Data Structures
Linked Lists and Trees

Real-Life Computational Problems

- All about organizing data!
  - What shape the data should have to solve your problem
  - Where the data should flow so it is available when you need it
  - How your data can accommodate change and evolution of …
    - … your own program
    - … the requirements of your application

Support from Programming Languages

- E.g., *Java* knows about all kinds of
  - Lists, trees, arrays, collections
  - You tell it what you want and it does the rest
- E.g., *Scheme* is entirely built on lists
  - Anything a list can do is easy!
  - Anything a list cannot do is hard!
- E.g., *Matlab* is about matrices and vectors
  - Extensive support for linear and non-linear algebras

In the case of *C*

- *You are on your own!*

- Only built-in tools
  - Arrays
  - **`structs`** and **`unions`**
  - Functions

- Everything must be done “long-hand”

Theoretically

- Every computational problem can be solved with loops, arrays, non-recursive functions, and an unlimited amount of memory.
    - I.e., in Fortran!

- In reality, most real-life problems are much, much *too hard* to solve that way

Common Data Structures for Real-Life Problems

- Linked lists
  - One-way
  - Doubly-linked
  - Circular
- Trees
  - Binary
  - Multiple branches
- Hash Tables
  - Combine arrays and linked list
  - Especially for searching for objects by value

Definitions

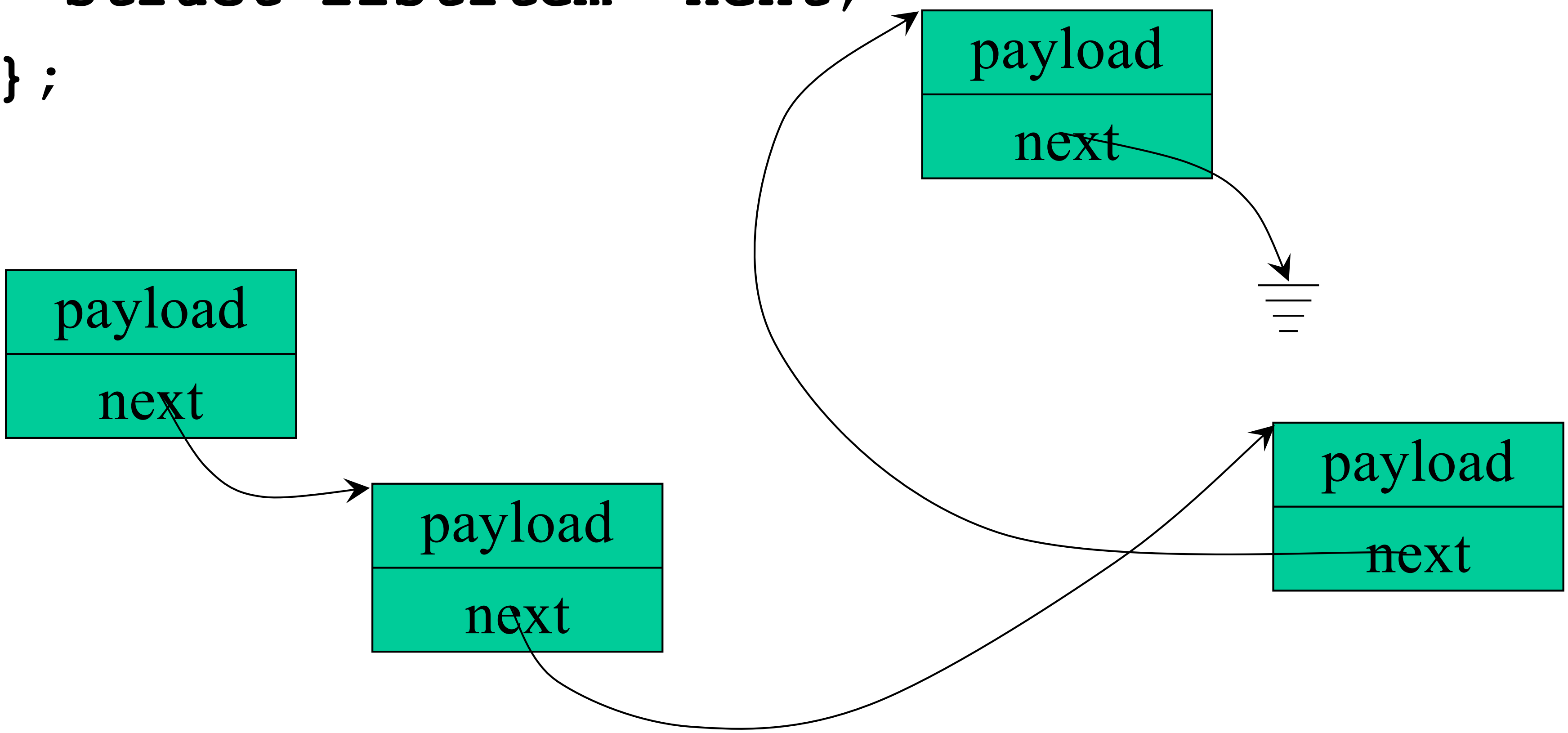
Note: elements are *usually* the same type (but not always).

- *Linked List*
  - A *data structure* in which each element is dynamically allocated and in which elements point to each other to define a *linear* relationship
  - Singly- or doubly-linked
  - Stack, queue, circular list
- *Tree*
  - A *data structure* in which each element is dynamically allocated and in which each element has more than one potential *successor*
  - Defines a *partial order*

Linked List

Note: payload may be multiple members.

```
struct listItem {
  type payload;
  struct listItem *next;
};
```



Linked List (continued)

- Items of list are usually same type
  - Generally obtained from **`malloc()`**
- Each item points to next item
- Last item points to null
- Need "**`head`**" to point to first item!

