Self-Referential Structures

We can define a structure containing
a pointer to the same type of structure:

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

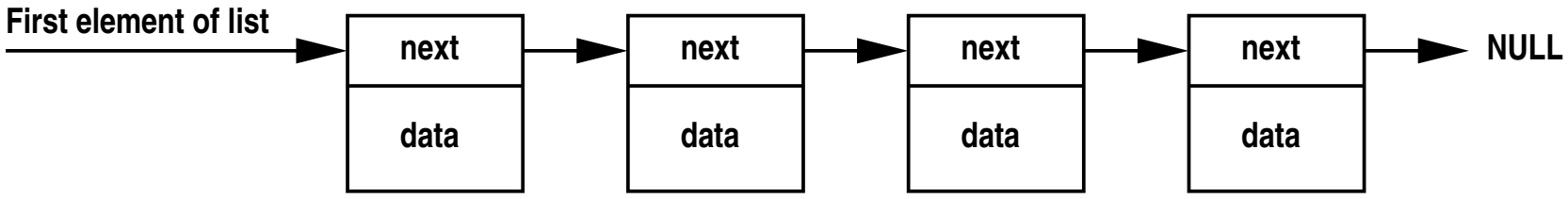
These "self-referential" pointers can be used to build larger "dynamic" data structures out of smaller building blocks.

Linked Lists

The most fundamental of these dynamic data structures is the *Linked List*:

- based on the idea of a sequence of data items or nodes
- linked lists are more flexible than arrays:
  - items don't have to be located next to each other in memory
  - items can easily be rearranged by altering pointers
  - the number of items can change dynamically
  - items can be added or removed in any order

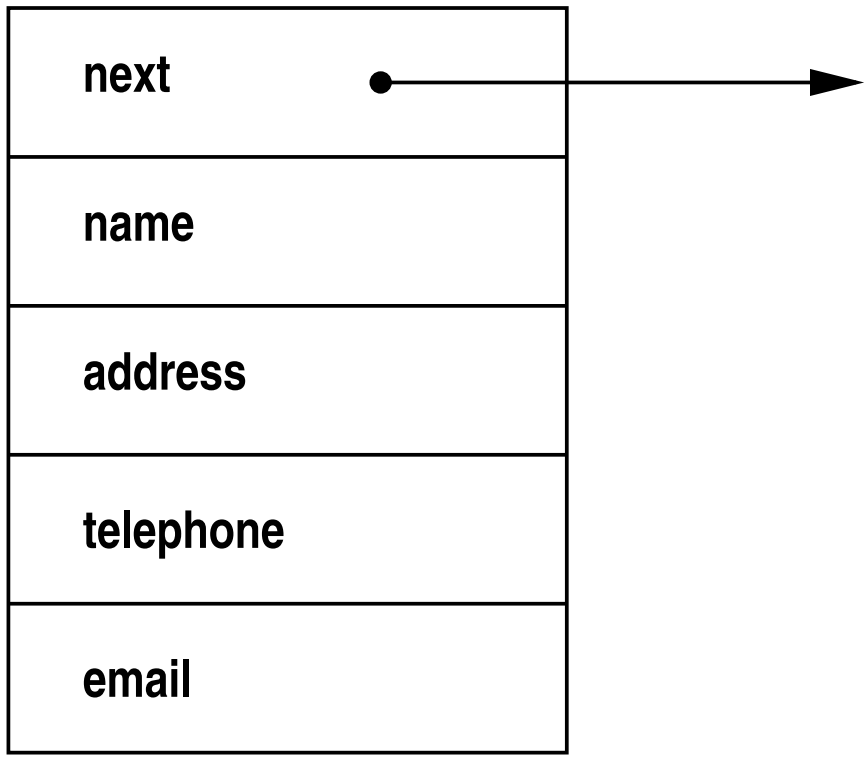
Linked List



- a *linked list* is a sequence of items
- each item contains data and a pointer to the next item
- need to separately store a pointer to the first item or "head" of the list
- the last item in the list is special
  it contains `NULL` in its `next` field instead of a pointer to an item

Example of List Item

Example of a list item used to store an address:



Example of List Item: C

```c
typedef struct address_node address_node;

struct address_node {
    address_node *next;
    char *telephone;
    char *email;
    char *address;
    char *telephone;
    char *email;
};
```

List Items

List items may hold large amount of data or many fields.
For simplicity, we'll assume each list item need store only a single int.

