Chapter 9: The List ADT and Generalized Lists

Data Structures and Algorithms with C

The list ADT contract · Array and linked implementations · Generalized lists · Copy · equality · traversal · Depth · inversion · Trees as lists · Sharing and reference counts

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides:
https://sercankulcu.github.io/files/dsa_c/slides/Chapter_09_List_ADT_and_Generalized_Lists.html

Learning Outcomes

The contract

Define the list ADT by its operations, preconditions, and postconditions; separate the implementation from the interface.

Comparison

Compare array and linked implementations on access, insertion, and memory.

Generalized lists

Define lists that contain atoms and sublists, and draw their memory layout.

Recursive routines

Write and trace recursive copy, equality, traversal, depth, and inversion.

Trees as lists

Turn a tree into a one-line linear form and explain the method's limits.

Sharing

Explain the pitfalls of shared and recursive lists, and the ideas behind reference counting and garbage collection.

The List ADT: What It Does, with the How Hidden

```c
typedef struct list List;                // opaque type

List  *list_create(void);
void   list_destroy(List *L);
int    list_length(const List *L);
bool   list_get(const List *L, int i, int *x);
bool   list_insert(List *L, int i, int x); // 0<=i<=n
bool   list_delete(List *L, int i, int *x); // 0<=i<n
int    list_find(const List *L, int x);   // -1 if absent
```

Linear list

$L = (a_0, a_1, \ldots, a_{n-1})$, $n \geq 0$; a_i comes before a_{i+1}.

Pre- / postcondition

`list_insert(L, i, x)` : before, $0 \leq i \leq n$; after, x is at position i, the length is n+1, and the old $a_i, \ldots, a_{n-1}$ have each moved one place right.

Opaque type

`struct list` is defined only in the .c file; client code can't tell whether it is an array or links, so swapping implementations is easy.

Array Implementation (Sequential Mapping)

```c
#define CAPACITY 100
typedef struct {
    int a[CAPACITY];
    int n;                  // item count
} ArrayList;

bool array_delete(ArrayList *L, int i, int *x) {
    int k;
    if (i < 0 || i >= L->n) return false;
    *x = L->a[i];
    for (k = i; k < L->n - 1; k++)
        L->a[k] = L->a[k + 1];      // shift left
    L->n--;
    return true;
}
```

Access

The address of `a[i]` is computed: **O(1)**.

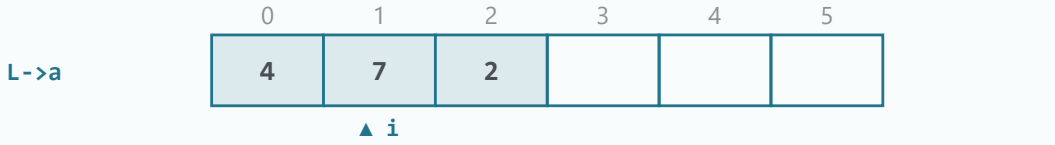
Insert / delete

The items after i must shift: **O(n)** in the worst case (insert or delete at the front).

Capacity

A fixed array fills up; doubling a dynamic array with `realloc` gives O(1) insertion on average.

Code Trace: `array_insert(&L, 1, 9)`



List [4, 7, 2], `n = 3`; 9 is to be inserted at **position 1** (0-based).

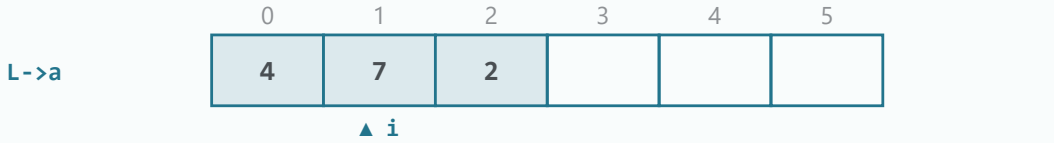
Step 1 / 9

```
▶ bool array_insert(ArrayList *L, int i, int x) {
2     int k;
3     if (L->n == CAPACITY || i < 0 || i > L->n)
4         return false;           // full/invalid
5     for (k = L->n; k > i; k--)
6         L->a[k] = L->a[k - 1];      // shift right
7     L->a[i] = x;
8     L->n++;
9     return true;
10 }
```

VARIABLES

`i = 1` `x = 9` `k = ?` `L→n = 3` `L→a = [4, 7, 2]`

Code Trace: `array_insert(&L, 1, 9)`



Check: not full, and `0 ≤ i ≤ n`; we can insert.

