C Programming Language:
C ADTs: Lists and Linked Data

Math 230

Assembly Language Programming
(Computer Organization)

Tuesday Feb 5, 2008

L07

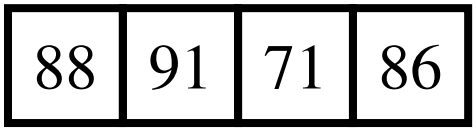
Overview

- strings and arrays, malloc, free
- struct and union
  - Linked Lists

Linked-List

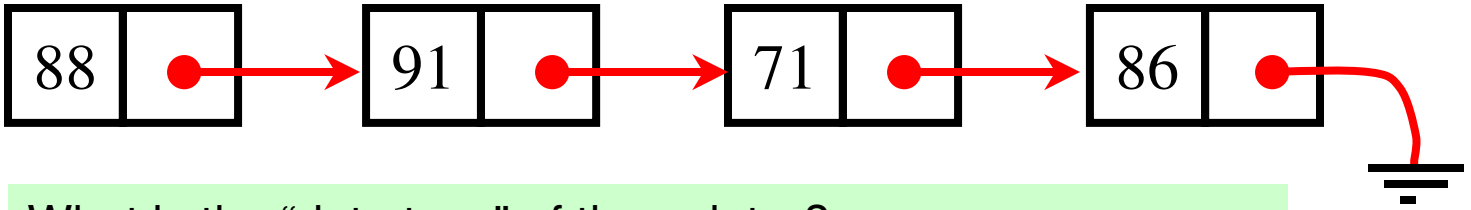
ADT
Abstract Data Type

array of grades:



The data here is held in an array of integers

linked-list of grades:



The data could be held in a "struct" containing an integer and a pointer

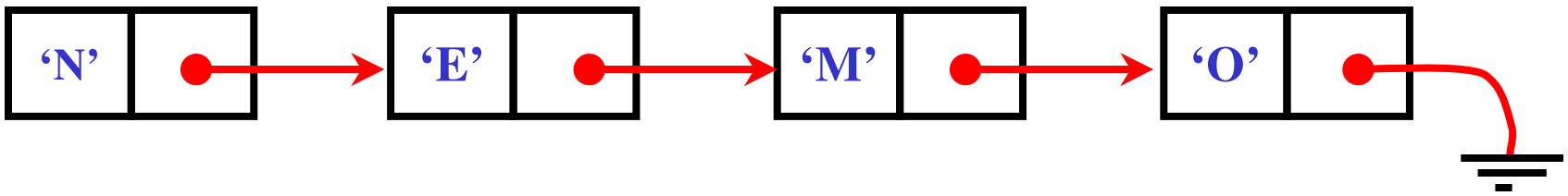
What is the "data type" of the pointer?

The pointer is defined such that it points to another node. The node it points to is the same type of `struct` in which it resides.

Linked-List Definition

- Definition: A linked list is a set of items where each item is a part of a node that also contains a link to a node.

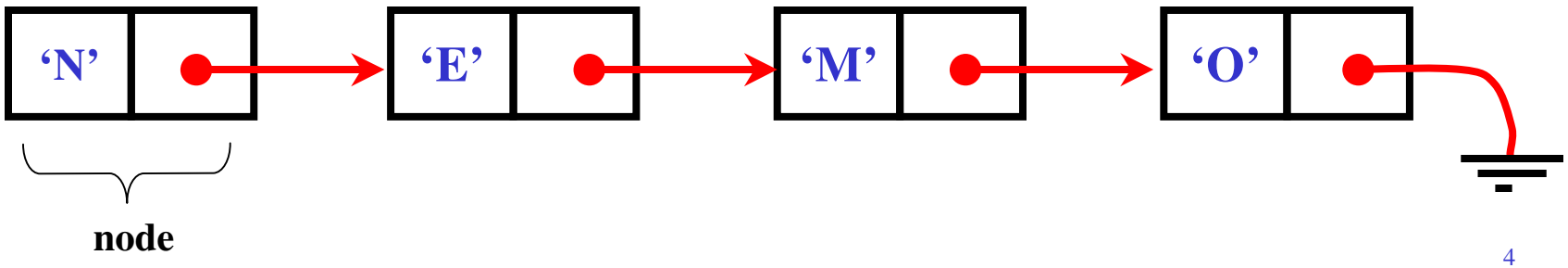
- It is a sequence of "nodes," each containing
  - data field(s) (the **`Item`**)
  - one (or more) links to the next node; a "reference"



Linked-List Definition

- Definition: A linked list is a set of items where each item is a part of a node that also contains a link to a node.