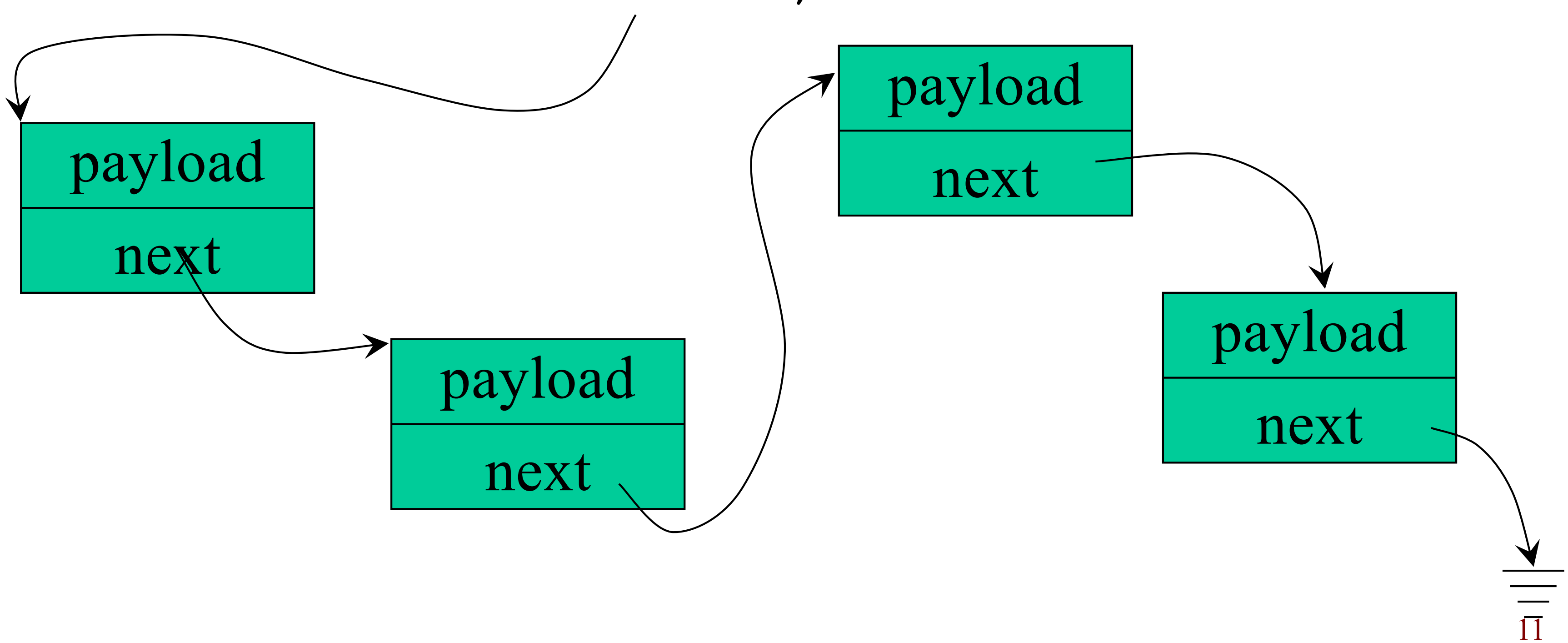
- "Payload" of item may be almost anything
  - A single member or multiple members
  - Any type of object whose size is known at compile time
  - Including **`struct`**, **`union`**, **`char *`** or other pointers
  - Also arrays of fixed size at compile time (see p. 214)

Usage of Linked Lists

- Not massive amounts of data
  - Linear search is okay
- Sorting not necessary
  - or sometimes not possible
- Need to add and delete data "on the fly"
  - Even from middle of list
- Items often need to be added to or deleted from the "ends"

Linked List (continued)

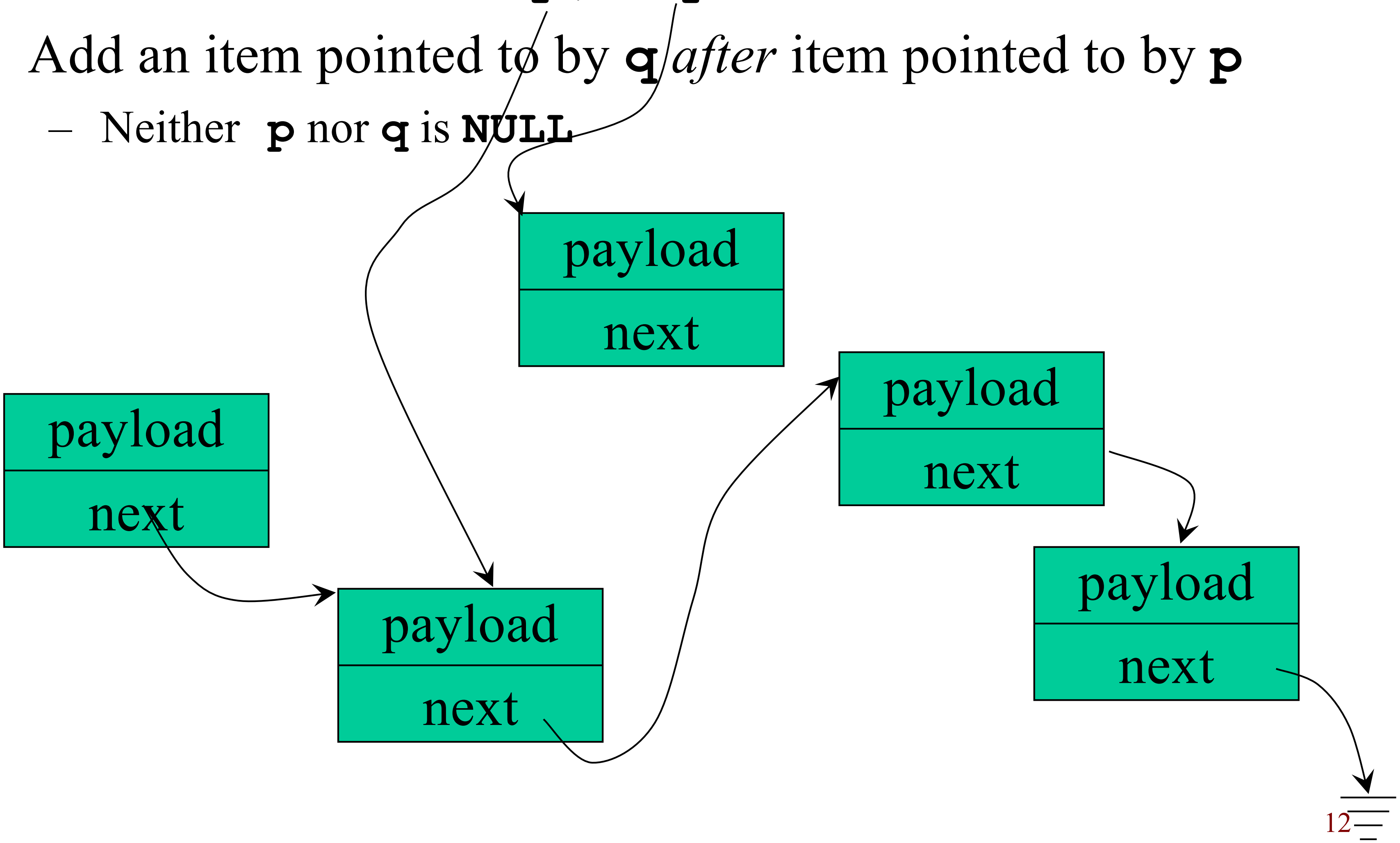
```
struct listItem {
  type payload;
  struct listItem *next;
};
struct listItem *head;
```