Step 2 / 9

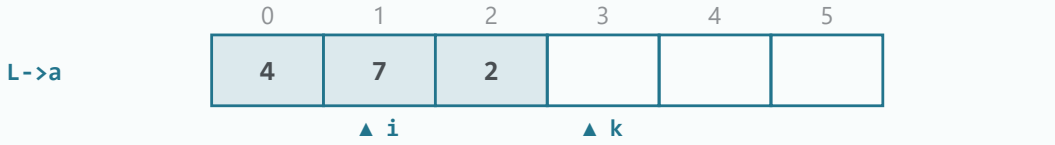
```
1  bool array_insert(ArrayList *L, int i, int x) {
2      int k;
▶      if (L->n == CAPACITY || i < 0 || i > L->n)
4          return false;           // full/invalid
5      for (k = L->n; k > i; k--)
6          L->a[k] = L->a[k - 1];      // shift right
7      L->a[i] = x;
8      L->n++;
9      return true;
10 }
```

VARIABLES

`i = 1` `x = 9` `k = ?` `L→n = 3`

`L→a = [4, 7, 2]`

Code Trace: `array_insert(&L, 1, 9)`



`k = 3` : `k > i` → true, shift.

Step 3 / 9

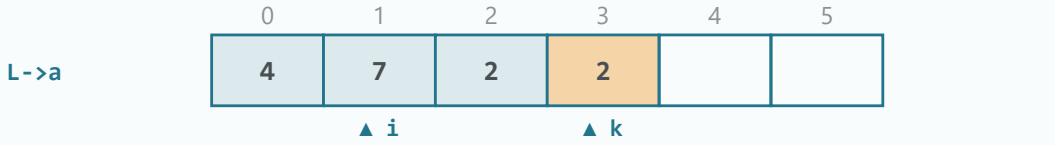
```
1  bool array_insert(ArrayList *L, int i, int x) {
2      int k;
3      if (L->n == CAPACITY || i < 0 || i > L->n)
4          return false;           // full/invalid
▶      for (k = L->n; k > i; k--)
6          L->a[k] = L->a[k - 1];      // shift right
7      L->a[i] = x;
8      L->n++;
9      return true;
10 }
```

VARIABLES

`i = 1` `x = 9` `k = 3` `L→n = 3`

`L→a = [4, 7, 2]`

Code Trace: `array_insert(&L, 1, 9)`



`a[3] = a[2]` : 2 moved one place right.

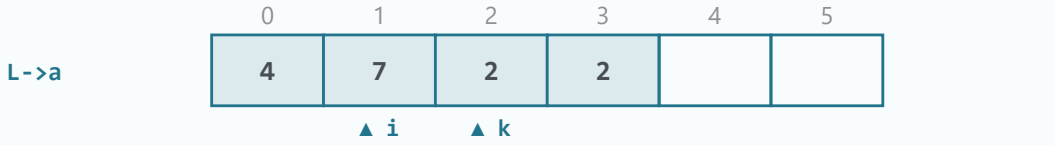
Step 4 / 9

```
1  bool array_insert(ArrayList *L, int i, int x) {
2      int k;
3      if (L->n == CAPACITY || i < 0 || i > L->n)
4          return false;           // full/invalid
5      for (k = L->n; k > i; k--)
▶          L->a[k] = L->a[k - 1];      // shift right
7      L->a[i] = x;
8      L->n++;
9      return true;
10 }
```

VARIABLES

`i = 1` `x = 9` `k = 3` `L→n = 3` `L→a = [4, 7, 2]`

Code Trace: `array_insert(&L, 1, 9)`



`k = 2` : `k > i` → true, shift.

Step 5 / 9

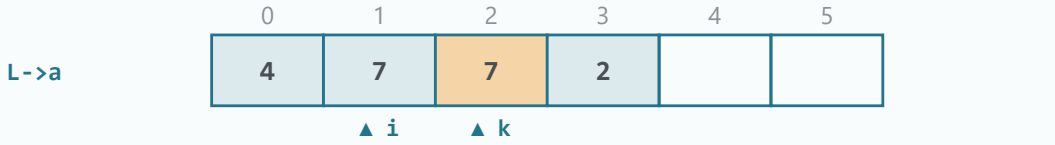
```
1  bool array_insert(ArrayList *L, int i, int x) {
2      int k;
3      if (L->n == CAPACITY || i < 0 || i > L->n)
4          return false;           // full/invalid
▶      for (k = L->n; k > i; k--)
6          L->a[k] = L->a[k - 1];      // shift right
7      L->a[i] = x;
8      L->n++;
9      return true;
10 }
```

VARIABLES

`i = 1` `x = 9` `k = 2` `L→n = 3`

`L→a = [4, 7, 2]`

Code Trace: `array_insert(&L, 1, 9)`



`a[2] = a[1]` : 7 moved one place right.

Step 6 / 9

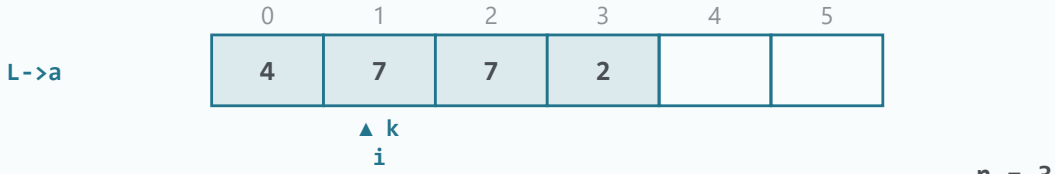
```
1  bool array_insert(ArrayList *L, int i, int x) {
2      int k;
3      if (L->n == CAPACITY || i < 0 || i > L->n)
4          return false;           // full/invalid
5      for (k = L->n; k > i; k--)
▶          L->a[k] = L->a[k - 1];      // shift right
7      L->a[i] = x;
8      L->n++;
9      return true;
10 }
```

VARIABLES

`i = 1` `x = 9` `k = 2` `L→n = 3`

`L→a = [4, 7, 7]`

Code Trace: `array_insert(&L, 1, 9)`



`k = 1` : `k > i` → false, the loop ends.