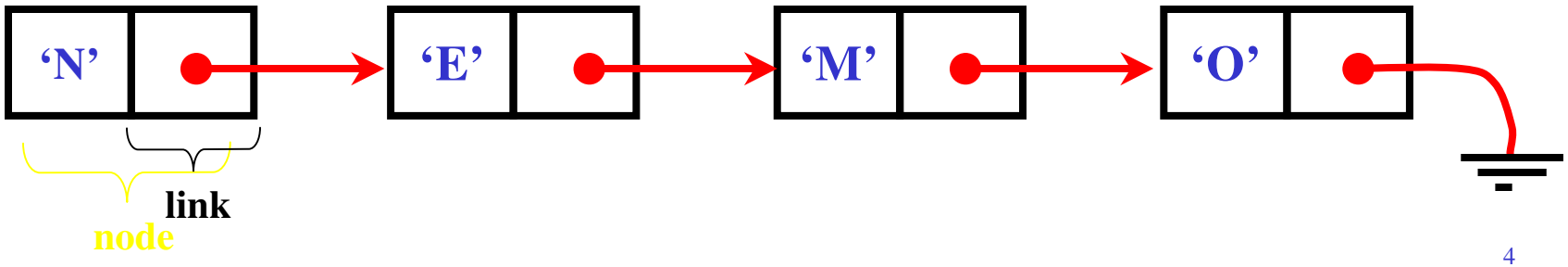
- It is a sequence of "nodes," each containing
  - data field(s) (the **`Item`**)
  - one (or more) links to the next node; a "reference"



Linked-List Definition

- Definition: A linked list is a set of items where each item is a part of a node that also contains a link to a node.

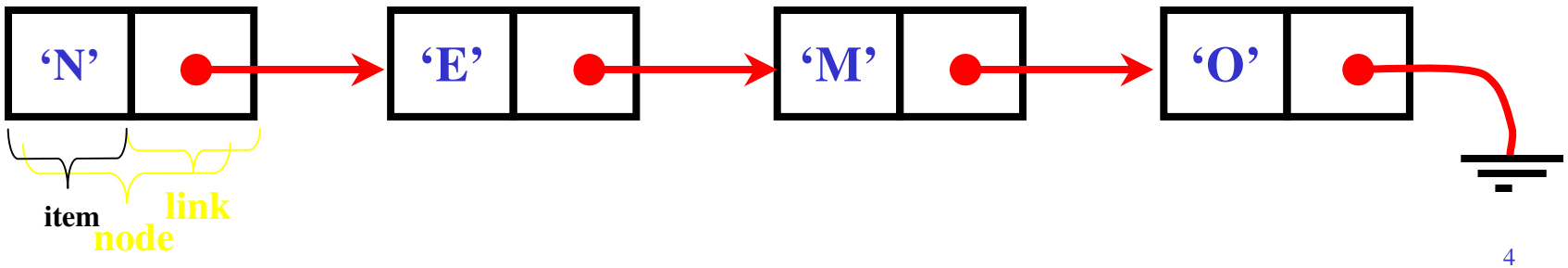
- It is a sequence of "nodes," each containing
  - data field(s) (the **`Item`**)
  - one (or more) links to the next node; a "reference"



Linked-List Definition

- Definition: A linked list is a set of items where each item is a part of a node that also contains a link to a node.

- It is a sequence of "nodes," each containing
  - data field(s) (the **`Item`**)
  - one (or more) links to the next node; a "reference"

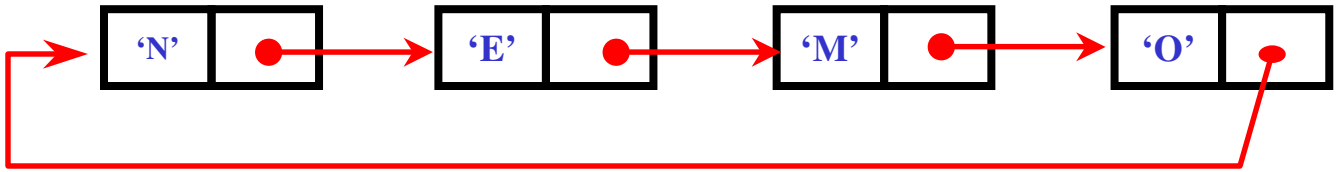


Linked Lists

- The link in the *final* node can be represented a number of ways
  - As a null link that points to no node

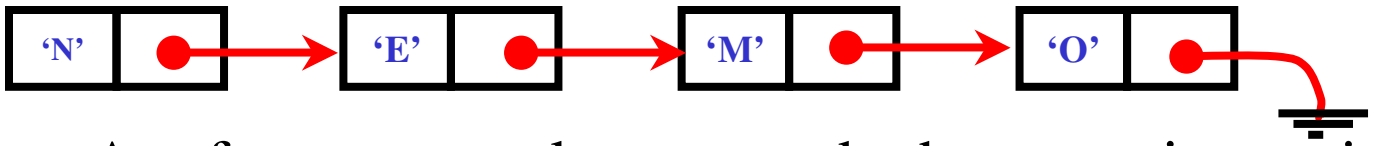
  - A reference to a dummy node that contains no item

  - A reference back to the first node, making the list a **circular list**.