Adding an Item to a List

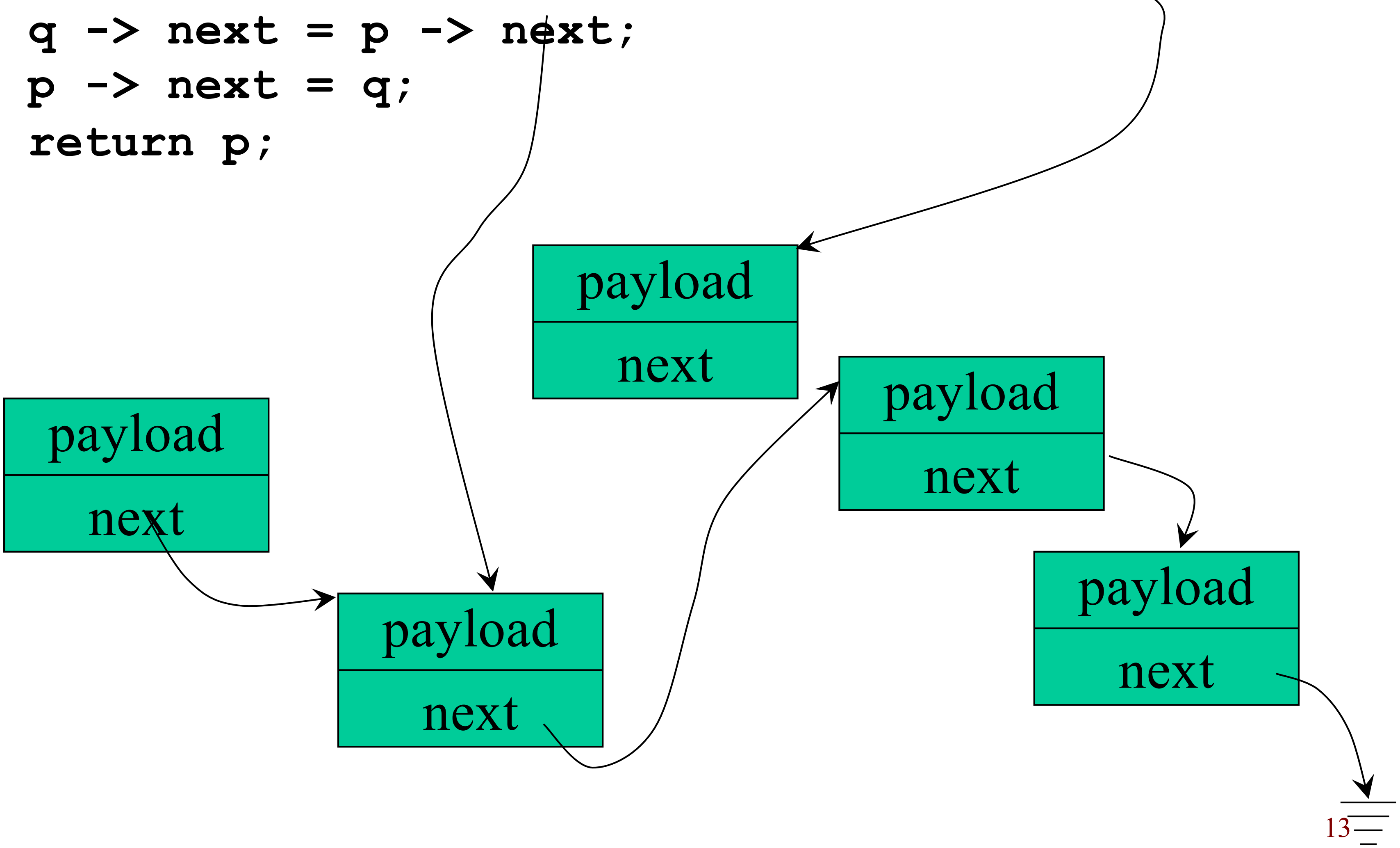
```
struct listItem *p, *q;
```

- Add an item pointed to by **`q`** *after* item pointed to by **`p`**
  - Neither **`p`** nor **`q`** is **`NULL`**



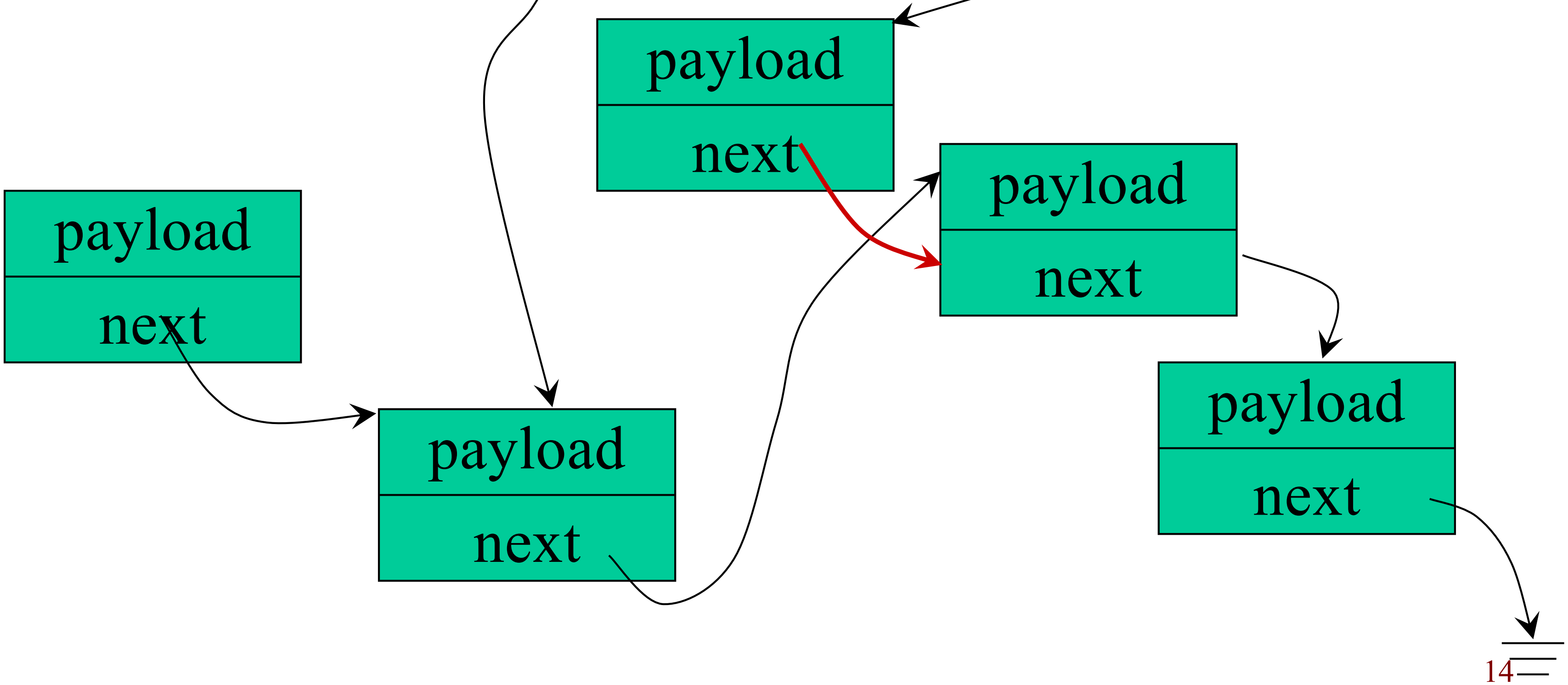
Adding an Item to a List

```
listItem *addAfter(listItem *p, listItem *q){
   q -> next = p -> next;
   p -> next = q;
   return p;
}
```