Step 7 / 9

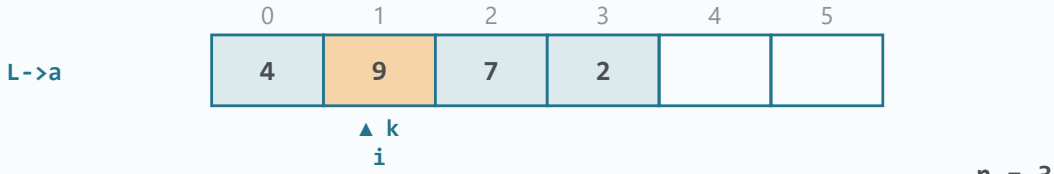
```
1  bool array_insert(ArrayList *L, int i, int x) {
2      int k;
3      if (L->n == CAPACITY || i < 0 || i > L->n)
4          return false;           // full/invalid
▶      for (k = L->n; k > i; k--)
6          L->a[k] = L->a[k - 1];      // shift right
7      L->a[i] = x;
8      L->n++;
9      return true;
10 }
```

VARIABLES

`i = 1` `x = 9` `k = 1` `L→n = 3`

`L→a = [4, 7, 7]`

Code Trace: `array_insert(&L, 1, 9)`



`a[i] = x` : 9 is written into the freed slot.

Step 8 / 9

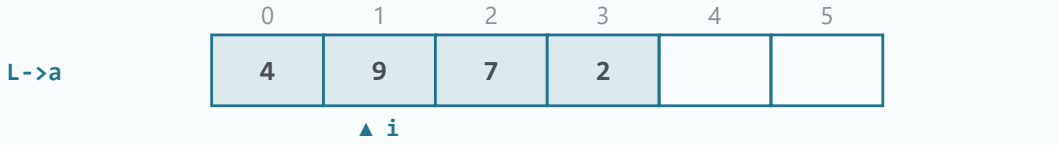
```
1  bool array_insert(ArrayList *L, int i, int x) {
2      int k;
3      if (L->n == CAPACITY || i < 0 || i > L->n)
4          return false;           // full/invalid
5      for (k = L->n; k > i; k--)
6          L->a[k] = L->a[k - 1];      // shift right
▶      L->a[i] = x;
8      L->n++;
9      return true;
10 }
```

VARIABLES

`i = 1` `x = 9` `k = 1` `L→n = 3`

`L→a = [4, 9, 7]`

Code Trace: `array_insert(&L, 1, 9)`



`n++` : the list is [4, 9, 7, 2]. The number of shifts is n − i: **O(n)** in the worst case (i = 0).

Step 9 / 9

```
1  bool array_insert(ArrayList *L, int i, int x) {
2      int k;
3      if (L->n == CAPACITY || i < 0 || i > L->n)
4          return false;           // full/invalid
5      for (k = L->n; k > i; k--)
6          L->a[k] = L->a[k - 1];      // shift right
7      L->a[i] = x;
▶      L->n++;
9      return true;
10 }
```

VARIABLES

`i = 1` `x = 9` `k = ?` `L→n = 4`

`L→a = [4, 9, 7, 2]`

Linked Implementation: a Pointer to a Pointer

```c
typedef struct node *node_pointer;
struct node { int data; node_pointer link; };
typedef struct { node_pointer head; int n; } LinkedList;

bool ll_insert(LinkedList *L, int i, int x) {
    node_pointer *pp = &L->head, new_node;
    if (i < 0 || i > L->n) return false;
    while (i-- > 0)
        pp = &(*pp)->link;       // walk to link i
    new_node = malloc(sizeof *new_node);
    if (new_node == NULL) return false;
    new_node->data = x;
    new_node->link = *pp;
    *pp = new_node;              // head or middle: same
    L->n++;
    return true;
}
```

`node_pointer *pp`

The **address of the link** to change: first `&L→head`, then `&prev->link`. Inserting at the head stops being a special case.

Cost

Walking to the position is O(i); linking is O(1). No shifting, no capacity limit.

Length

If an `n` field is kept, `list_length` is O(1).

Array or Linked List?

Criterion	Array (sequential)	Singly linked list
Access the i-th item	O(1)	O(i)
Insert / delete at the front	O(n) shifting	O(1)
Append at the end	O(1)* (if room)	O(n), O(1) with a tail pointer
Insert at a known position	O(n)	O(1) (if the predecessor is known)
Search (unsorted)	O(n)	O(n)
Search (sorted)	O(log n) binary search	O(n)
Memory	Data only; unused capacity	A pointer per node
Cache behavior	Contiguous: very good	Scattered: poor

Rule of thumb: **frequent access / binary search** → array; **frequent insertion and deletion in the middle, unknown size** → linked list.