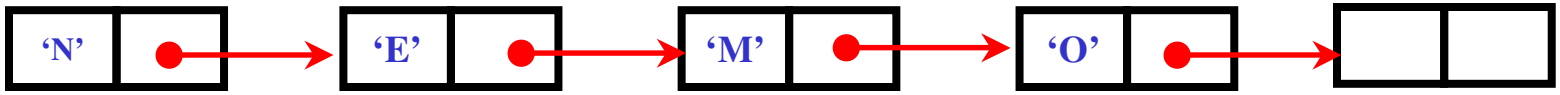


Linked Lists

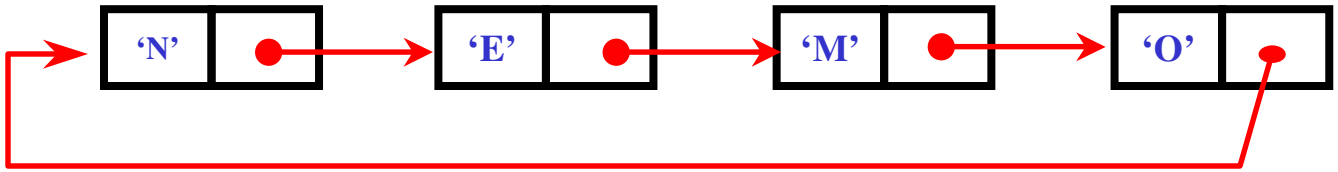
- The link in the *final* node can be represented a number of ways
  - As a null link that points to no node



  - A reference to a dummy node that contains no item



  - A reference back to the first node, making the list a **circular list**.

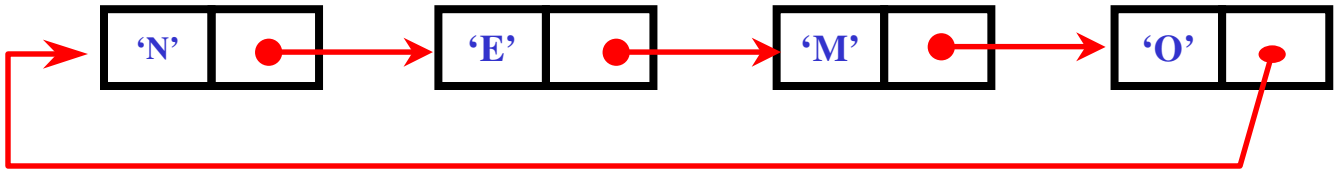


Linked Lists

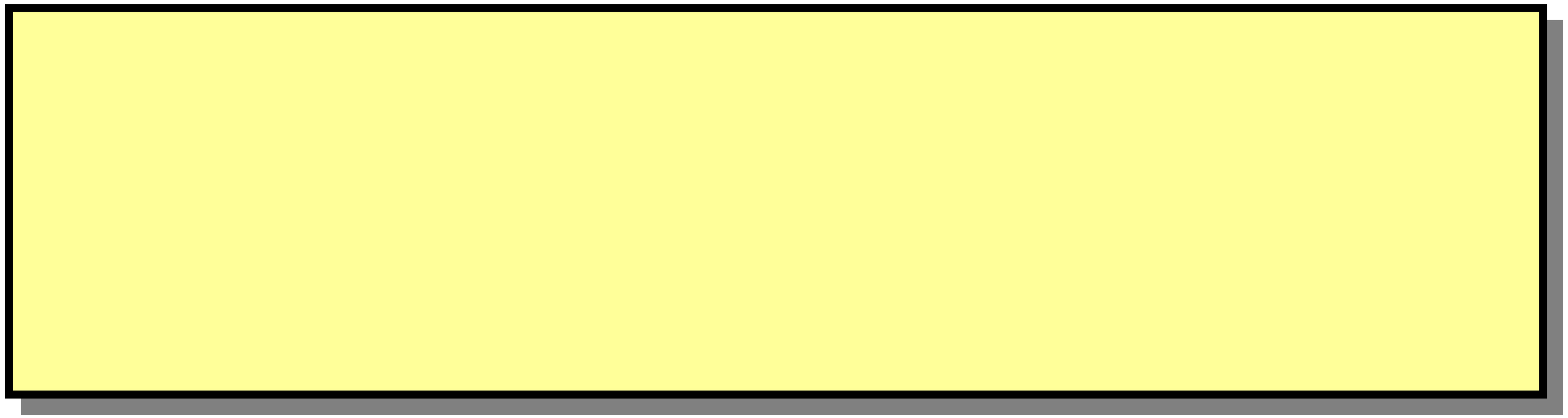
- The link in the *final* node can be represented a number of ways
  - As a null link that points to no node

  - A reference to a dummy node that contains no item

  - A reference back to the first node, making the list a **circular list**.