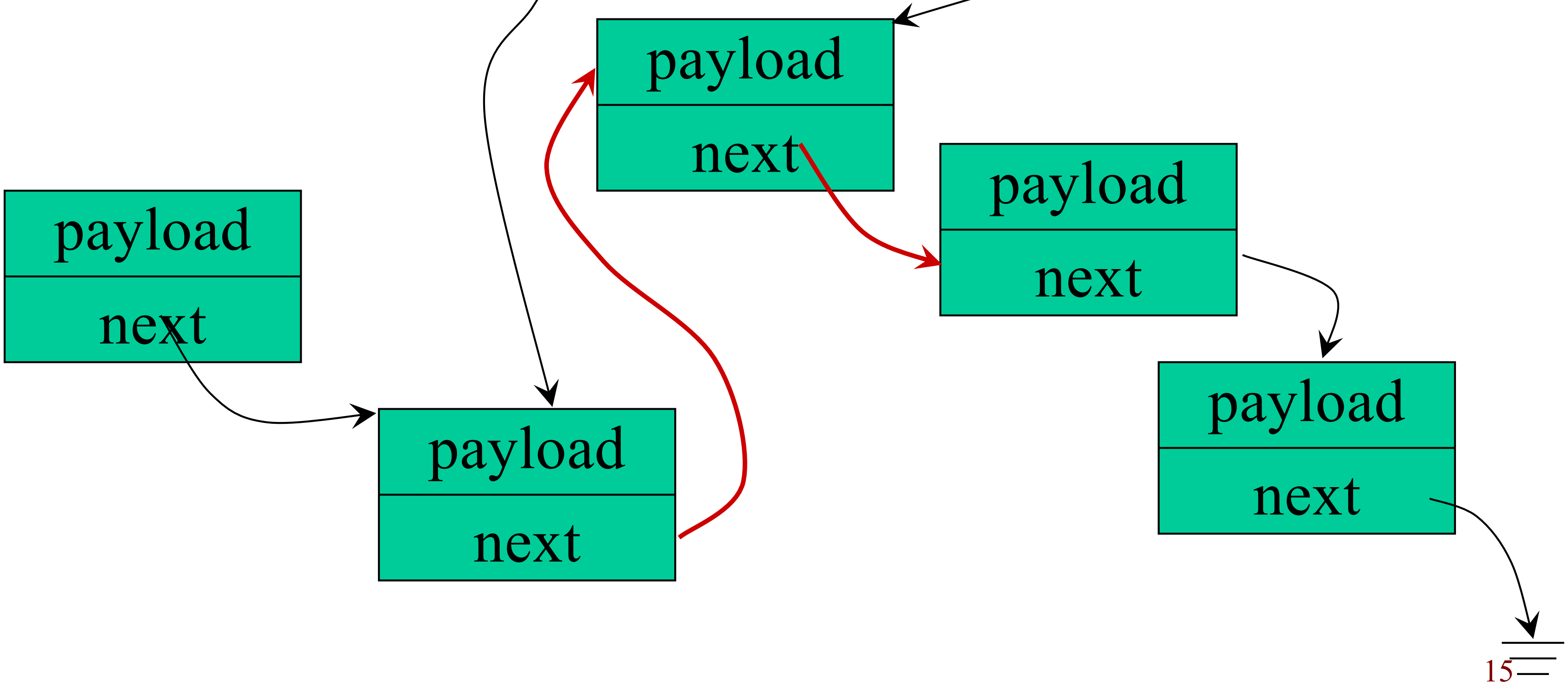
Adding an Item to a List

```
listItem *addAfter(listItem *p, listItem *q){
   q -> next = p -> next;
   p -> next = q;
   return p;
}
```



Adding an Item to a List

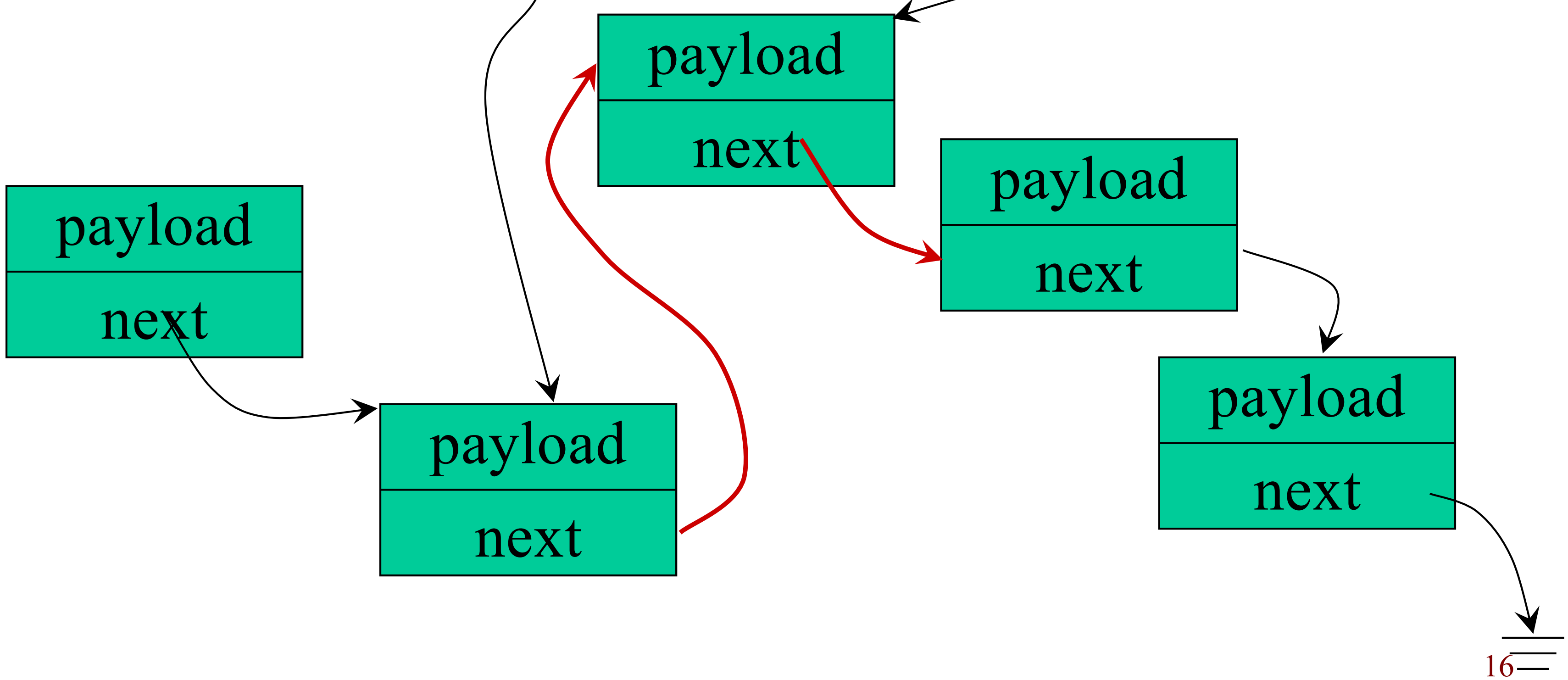
```
listItem *addAfter(listItem *p, listItem *q){
   q -> next = p -> next;
   p -> next = q;
   return p;
}
```