From Linear Lists to Generalized Lists

Linear list

A = (α_0, …, α_{n-1}): every element is an **atom** (arrays, singly linked lists).

Definition

If each α_i may be an atom **or a sublist**, A is a **generalized list**. A is the list's name and n its length.

Head and tail

α_0 is the **head** of the list and (α_1, …, α_{n-1}) its **tail**. The tail is always a list.

The LISP connection

Head/tail are LISP's `car` / `cdr`; LISP programs are themselves generalized lists.

List	Length	Description
D = ()	0	empty list
A = (a, (b, c))	2	first element an atom, second the sublist (b, c); head(A) = a, tail(A) = ((b, c))
B = (A, A, ())	3	a **shared** sublist: the first two elements are the same A; the third is the empty list
C = (a, C)	2	a **recursive** list: it contains itself, C = (a, (a, (a, …)))

Example: a Multivariate Polynomial

```
typedef enum { VAR, PTR, NO } tag_t;
typedef struct pnode *poly_pointer;
struct pnode {
    tag_t tag;              // three kinds
    union {
        char var;           // VAR: variable name
        poly_pointer dlink; // PTR: coef is a list
        int coef;           // NO : integer coef
    };
    int exp;                // exponent
    poly_pointer link;
};
```

Three kinds of node

VAR: list header, the variable's name (exp = 0). **PTR**: the coefficient is a sublist. **NO**: the coefficient is an integer.

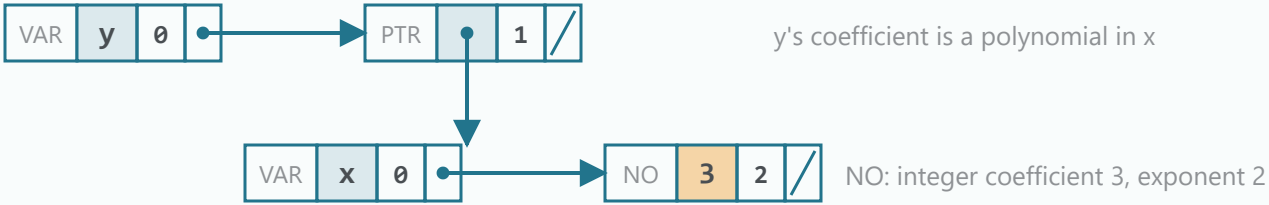
Grouping

P(x,y,z) is first grouped by z; each z-term's coefficient is a polynomial in y, and theirs are polynomials in x.

Instructor "This is just one way to implement polynomials with generalized lists. There are others. But you must also be able to write the routines that convert to and from it."

The Polynomial's Memory Layout

$3x^2y = (3x^2)\cdot y^1$

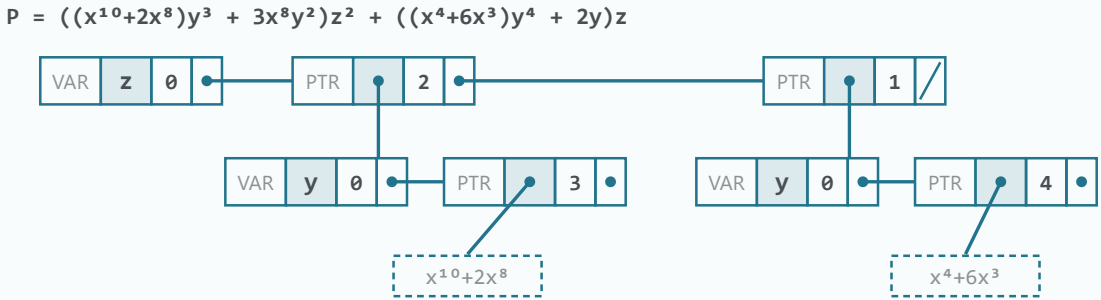


VAR: list header, the variable's name in `var`. **PTR**: the coefficient is the list in `dlink`.
NO: the coefficient is an integer.

Step 1 / 2

Lower levels are abbreviated to keep the drawing simple.

The Polynomial's Memory Layout



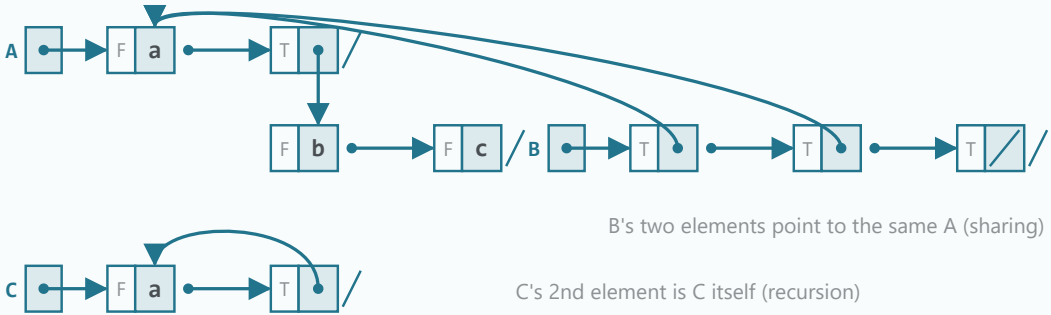
(the remaining y-list nodes and the x-lists are abbreviated)

$P(x,y,z) = x^{10}y^3z^2 + 2x^8y^3z^2 + 3x^8y^2z^2 + x^4y^4z + 6x^3y^4z + 2yz$, grouped by z and laid out as nested lists.