Linked-List Code



Linked-List Code

```
struct node {

}
```

Linked-List Code

```
struct node {
      int item;
```

Linked-List Code

```
struct node {
      int item;
      struct node* next;
```

Linked-List Code

```
struct node {
      int item;
      struct node* next;
};
```

Linked-List Code

```
struct node {
      int item;
      struct node* next;
};
```

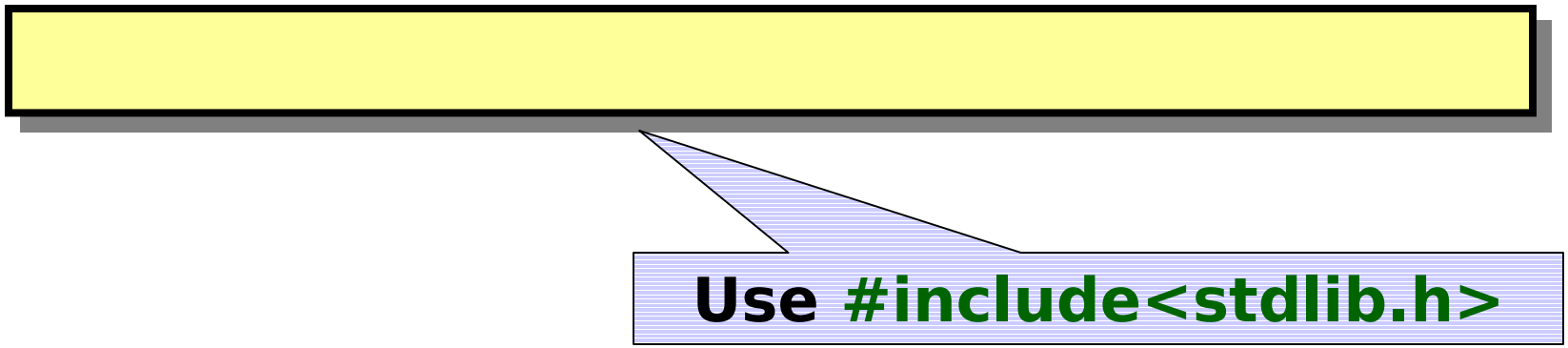
```
typedef  int  Item; //or char, float,char*,etc
typedef struct node* link;
struct node {
      Item item;
      link next;
};
```

The Utility of Lists

- Linked lists help us manage constantly changing lists of data
- They help us with the *insertion* and *deletion* of existing records
- Each item contains information on how to get to the next item
- It is a set of items where each item is part of a node that also contains a link to a node

`malloc()` and Linked-Lists

- We've created one "node" data type
- We'll have many instances of this one type
  - Recall: we can have also have muliple instances of int
- To create a new instance of a node and reserve memory for it:



`malloc()` and Linked-Lists

- We've created one "node" data type
- We'll have many instances of this one type
  - Recall: we can have also have muliple instances of int
- To create a new instance of a node and reserve memory for it:

```
link x = malloc(sizeof *x );
```

Use #include<stdlib.h>

`free()` – letting go of previously allocated memory

- When you're done with the node, you should free the previously allocated memory using "`free()`"
- The interface:

- Its usage

- Helps you avoid memory leaks. Continuously allocating memory without freeing any could lead to a "crash."
  - memory leaks!

`free()` – letting go of previously allocated memory

- When you're done with the node, you should free the previously allocated memory using "`free()`"
- The interface:

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

- Its usage

```

```

- Helps you avoid memory leaks. Continuously allocating memory without freeing any could lead to a "crash."
  - memory leaks!

`free()` – letting go of previously allocated memory

- When you're done with the node, you should free the previously allocated memory using "`free()`"
- The interface:

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

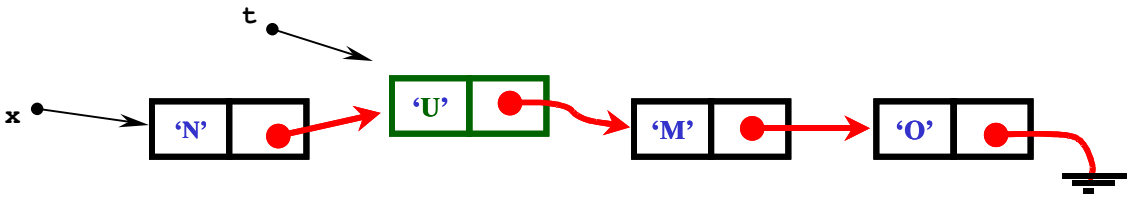
- Its usage

```
free(x);
```

- Helps you avoid memory leaks. Continuously allocating memory without freeing any could lead to a "crash."
  - memory leaks!

Accessing list-node information

- Dereference the pointer, then use the structure member names
- The "item" in the node referenced by link x is
  `(*x).item` or **`x->item`**
  - (the data type of item is **Item**)
- The link to the next node is indicated by
  `(*x).next`, or **`x->next`**
  - (the data type of next is **link**)



Fundamental Operations on Linked Lists

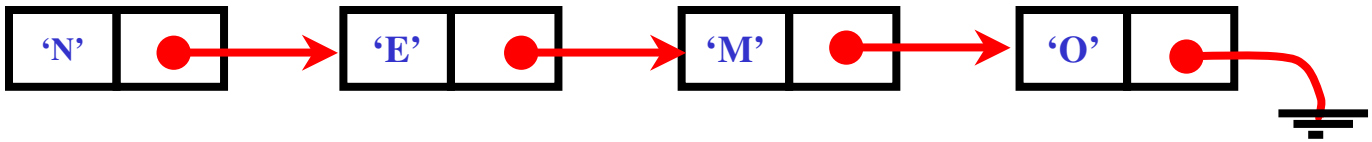
- Linked-list Deletion
- Linked-list Insertion

Linked-list Deletion

- The node containing ‘E’ can be removed by redirecting the node that points to it:

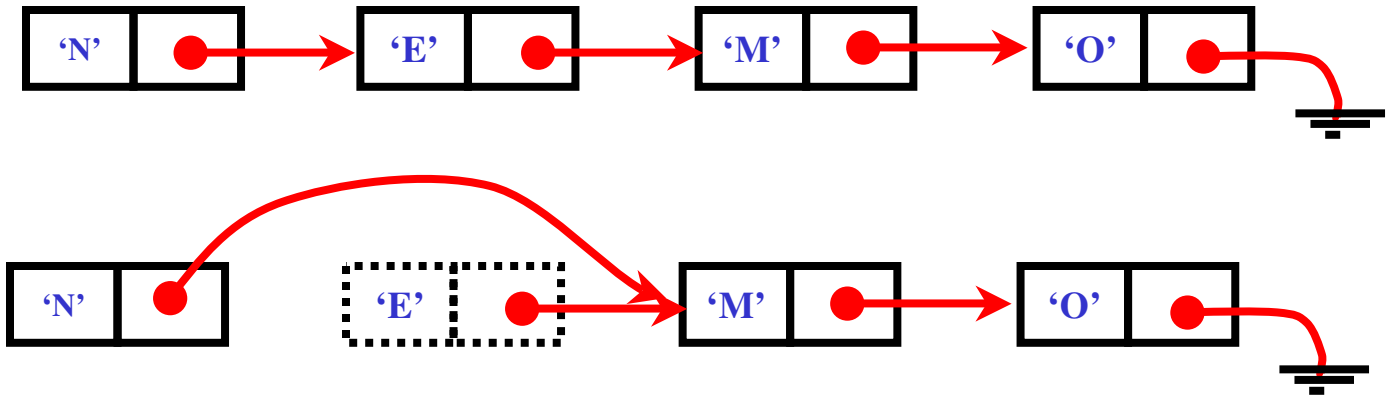
Linked-list Deletion

- The node containing ‘E’ can be removed by redirecting the node that points to it:



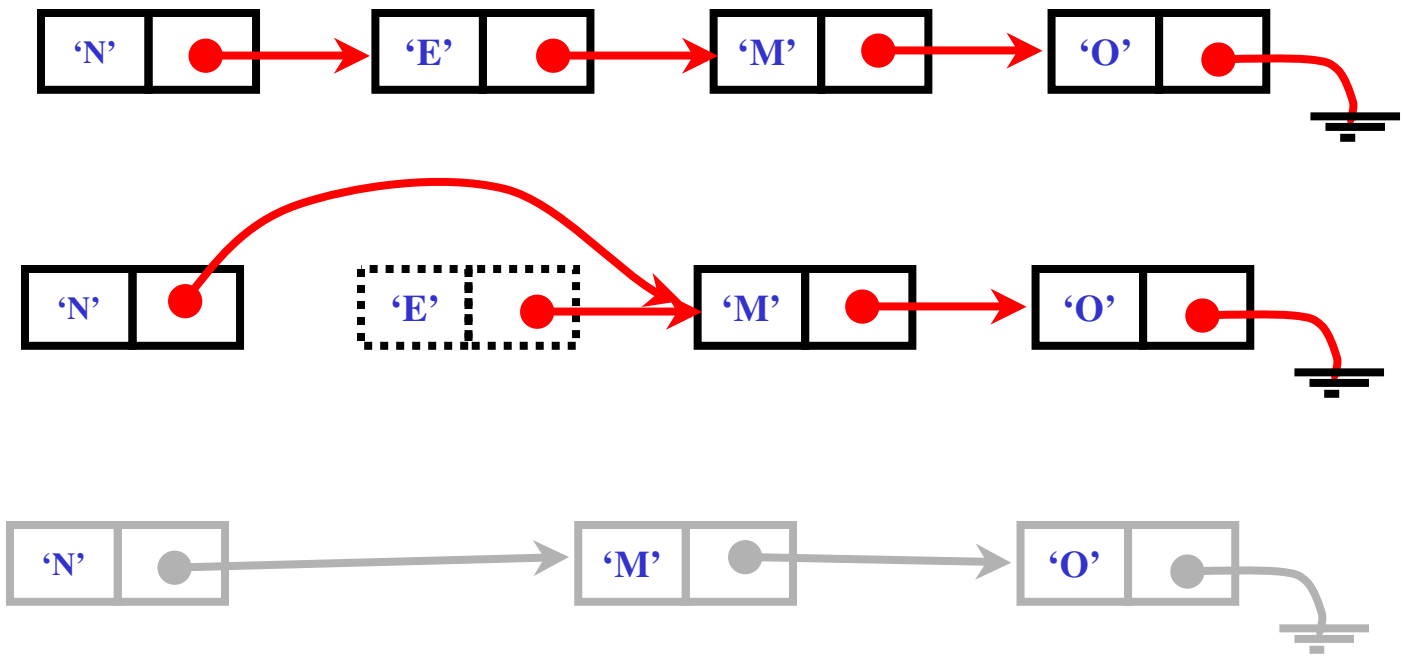
Linked-list Deletion

- The node containing ‘E’ can be removed by redirecting the node that points to it:



