Data Structures

Data Structures

- Building blocks of a solution
- The most important factor in designing a solution
- A smart selection of data structures wins!

Structure in C (review)

- Array: Collection of Same Types of Data
- Structure: Collection of Different Types of Data

```
struct {
    char   name[20];
    int    age;
    float  salary;
    char   hobby[3][20];
}  employee;

/*  name, age, salary, hobby:  members      */
/*  employee:   variable of type  struct { } */
```

Structure tag name

```
struct  EMPRECORD {
    char  name[20];
    int    age;
    float  salary;
    char  hobby[3][20];
} employee, former_employee;

/*  EMPRECORD:  tag name for { }        */

/*  struct  EMPRECORD
            employee, former_employee;  */
```

Member Data Types

- Primitive Types
  - int, float, double, char
  - pointer
- Array
- Structure
  - other struct
  - defining struct

Structure Member Access

```
struct {
    char   name[20];
    int    age;
    float  salary;
    char   hobby[3][20];
} employee;
```

```
struct_variable.member_name
(&struct_variable)->member_name
```

```
/*  employee.name          */
/*  employee.hobby[3][15]  */
```

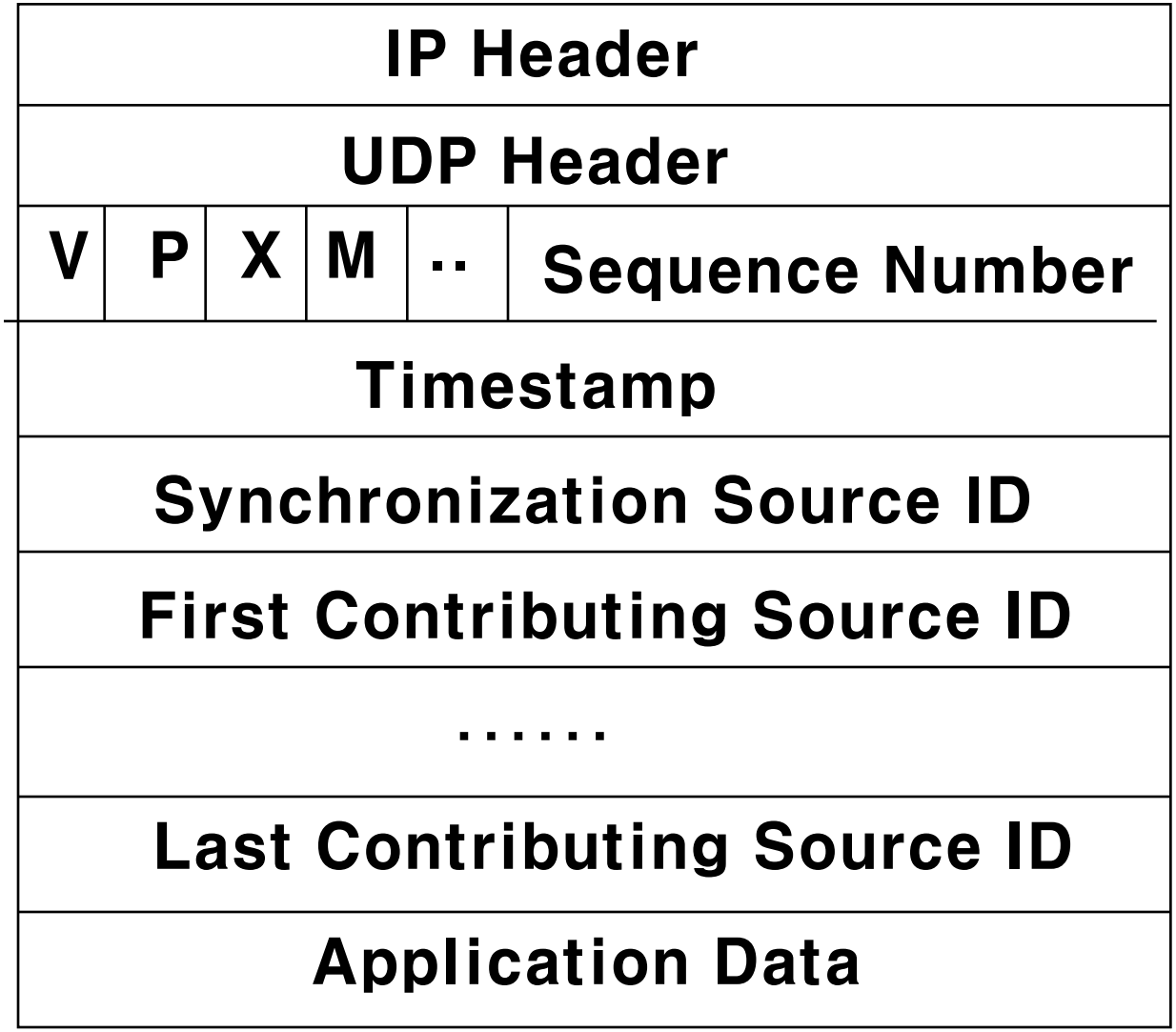
Struct Member Initialization

```
struct {
    char   name[20];
    int    age;
    float  salary;
    char   hobby[3][20];
} employee;

employee.name[] = "Neil Diamond";
employee hobby[3][] ="tennis and walking";

employee = {"hong gildong", 25, 35000.0, "jump"};
```