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

List Operations

Basic list operations:

- create a new item with specified data
- search for a item with particular data
- insert a new item to the list
- remove a item from the list

Many other operations are possible.

Creating a List Item

```c
// Create a new struct node containing the specified data,
// and next fields, return a pointer to the new struct node.

struct node *create_node(int data, struct node *next) {
    struct node *n;

    n = malloc(sizeof (struct node));
    if (n == NULL) {
        fprintf(stderr, "out of memory\n");
        exit(1);
    }
    n->data = data;
    n->next = next;
    return n;
}
```

Building a list

Building a list containing the 4 ints: 13, 17, 42, 5

```
struct node *head = create_node(5, NULL);
head = create_node(42, head);
head = create_node(17, head);
head = create_node(13, head);
```

Summing a List

```
// return sum of list data fields
int sum(struct node *head) {
    int sum = 0;
    struct node *n = head;
    // execute until end of list
    while (n != NULL) {
        sum += n->data;
        // make n point to next item
        n = n->next;
    }
    return sum;
}
```

Summing a List: For Loop

Same function but using a for loop instead of a while loop.
Compiler will produce same machine code as previous function.

```
// return sum of list data fields
int sum(struct node *head) {
    int sum = 0;
    for (struct node *n = head; n != NULL; n = n->next) {
        sum += n->data;
    }
    return sum;
}
```

Summing a List: Recursive

Same function but using a recursive call.
Compiler will produce same machine code as previous function.

```
// return sum of list data fields
int sum2(struct node *head) {
    if (head == NULL) {
        return 0;
    }
   return head->data + sum2(head->next);
}
```

Finding an Item in a List

```
// return pointer to first node containing
// specified value, return NULL if no such node
struct node *find_node(struct node *head, int data) {
    struct node *n = head;
    // search until end of list reached
    while (n != NULL) {
        if (n->data == data) {
            // matching item found
            return n;
        }
        // make n point to next item
        n = n->next;
    }
    // item not in list
    return NULL;
}
```

Finding an Item in a List: For Loop

Same function but using a for loop instead of a while loop.
Compiler will produce same machine code as previous function.

```
// return pointer to first node containing
// specified value, return NULL if no such node

struct node *find_node(struct node *head, int data) {
    for (struct node *n = head; n != NULL; n = n->next) {
        if (n->data == data) {
            return n;
        }
    }
    return NULL;
}
```

Finding an Item in a List: Shorter While Loop

Same function but using a more concise while loop.
Shorter does not always mean more readable.
Compiler will produce same machine code as previous functions.

```
// return pointer to first node containing
// specified value, return NULL if no such node

struct node *find_node(struct node *head, int data) {
    struct node *n = head;
    while (n != NULL && n->data != data) {
        n = n->next;
    }
    return n;
}
```

Finding an Item in a List: Recursive

Same function but function calls itself
Good compiler will produce same machine code as previous functions.

```
// return pointer to first node containing
// specified value, return NULL if no such node

struct node *find_node(struct node *head, int data) {
    if (head == NULL) {
        return NULL;
    }
    if (head->data == data) {
        return head;
    }
    return find_node(head->next, data);
}
```

Finding an Item in a List: Shorter Recursive

Same function but a more conside recursive version.
Shorter does not always mean more readable.
Good compiler will produce same machine code as previous functions.

```
// return pointer to first node containing
// specified value, return NULL if no such node

struct node *find_node(struct node *head, int data) {
    if (head == NULL || head->data == data) {
        return head;
    }
    return find_node(head->next, data);
}
```