Step 2 / 2

Lower levels are abbreviated to keep the drawing simple.

Node Structure and Memory Layout



B's two elements point to the same A (sharing)

C's 2nd element is C itself (recursion)

```
typedef enum { FALSE, TRUE } boolean;
typedef struct list_node *list_pointer;
struct list_node {
    boolean tag;           // FALSE: atom
    union {
        char data;         // atom value
        list_pointer dlink; // sublist head
    };
    list_pointer link;     // next on this level
};
```

Node

[tag | data/dlink | link]: F = atom, T = sublist.

In this deck

Every routine uses this structure (nodes with a tag field).

The anonymous `union` is a C11 feature; for C99, name the union and write `L→u.data`.

Why Generalized Lists?

Compact

Hierarchical data such as a tree fits in the space of a one-line string: a kind of compression.

Natural hierarchy

File-system directories, popup menus, nested XML/JSON, LISP expressions.

Algorithm space

We move the data into a structure we know well; now every list routine is available.

Instructor "You could store the data as a tree, but it doesn't compare with text. 'I'll store a bitmap, I'll store the tree's memory image.' Fine, you can. But stored this way it takes less space… A computer engineer has to be careful even when storing data this small."

Instructor "Moving data into another space… Z-transformation… it is moved into a space that is well known and well understood. We took the data and moved it into an algorithm space we know very well."

QUESTION: copy a List with No Sharing and No Recursion

```
list_pointer copy(list_pointer L) {
    list_pointer q = NULL;
    if (L) {                    // L not NULL?
        q = get_node();         // get a node
        q->tag = L->tag;
        if (!L->tag) q->data = L->data; // atom
        else q->dlink = copy(L->dlink);
        q->link = copy(L->link);
    }
    return q;
}
```

Structure

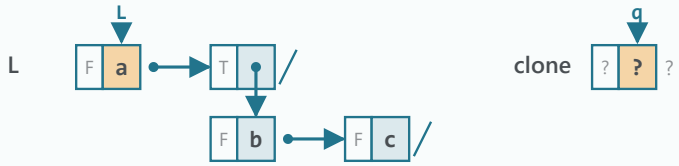
For each node, get a new node and copy the tag; for an atom copy the value, otherwise link in a **copy** of the sublist; then copy the rest.

Limitation

The routine is correct only for **unshared** lists; a shared sublist is copied separately everywhere it is referenced.

`get_node()` : a helper doing `malloc(sizeof *p)` plus a NULL check (it stops the program when memory runs out).

Code Trace: `copy(L)`, L = (a,(b,c))



`get_node()` : a new node is allocated for a (its fields are still garbage).

Step 1 / 20

```
1  list_pointer copy(list_pointer L) {
2      list_pointer q = NULL;
3      if (L) {                    // L not NULL?
▶          q = get_node();         // get a node
5          q->tag = L->tag;
6          if (!L->tag) q->data = L->data; // atom
7          else q->dlink = copy(L->dlink);
8          q->link = copy(L->link);
9      }
10     return q;
11 }
```

VARIABLES

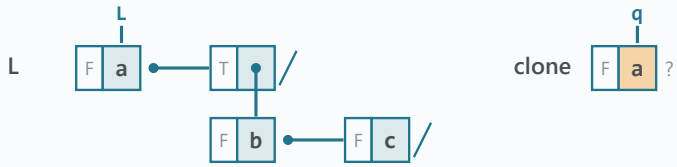
`L = &node(a)` `q = &clone(a)`

CALL STACK

`copy(a)`

`main()`

Code Trace: `copy(L)`, L = (a,(b,c))



Atom: `q→tag = FALSE`, `q→data = 'a'`.

Step 2 / 20

```
1  list_pointer copy(list_pointer L) {
2      list_pointer q = NULL;
3      if (L) {                    // L not NULL?
4          q = get_node();         // get a node
5          q->tag = L->tag;
▶          if (!L->tag) q->data = L->data; // atom
7          else q->dlink = copy(L->dlink);
8          q->link = copy(L->link);
9      }
10     return q;
11 }
```

VARIABLES

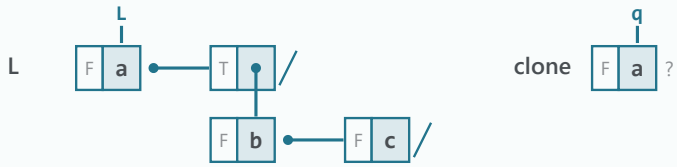
`L = &node(a)` `q = &clone(a)`

CALL STACK

`copy(a)`

`main()`

Code Trace: `copy(L)`, L = (a,(b,c))



Call for the next node: `copy(L→link)`.