Member Name Scope

```
struct  EMPRECORD {
    char  name[20];
    int    age;
    float  salary;
    char  hobby[3][20];
} employee, former_employee;

struct  PERSON {
    char  name[20];
    int    age;
    char  address[30]'
};

/*  unique only within a single structure      */
```

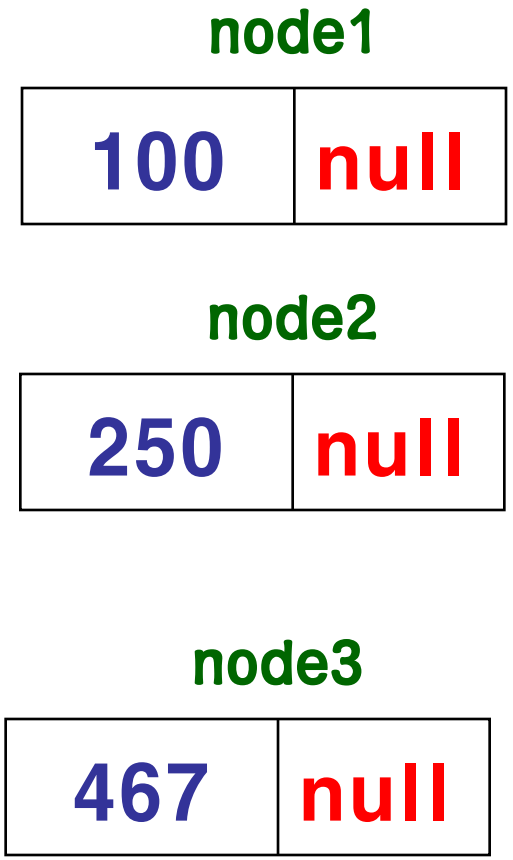
Structure: Use Example – IP Packet



Real-Time Packet Transfer Protocol Message Format

Pointer and Self-Referential Structure