Adding an Item to a List (continued)

Note test for non-null *p* and *q*

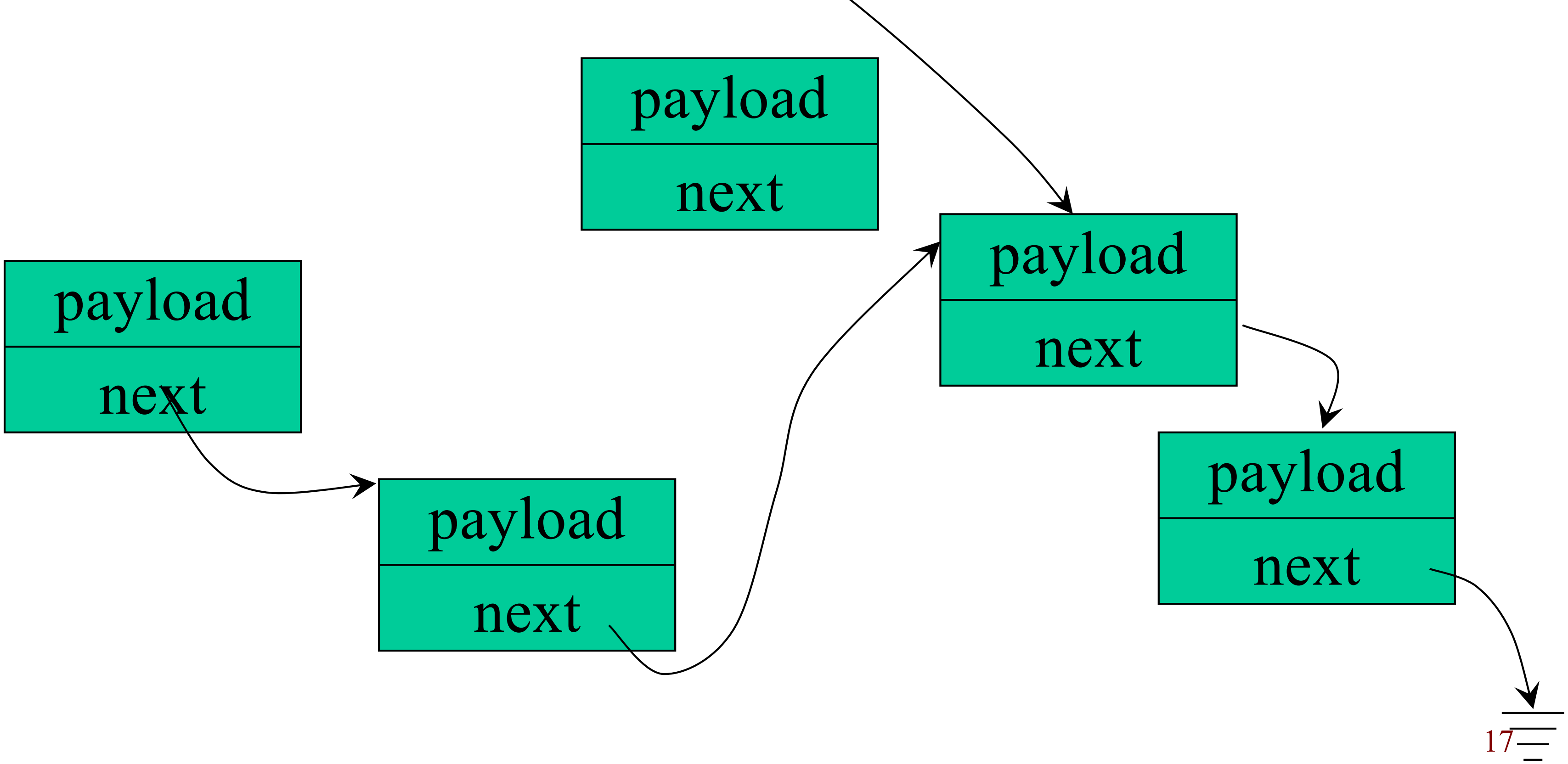
```
listItem *addAfter(listItem *p, listItem *q){
   if (p && q) {
        q -> next = p -> next;
        p -> next = q;
   }
   return p;
}
```



What about Adding an Item *before* another Item?

`struct listItem *p;`

- Add an item *before* item pointed to by **`p`** (**`p != NULL`**)