Step 3 / 20

```
1  list_pointer copy(list_pointer L) {
2      list_pointer q = NULL;
3      if (L) {                    // L not NULL?
4          q = get_node();         // get a node
5          q->tag = L->tag;
6          if (!L->tag) q->data = L->data; // atom
7          else q->dlink = copy(L->dlink);
▶          q->link = copy(L->link);
9      }
10     return q;
11 }
```

VARIABLES

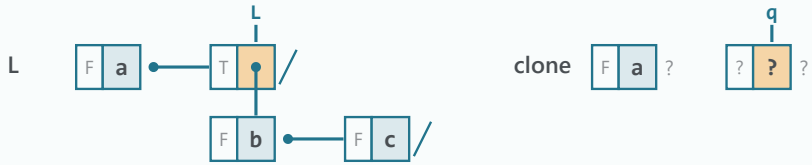
`L = &node(a)` `q = &clone(a)`

CALL STACK

`copy(a)`

`main()`

Code Trace: `copy(L)`, L = (a,(b,c))



`get_node()` : a new node is allocated for (b,c) (its fields are still garbage).

Step 4 / 20

```
1  list_pointer copy(list_pointer L) {
2      list_pointer q = NULL;
3      if (L) {                    // L not NULL?
▶          q = get_node();         // get a node
5          q->tag = L->tag;
6          if (!L->tag) q->data = L->data; // atom
7          else q->dlink = copy(L->dlink);
8          q->link = copy(L->link);
9      }
10     return q;
11 }
```

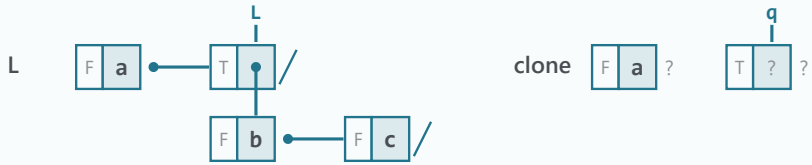
VARIABLES

- `L = &node((b,c))`
- `q = &clone((b,c))`

CALL STACK

- `copy((b,c))`
- `copy(a)`
- `main()`

Code Trace: `copy(L)`, L = (a,(b,c))



Sublist: first ask for a **copy of the sublist** (a recursive call).

Step 5 / 20

```
1  list_pointer copy(list_pointer L) {
2      list_pointer q = NULL;
3      if (L) {                    // L not NULL?
4          q = get_node();         // get a node
5          q->tag = L->tag;
6          if (!L->tag) q->data = L->data; // atom
▶          else q->dlink = copy(L->dlink);
8          q->link = copy(L->link);
9      }
10     return q;
11 }
```

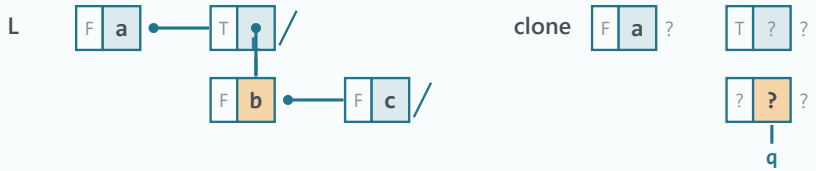
VARIABLES

- `L = &node((b,c))`
- `q = &clone((b,c))`

CALL STACK

- `copy((b,c))`
- `copy(a)`
- `main()`

Code Trace: `copy(L)`, L = (a,(b,c))



`get_node()` : a new node is allocated for b (its fields are still garbage).

Step 6 / 20

```
1  list_pointer copy(list_pointer L) {
2      list_pointer q = NULL;
3      if (L) {                    // L not NULL?
▶          q = get_node();         // get a node
5          q->tag = L->tag;
6          if (!L->tag) q->data = L->data; // atom
7          else q->dlink = copy(L->dlink);
8          q->link = copy(L->link);
9      }
10     return q;
11 }
```

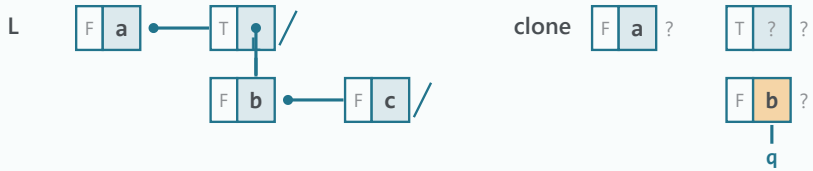
VARIABLES

`L = &node(b)` `q = &clone(b)`

CALL STACK

- `copy(b)`
- `copy((b,c))`
- `copy(a)`
- `main()`

Code Trace: `copy(L)`, L = (a,(b,c))



Atom: `q→tag = FALSE`, `q→data = 'b'`.