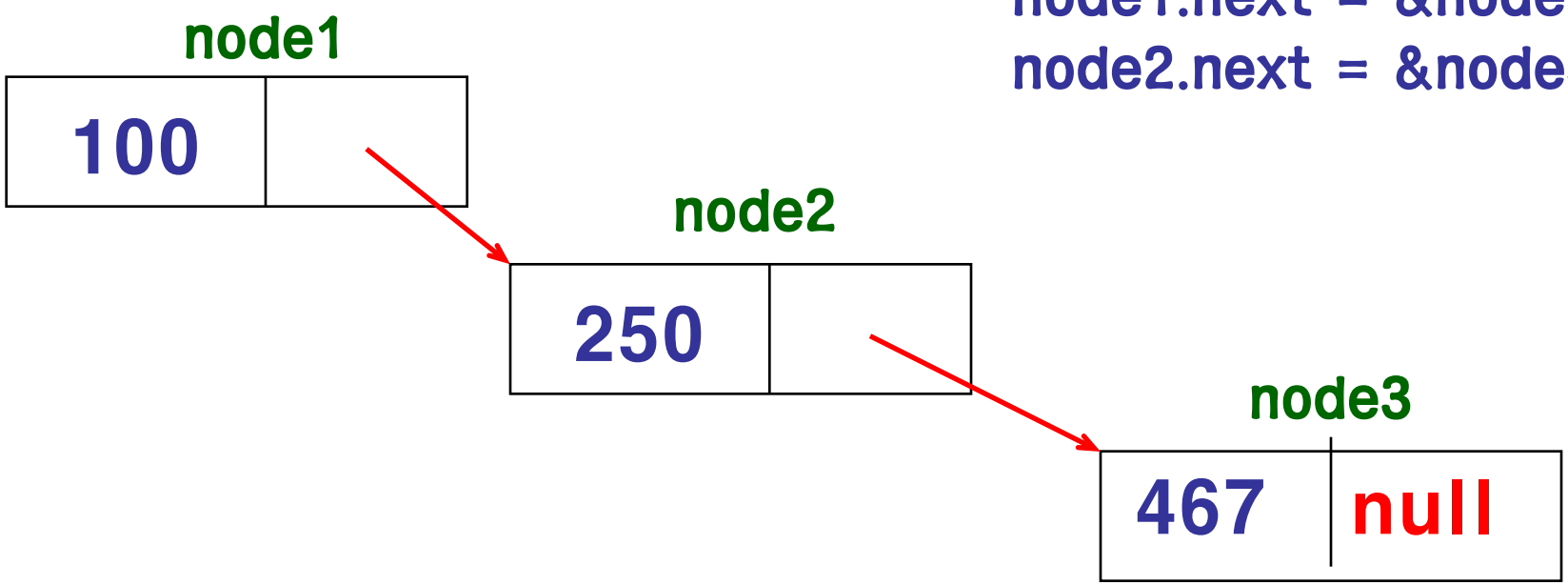
```
struct  NODE {
    int                 key;
    struct NODE     *next;
} node  node1, node2, node3;

node1.key = 100;
node2.key = 250;
node3.key = 467;
node1.next = node2.next = node3.next =
NULL;
;
```



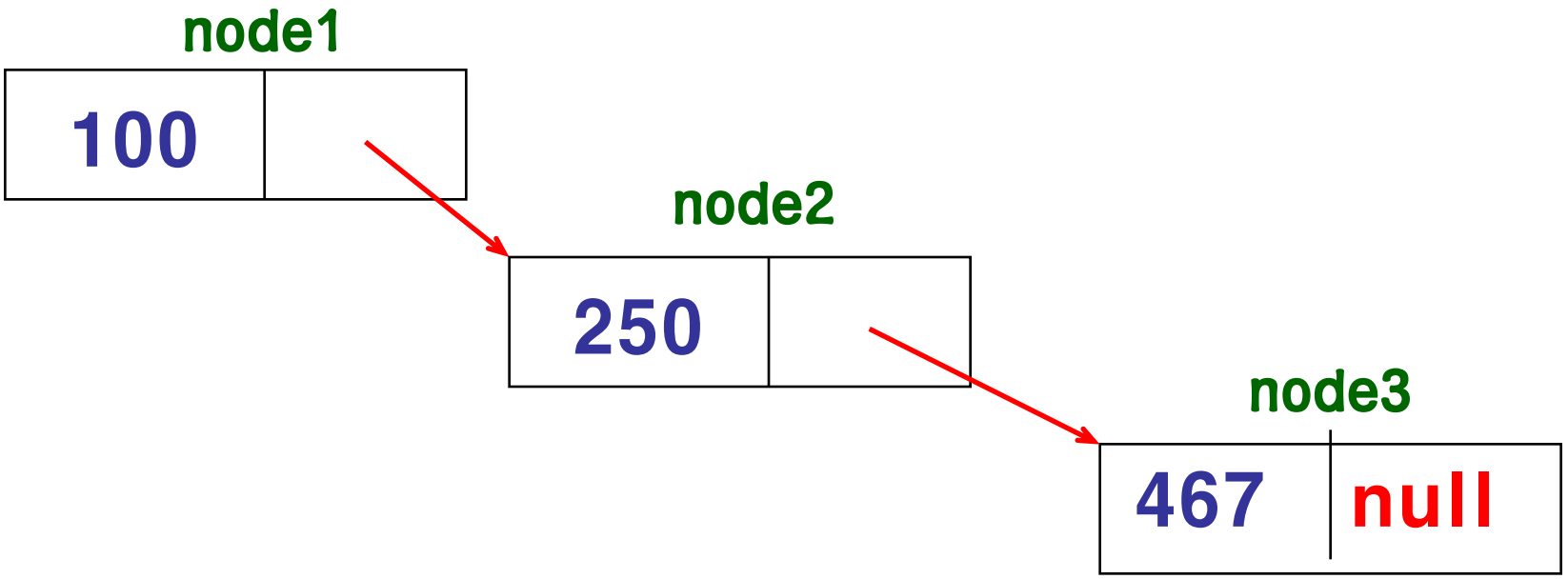
Example (cont'd)

```
struct  NODE {
    int                 key;
    struct NODE     *next;
} node  node1, node2, node3;
```

```
node1.next = &node2;
node2.next = &node3;
```