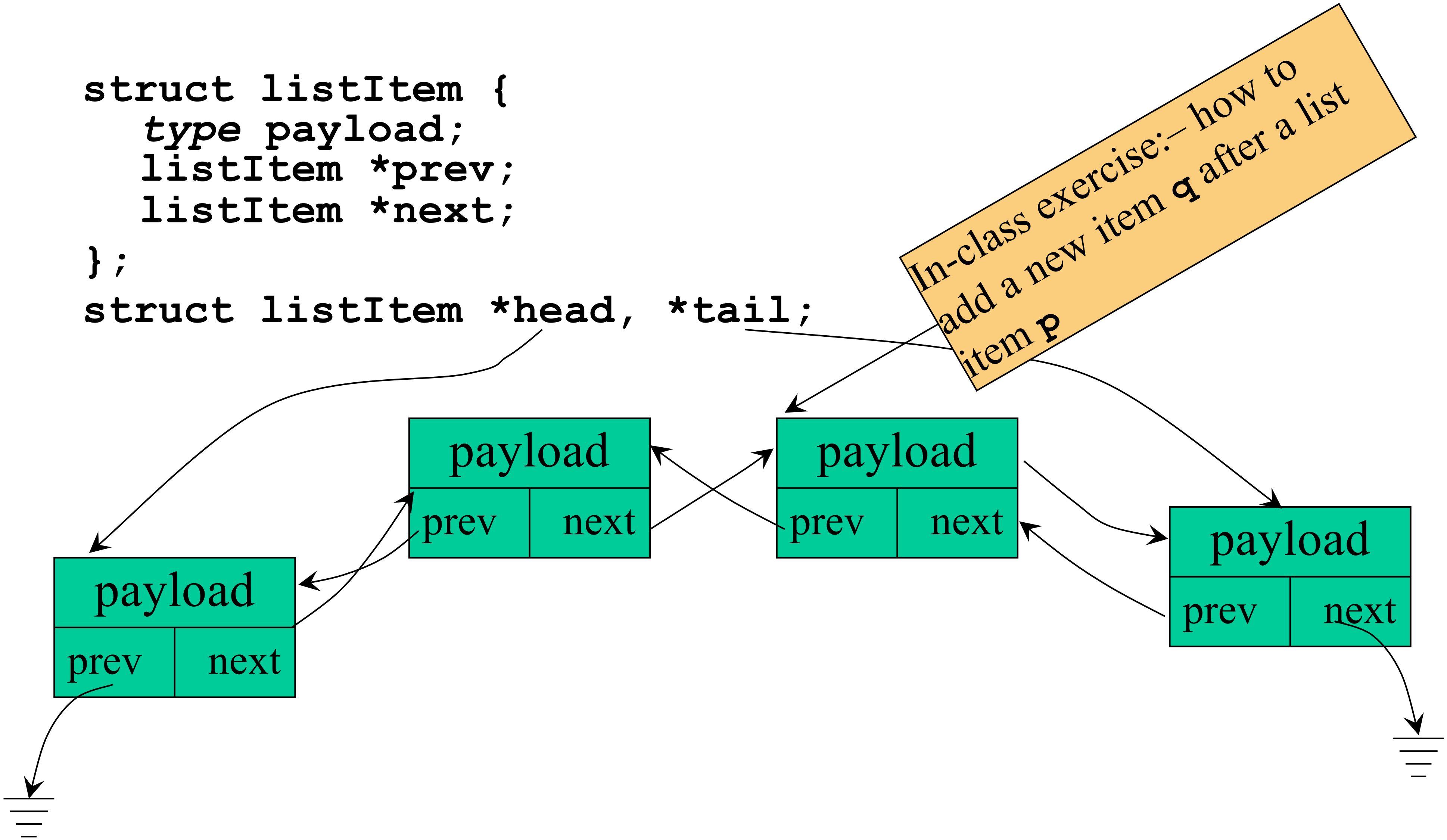


What about Adding an Item *before* another Item?

- Answer:–
  - Need to search list from beginning to find previous item
  - Add new item after previous item

Doubly-Linked List

```
struct listItem {
   type payload;
   listItem *prev;
   listItem *next;
};
struct listItem *head, *tail;
```



Other Kinds of List Structures

- *Queue* — FIFO (First In, First Out)
  - Items added at *end*
  - Items removed from *beginning*
- *Stack* — LIFO (Last In, First Out)
  - Items added at *beginning*, removed from *beginning*
- *Circular list*
  - Last item points to first item
  - Head may point to first or last item
  - Items added to *end*, removed from *beginning*

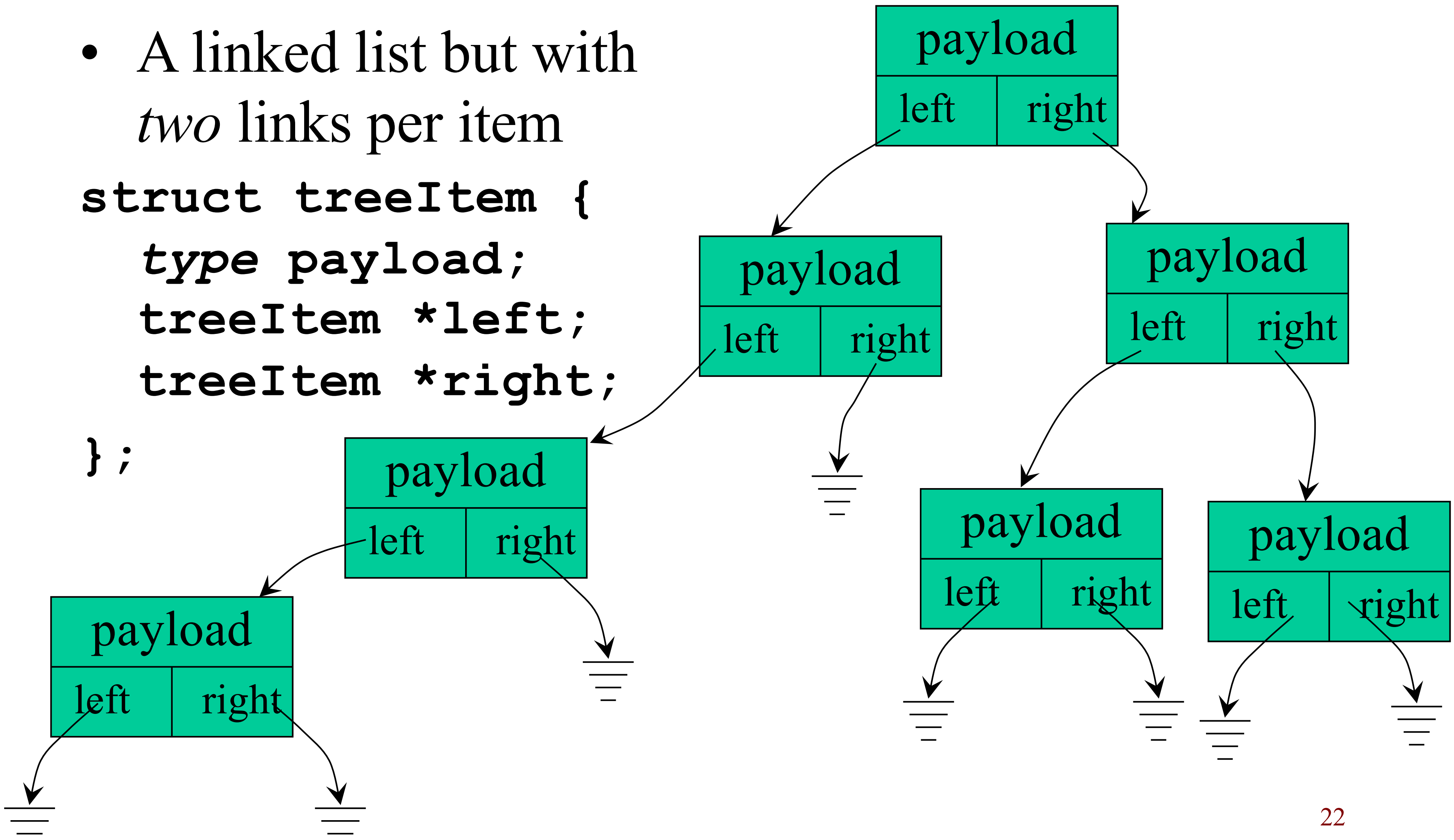
Definitions

- *Linked List*
  - A *data structure* in which each element is dynamically allocated and in which elements point to each other to define a linear relationship
  - Singly- or doubly-linked
  - Stack, queue, circular list
- *Tree*
  - A *data structure* in which each element is dynamically allocated and in which each element has more than one potential *successor*
  - Defines a *partial order*

Binary Tree

- A linked list but with *two* links per item

```
struct treeItem {
   type payload;
   treeItem *left;
   treeItem *right;
};
```



Binary Tree (continued)