Step 7 / 20

```
1  list_pointer copy(list_pointer L) {
2      list_pointer q = NULL;
3      if (L) {                    // L not NULL?
4          q = get_node();         // get a node
5          q->tag = L->tag;
▶          if (!L->tag) q->data = L->data; // atom
7          else q->dlink = copy(L->dlink);
8          q->link = copy(L->link);
9      }
10     return q;
11 }
```

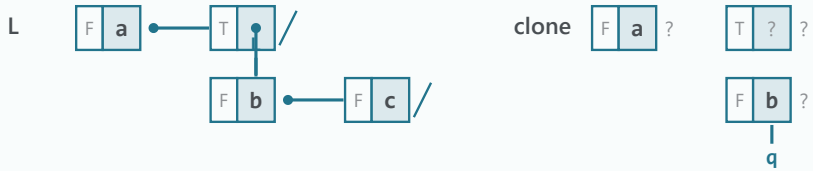
VARIABLES

`L = &node(b)` `q = &clone(b)`

CALL STACK

- `copy(b)`
- `copy((b,c))`
- `copy(a)`
- `main()`

Code Trace: `copy(L)`, L = (a,(b,c))



Call for the next node: `copy(L→link)`.

Step 8 / 20

```
1  list_pointer copy(list_pointer L) {
2      list_pointer q = NULL;
3      if (L) {                    // L not NULL?
4          q = get_node();         // get a node
5          q->tag = L->tag;
6          if (!L->tag) q->data = L->data; // atom
7          else q->dlink = copy(L->dlink);
▶          q->link = copy(L->link);
9      }
10     return q;
11 }
```

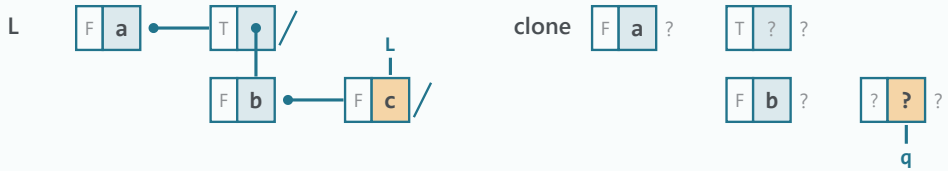
VARIABLES

`L = &node(b)` `q = &clone(b)`

CALL STACK

- `copy(b)`
- `copy((b,c))`
- `copy(a)`
- `main()`

Code Trace: `copy(L)`, L = (a,(b,c))



`get_node()` : a new node is allocated for c (its fields are still garbage).

Step 9 / 20

```
1  list_pointer copy(list_pointer L) {
2      list_pointer q = NULL;
3      if (L) {                    // L not NULL?
▶          q = get_node();         // get a node
5          q->tag = L->tag;
6          if (!L->tag) q->data = L->data; // atom
7          else q->dlink = copy(L->dlink);
8          q->link = copy(L->link);
9      }
10     return q;
11 }
```

VARIABLES

`L = &node(c)` `q = &clone(c)`

CALL STACK

- `copy(c)`
- `copy(b)`
- `copy((b,c))`
- `copy(a)`
- `main()`

Code Trace: `copy(L)`, L = (a,(b,c))



Atom: `q→tag = FALSE`, `q→data = 'c'`.

Step 10 / 20

```
1  list_pointer copy(list_pointer L) {
2      list_pointer q = NULL;
3      if (L) {                    // L not NULL?
4          q = get_node();         // get a node
5          q->tag = L->tag;
▶          if (!L->tag) q->data = L->data; // atom
7          else q->dlink = copy(L->dlink);
8          q->link = copy(L->link);
9      }
10     return q;
11 }
```

VARIABLES

`L = &node(c)` `q = &clone(c)`

CALL STACK

- `copy(c)`
- `copy(b)`
- `copy((b,c))`
- `copy(a)`
- `main()`

Code Trace: `copy(L)`, L = (a,(b,c))



Call for the next node: `copy(L→link)`.

Step 11 / 20

```
1  list_pointer copy(list_pointer L) {
2      list_pointer q = NULL;
3      if (L) {                    // L not NULL?
4          q = get_node();         // get a node
5          q->tag = L->tag;
6          if (!L->tag) q->data = L->data; // atom
7          else q->dlink = copy(L->dlink);
▶          q->link = copy(L->link);
9      }
10     return q;
11 }
```

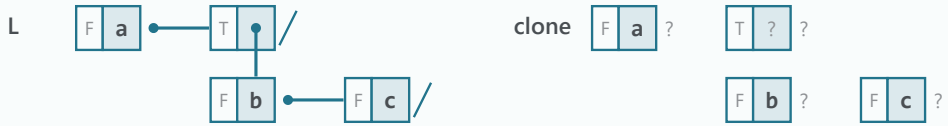
VARIABLES

`L = &node(c)` `q = &clone(c)`

CALL STACK

- `copy(c)`
- `copy(b)`
- `copy((b,c))`
- `copy(a)`
- `main()`

Code Trace: `copy(L)`, L = (a,(b,c))



`copy(NULL)` (link): `L` is NULL → returns `q = NULL` without allocating anything.