Printing a List - Python Syntax

```c
// print contents of list in Python syntax
void print_list(struct node *head) {
    printf("[");
    for (struct node *n = head; n != NULL; n = n->next) {
        printf("%d", n->data);
        if (n->next != NULL) {
            printf(", ");
        }
    }
    printf("]");
}
```

Finding Last Item in List

```
// return pointer to last node in list
// NULL is returned if list is empty
struct node *last(struct node *head) {
    if (head == NULL) {
        return NULL;
    }

    struct node *n = head;
    while (n->next != NULL) {
        n = n->next;
    }
    return n;
}
```

Appending to List

```
// append integer to end of list

struct node *append(int value, struct node *head) {
    struct node *n;
    n = create_node(value, NULL);
    if (head == NULL) {
        // new node is now head of the list
        return n;
    } else {
        struct node *l = last(head);
        l->next = n;
        return list;
    }
}
```

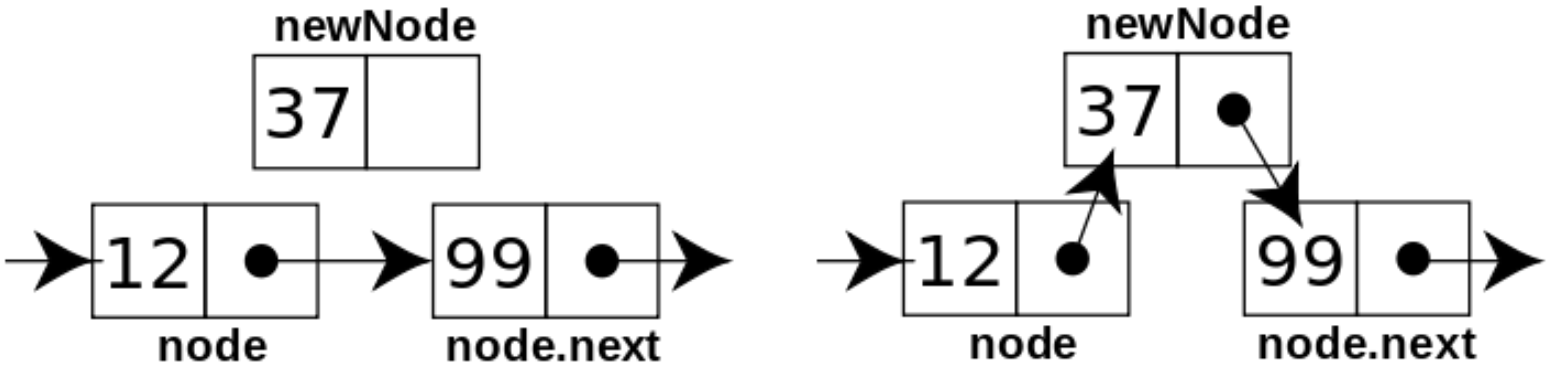
Deleting all items from a List

```
// Delete all the items from a linked list.

void delete_all(struct node *head) {
    struct node *n = head;
    struct node *tmp;

    while (n != NULL) {
        tmp = n;
        n = n->next;
        free(tmp);
    }
}
```