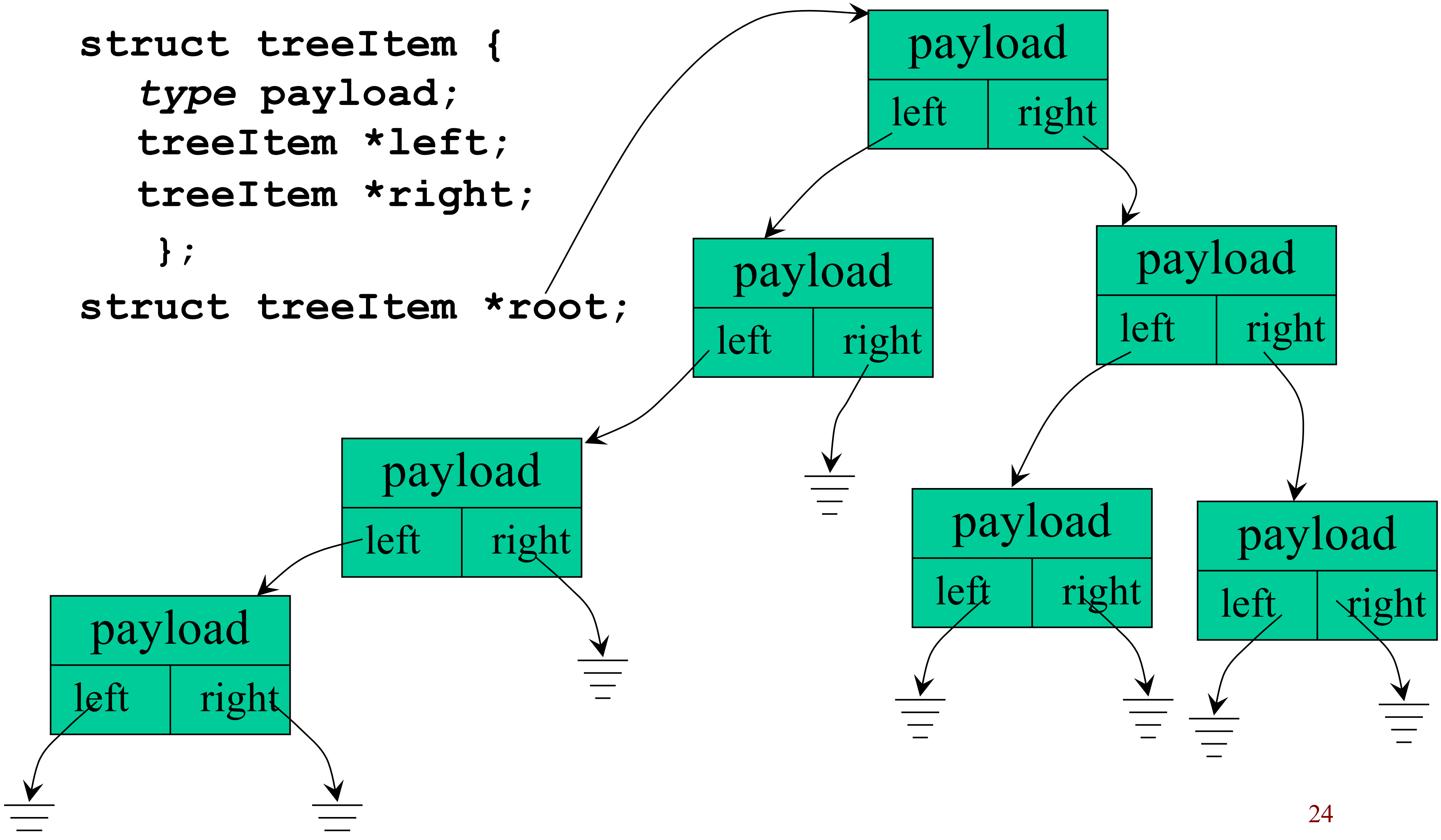
- Binary tree needs a *root*

```
struct treeItem {
  type payload;
  treeItem *left; treeItem *right;
};
struct treeItem *root;
```

- Binary trees often drawn with root at top!
  - Unlike ordinary trees in the forest
  - More like the root systems of a tree

Binary Tree

```
struct treeItem {
   type payload;
   treeItem *left;
   treeItem *right;
 };
struct treeItem *root;
```



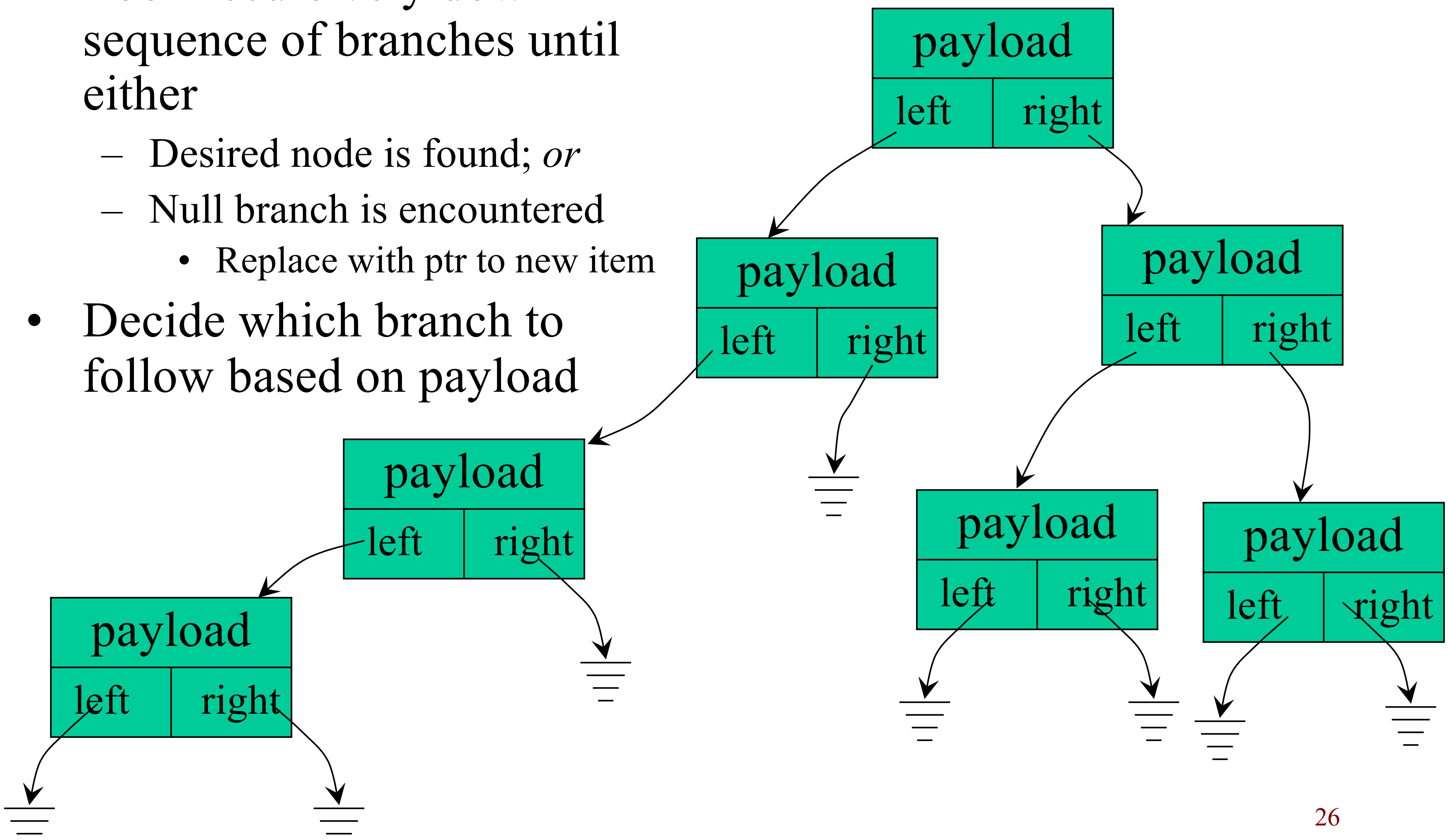
Purpose of a Tree

- (Potentially) a *very* large data structure
  - Capable of storing *very many* items
- Need to find items *by value*
  - I.e., need to search through the data structure to see if it contains an item with the value we want
- Need to add new items
  - If value is not already in the tree, add a new item …
  - …so that it can be easily found in future