Linked-list Deletion

- The node containing ‘E’ can be removed by redirecting the node that points to it:

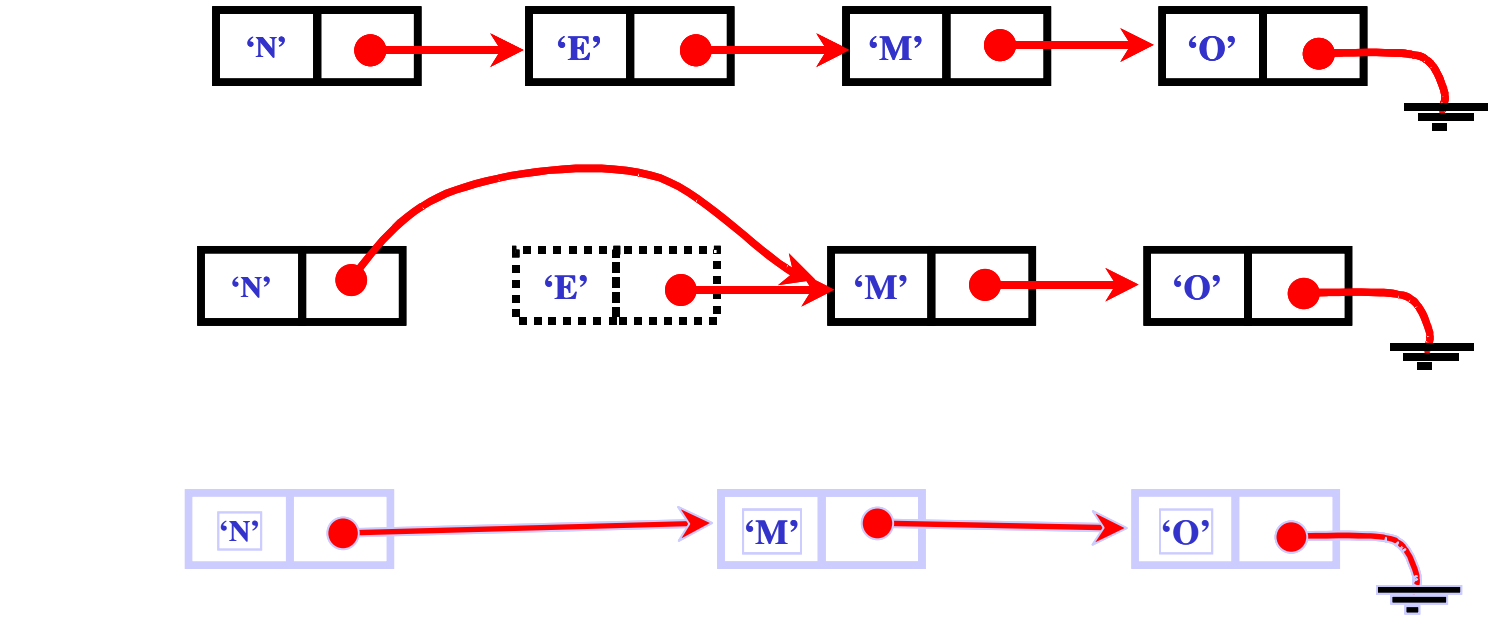


Linked-list Deletion

- To "delete" the node containing 'E' (the 2 below are equiv)

```
t = x->next;
x->next = t->next;
```

```
x->next = x->next->next;
```