Insert a Node into an Ordered List



Insert a Node into an Ordered List

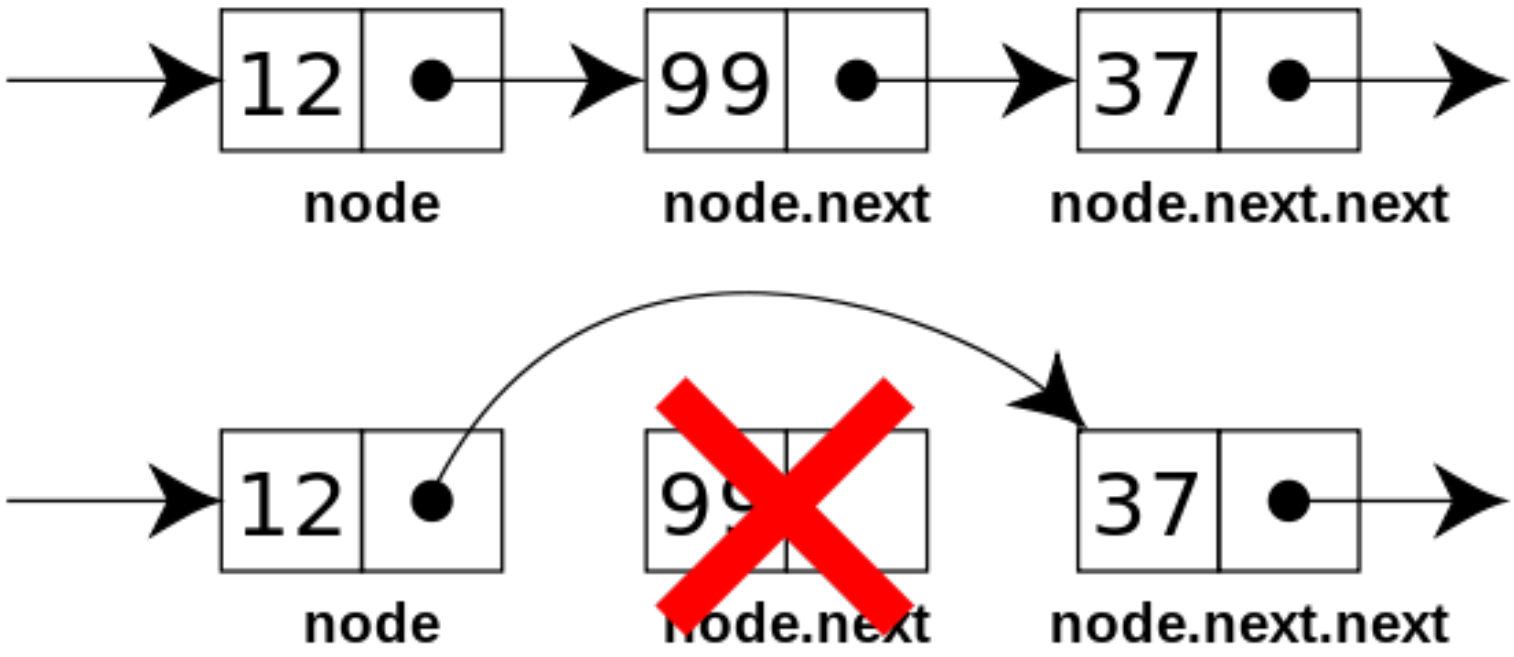
```
struct node *insert(struct node *head, struct node *node) {
    struct node *previous;
    struct node *n = head;
    // find correct position
    while (n != NULL && node->data > n->data) {
        previous = n;
        n = n->next;
    }

    // link new node into list
    if (previous == NULL) {
        head = node;
    } else {
        previous->next = node;
    }
    node->next = n;
    return head;
}
```

Insert a Node into an Ordered List: recursive

```
struct node *insert(struct node *head, struct node *node) {
    if (head == NULL || head->data >= node->data) {
        node->next = head;
        return node;
    }
    head->next = insert(head->next, node);
    return head;
}
```

Delete a Node from a List



Delete a Node from a List

```
struct node *delete(struct node *head, struct node *node) {
    if (node == head) {
        head = head->next;          // remove first item
        free(node);
    } else {
        struct node *previous = head;
        while (previous != NULL && previous->next != node) {
            previous = previous->next;
        }
        if (previous != NULL) { // node found in list
            previous->next = node->next;
            free(node);
        } else {
            fprintf(stderr, "warning: node not in list\n");
        }
    }
    return head;
}
```

Delete a Node from a List: Recursive

```
struct node *delete(struct node *head, struct node *node) {
    if (head == NULL) {
        fprintf(stderr, "warning: node not in list\n");
    } else if (node == head) {
        head = head->next;          // remove first item
        free(node);
    } else if (head == head) {
        head->next = delete(head->, node)
    }
    return head;
}
```