- Why not use a *linked list?*

Searching and Adding to a Binary Tree

- Look recursively down sequence of branches until either
  - Desired node is found; *or*
  - Null branch is encountered
    - Replace with ptr to new item
- Decide which branch to follow based on payload



Example — Searching a Tree

```
typedef struct _treeItem {
   char *word;          // part of payload
   int count;           // part of payload
   _treeItem *left, *right;
   } treeItem;

treeItem *findItem(treeItem *p, char *w) {
   if (p == NULL)
       return NULL; // item not found

   int c = strcmp(w, p->word);
   if (c == 0)
       return p;
   else if (c < 0)
       return findItem(p->left, w);
   else
       return findItem(p->right, w);
}
```

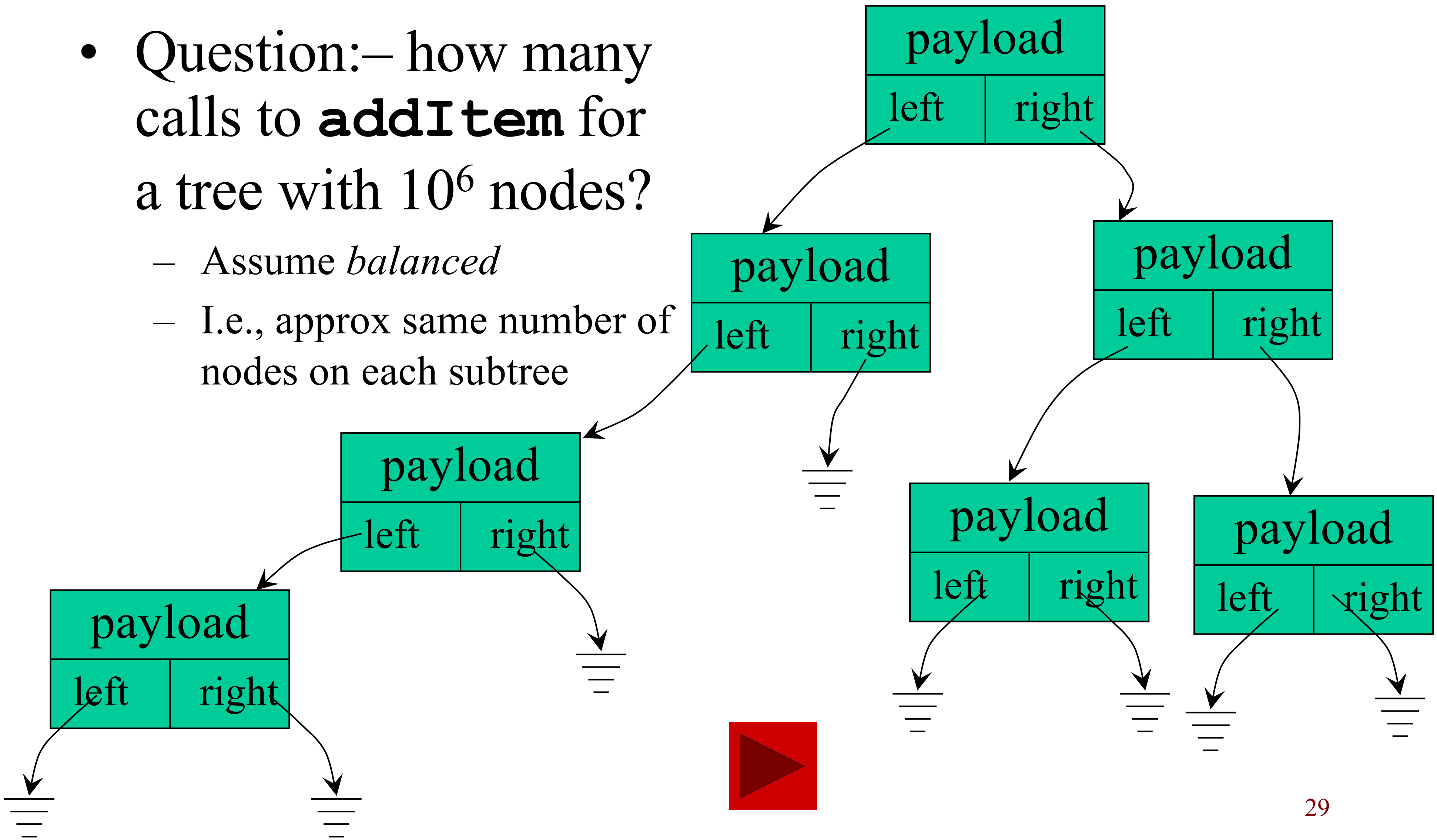
Example — Adding an Item

```
treeItem *addItem(treeItem *p, char *w) {
   if (p == NULL){
        p = malloc(sizeof(treeItem));
        char *c = malloc(strlen(w)+1);
        p->word = strcpy(c, w);
        p->count = 1;
        p->left = p->right = NULL;
        return p;
   };
   int c = strcmp(w, p->word);
   if (c == 0)
        p->count++;
   else if (c < 0)
        p->left = addItem(p->left, w);
   else
        p->right = addItem(p->right, w);
   return p;
}
```

Why do this?

Binary Tree

- Question:– how many calls to **`addItem`** for a tree with 10^6 nodes?
  - Assume *balanced*
  - I.e., approx same number of nodes on each subtree



Observation

- Problems like this occur in real life *all the time*
- Need to maintain a lot of data
  - Usually random
- Need to search through it quickly
- Need to add (or delete) items dynamically
- Need to sort "on the fly"
  - I.e., as you are adding and/or deleting items

Binary Trees (continued)

- Binary tree does *not* need to be "balanced"
  - i.e., with approximate same # of nodes hanging from right or left

- However, it often helps with performance

- Multiply-branched trees
  - Like binary trees, but with more than two links per node

Binary Trees (continued)

- Binary tree does *not* need to be "balanced"
  - i.e., with approximate same # of nodes hanging from right or left

"Big-O" notation:– means "order of"

- However, it helps with performance
  - Time to reach a *leaf* node is O($log_2\ n$), where *n* is number of nodes in tree

- Multiply-branched trees
  - Like binary trees, but with more than two links per node

Binary Tree Example

- Payload:–
  - **`char *word`** — the word at that node
  - **`int count`** — number of occurrences
  - Possibly other data
- When we are pointing to any node in the tree and have a word **`w`**, either:–
  - **`w`** is the same word as at that node, so just increase its **`count`**,
  - **`w`** is alphabetically *before* the word at that node, so look for it in the left subtree,
  - **`w`** is alphabetically *after* the word at that node, so look for it in the right subtree, or
  - The node is empty (i.e., null), so create one for that word.