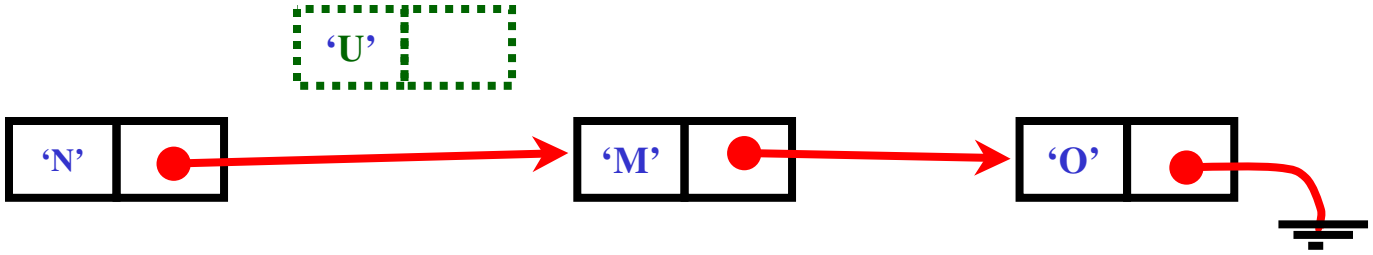


Linked-list Insertion

- The node containing ‘U’ can be inserted into the list

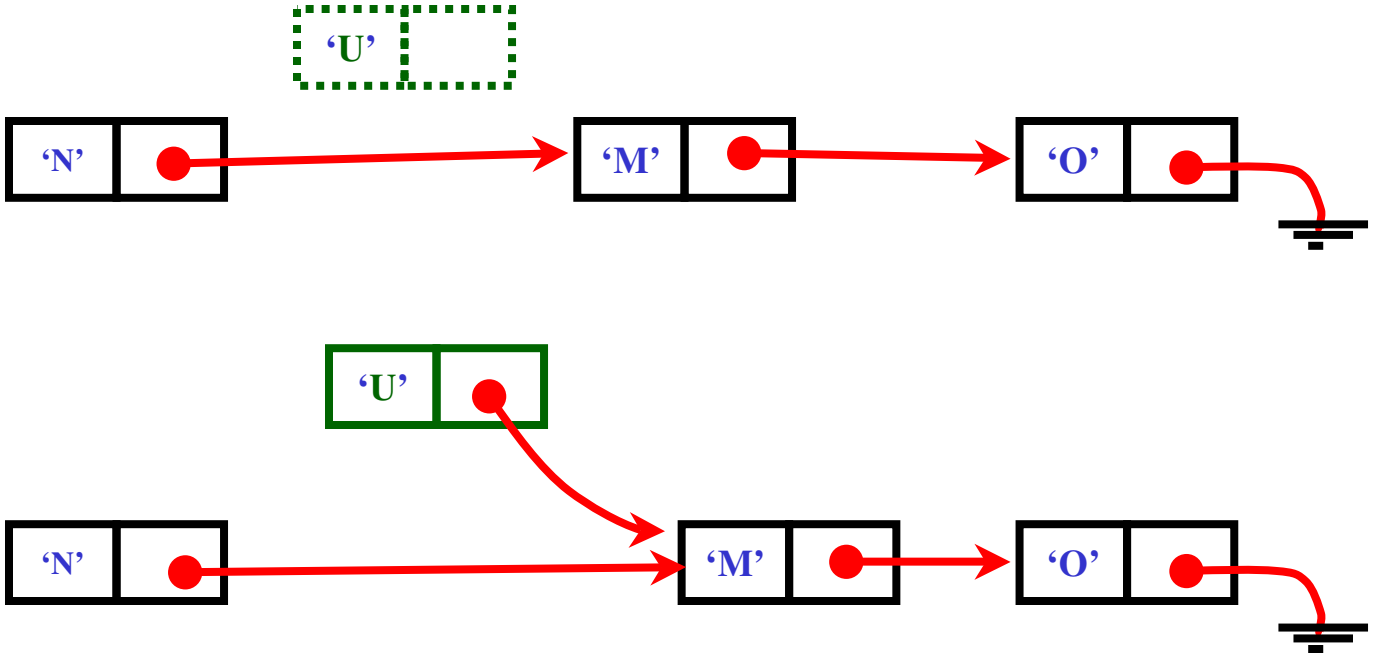
Linked-list Insertion

- The node containing ‘U’ can be inserted into the list



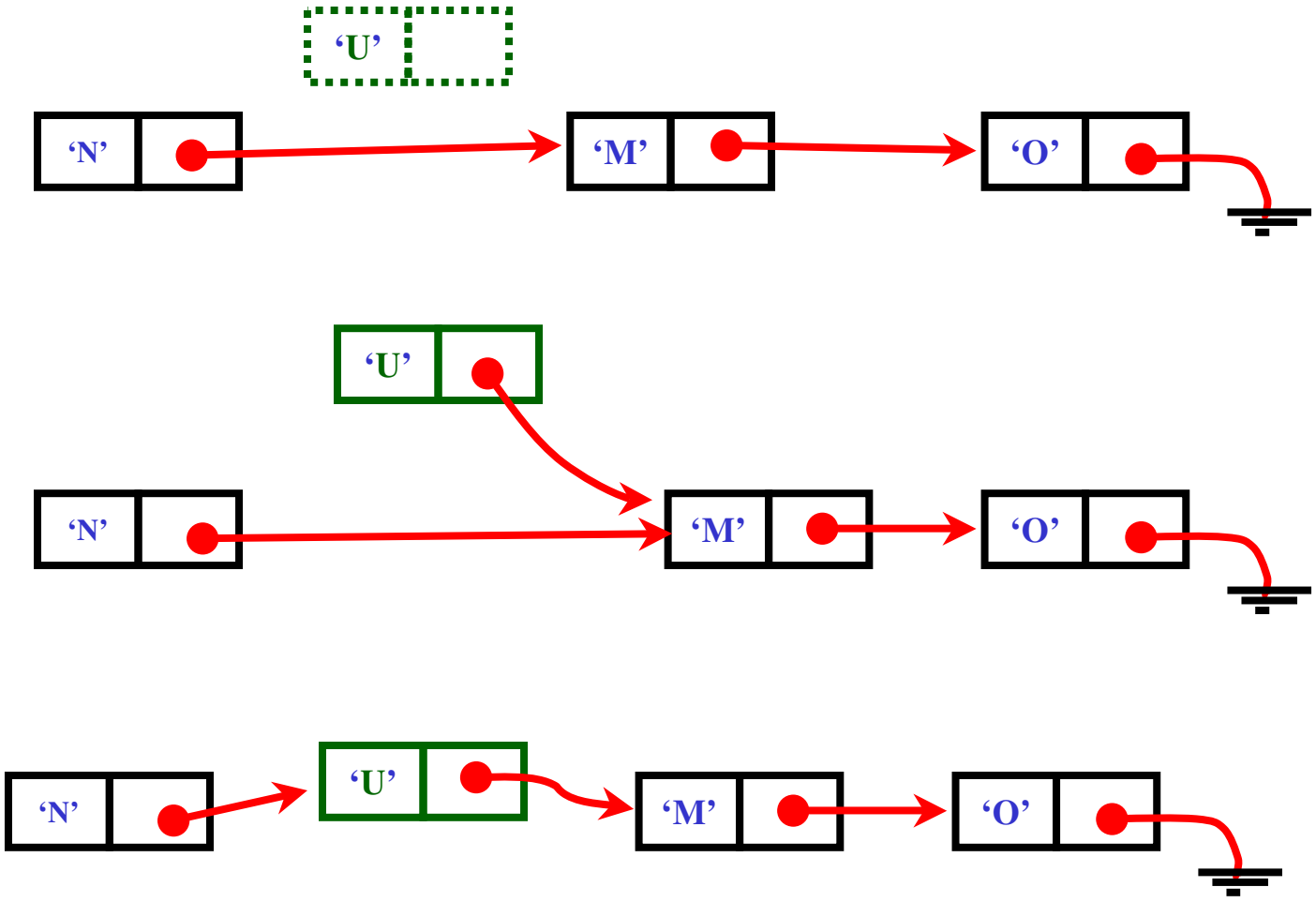
Linked-list Insertion

- The node containing ‘U’ can be inserted into the list



Linked-list Insertion

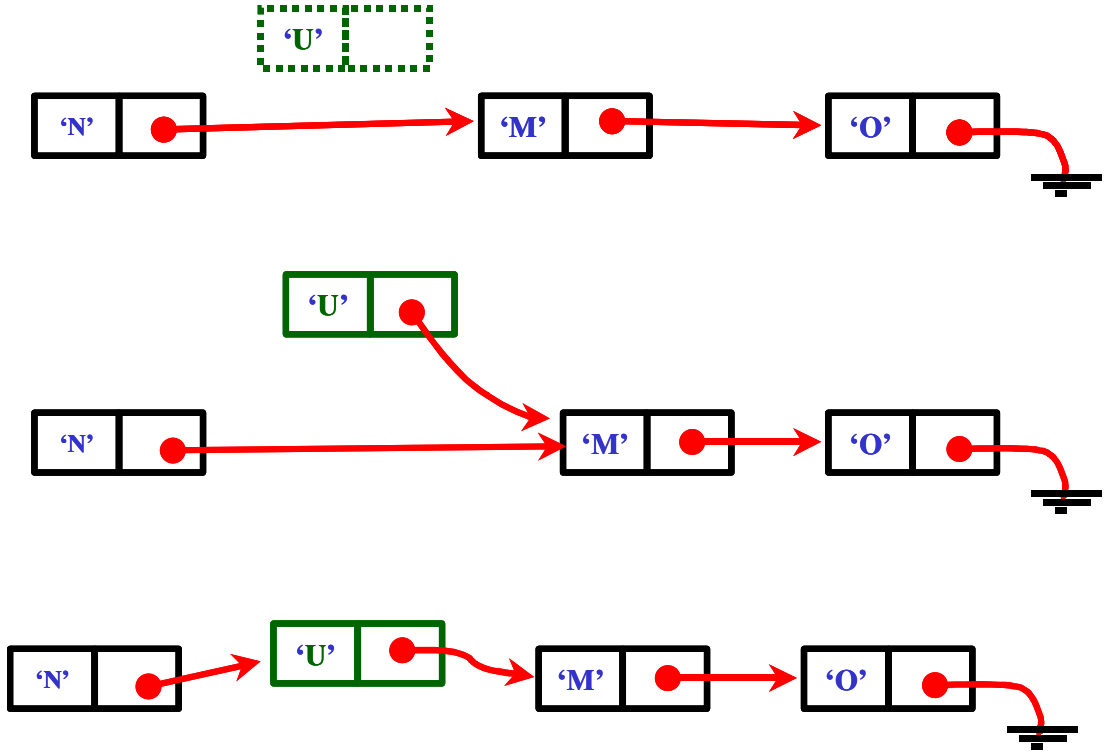
- The node containing ‘U’ can be inserted into the list



Linked-list Insertion

- To insert node containing ‘U’ into a list

```
t->next = x->next;
x->next = t;
```



Array vs Linked-List

- Given an array of fixed size, inserting an element into the n^{th} positon is tricky

- Access into a linked-list is not as efficient as an array when finding the k^{th} item simply using a[k]

- Linked lists are an alternative to arrays

NULL - End of List

- The NULL pointer is frequently used as a sentinel to mark the end-of-string NULL.
  - both serve similar purposes

Summary

- Linked lists
  - help us manage constantly changing lists of data
  - help us with insertion and deletion of existing records
  - is a basic data structure where each item contains information on how to get to the next item
  - is a set of items where each item is part of a node that also contains a link to a node