Member Access via a pointer



node1.next -> key 250

node1.next -> next -> key 467

Linked Lists (vs. array)

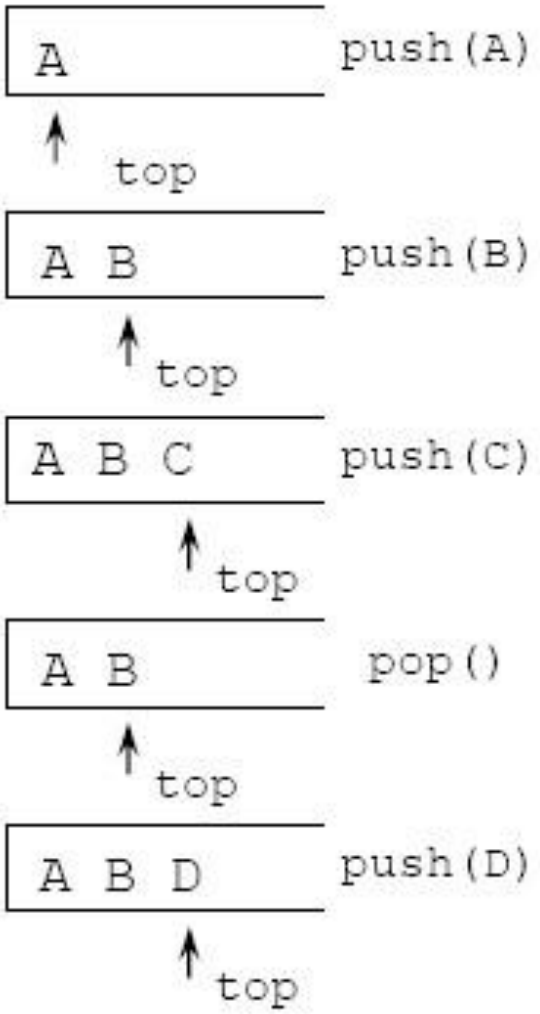
- You just saw them
- pros
  - size of the list can grow/shrink easily
  - easy to insert
  - easy to delete
  - easy to lookup
- cons ?

Stack and Queue

- List
  - insert/delete an element at any location
- Queue
  - insert at the head
  - delete at the tail
  - First-In-First-Out (FIFO)
- Stack
  - only one position to insert and delete
  - Last-In-First-Out (LIFO)

Stack

- Operations
  - push(x, S) inserts x onto a stack S
  - pop(S) deletes the most new elements from S, and returns it to the callee
  - initialize(S) creates a new empty stack
  - Full(S), Empty(S) check if the stack S is full/empty



Stack Implementation

- data structures
  - array – S
  - pointer – top

```
#define MAX_STACK_SIZE 100
struct stack{
        int data[MAX_STACK_SIZE];
        int top = -1;
} s1, s2, ...;
```

- full/empty

```
int empty(stack s)
{
   if (s.top < 0 )
        return (1);
   else
        return (0);
}
```

```
int full(stack s)
{
   if (s.top >= MAX_STACK_SIZE)
        return (1);
   else
        return (0);
}
```

```
int pop(stack s)
{
   if empty(s)
       return stack_empty();
   return s.data[s.top--];
}
```

```
void push(int x, stack s)
{
       if (full(s)) {
              stack_full();
              return;
       }
       s.data[++top] = x;
}
```

List, Queue

- queue is similar to a stack
  - array vs linked list
- but list is a little bit difficult
  - if we know the maximum size, array will do
  - if when we know the size,
    - insert/delete ops will waste memory
  - linked list looks good, but
    - link traverse is expensive
  - then, what is your solution?