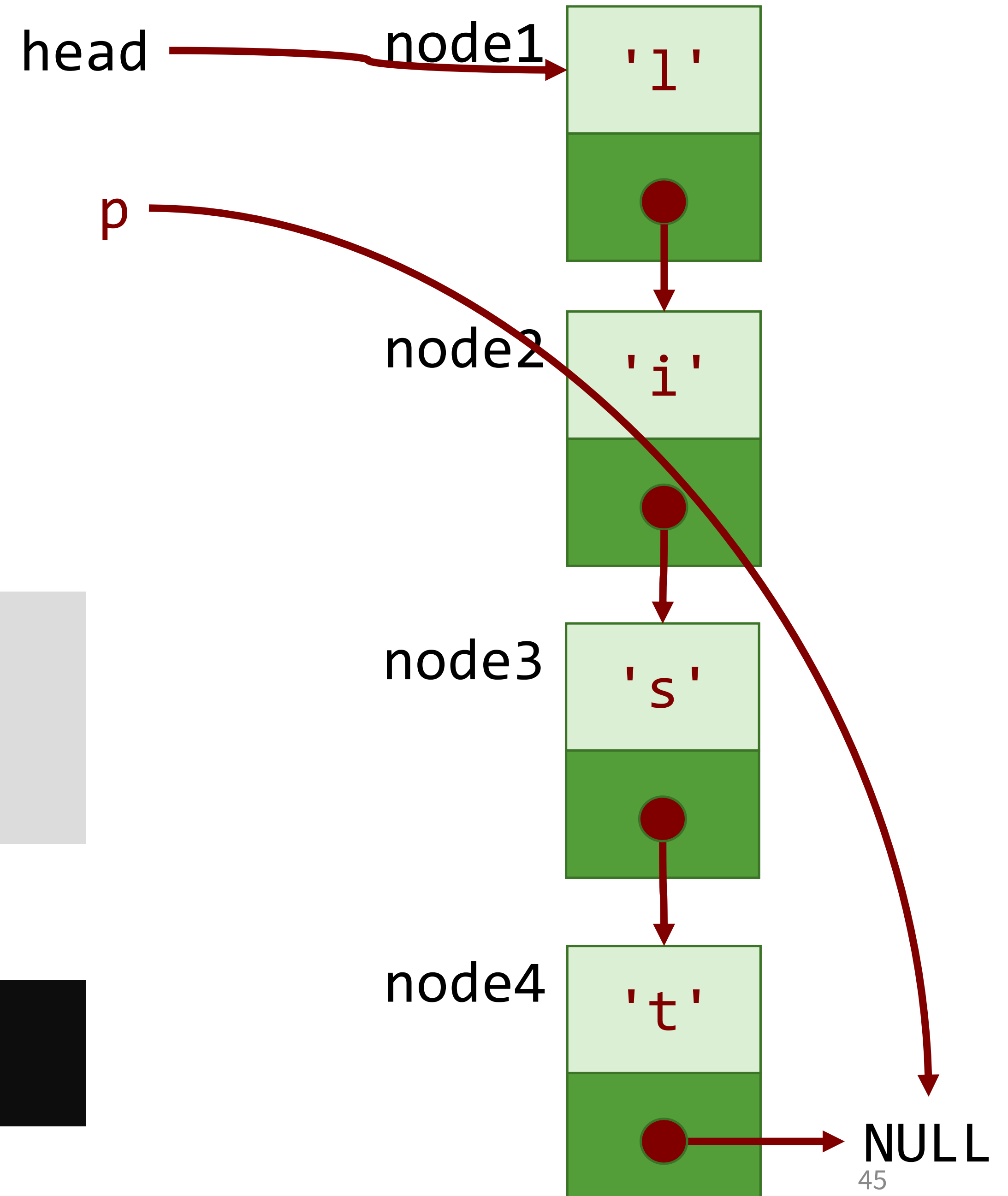
- Will always begin from head
- Visit every node until last node

- Example:

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

- Output:

```
list
```



Next Lecture

- More on Linked Lists