Step 12 / 20

```
1  list_pointer copy(list_pointer L) {
2      list_pointer q = NULL;
▶      if (L) {                    // L not NULL?
4          q = get_node();         // get a node
5          q->tag = L->tag;
6          if (!L->tag) q->data = L->data; // atom
7          else q->dlink = copy(L->dlink);
8          q->link = copy(L->link);
9      }
10     return q;
11 }
```

VARIABLES

`L = NULL` `q = NULL`

CALL STACK

- `copy(NULL)`
- `copy(c)`
- `copy(b)`
- `copy((b,c))`
- `copy(a)`
- `main()`

Code Trace: `copy(L)`, L = (a,(b,c))



Return: `q→link` = NULL (the last node).

Step 13 / 20

```
1  list_pointer copy(list_pointer L) {
2      list_pointer q = NULL;
3      if (L) {                    // L not NULL?
4          q = get_node();         // get a node
5          q->tag = L->tag;
6          if (!L->tag) q->data = L->data; // atom
7          else q->dlink = copy(L->dlink);
▶          q->link = copy(L->link);
9      }
10     return q;
11 }
```

VARIABLES

`L = &node(c)` `q = &clone(c)`

CALL STACK

- `copy(c)`
- `copy(b)`
- `copy((b,c))`
- `copy(a)`
- `main()`

Code Trace: `copy(L)`, L = (a,(b,c))



Return: `q→link` ← &clone(c).

Step 14 / 20

```
1  list_pointer copy(list_pointer L) {
2      list_pointer q = NULL;
3      if (L) {                    // L not NULL?
4          q = get_node();         // get a node
5          q->tag = L->tag;
6          if (!L->tag) q->data = L->data; // atom
7          else q->dlink = copy(L->dlink);
▶          q->link = copy(L->link);
9      }
10     return q;
11 }
```

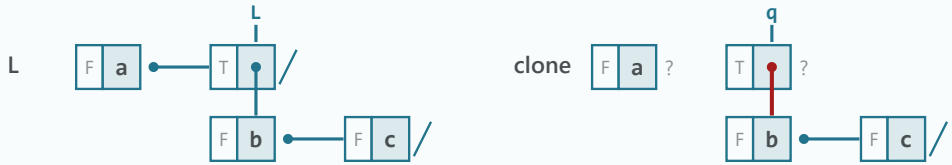
VARIABLES

`L = &node(b)` `q = &clone(b)`

CALL STACK

- `copy(b)`
- `copy((b,c))`
- `copy(a)`
- `main()`

Code Trace: `copy(L)`, L = (a,(b,c))



Return: `q→dlink` ← the sublist's copy &clone(b).

Step 15 / 20

```
1  list_pointer copy(list_pointer L) {
2      list_pointer q = NULL;
3      if (L) {                    // L not NULL?
4          q = get_node();         // get a node
5          q->tag = L->tag;
6          if (!L->tag) q->data = L->data; // atom
▶          else q->dlink = copy(L->dlink);
8          q->link = copy(L->link);
9      }
10     return q;
11 }
```

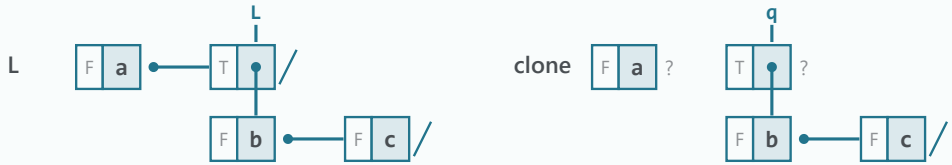
VARIABLES

- `L = &node((b,c))`
- `q = &clone((b,c))`

CALL STACK

- `copy((b,c))`
- `copy(a)`
- `main()`

Code Trace: `copy(L)`, L = (a,(b,c))



Call for the next node: `copy(L→link)`.

Step 16 / 20

```
1  list_pointer copy(list_pointer L) {
2      list_pointer q = NULL;
3      if (L) {                    // L not NULL?
4          q = get_node();         // get a node
5          q->tag = L->tag;
6          if (!L->tag) q->data = L->data; // atom
7          else q->dlink = copy(L->dlink);
▶          q->link = copy(L->link);
9      }
10     return q;
11 }
```

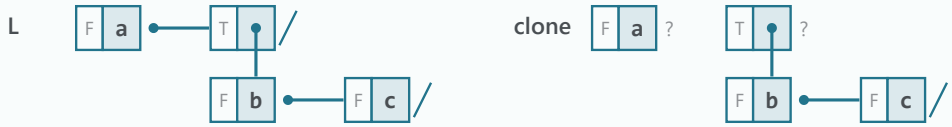
VARIABLES

- `L = &node((b,c))`
- `q = &clone((b,c))`

CALL STACK

- `copy((b,c))`
- `copy(a)`
- `main()`

Code Trace: `copy(L)`, L = (a,(b,c))



`copy(NULL)` (link): `L` is NULL → returns `q = NULL` without allocating anything.

Step 17 / 20

```
1  list_pointer copy(list_pointer L) {
2      list_pointer q = NULL;
▶      if (L) {                    // L not NULL?
4          q = get_node();         // get a node
5          q->tag = L->tag;
6          if (!L->tag) q->data = L->data; // atom
7          else q->dlink = copy(L->dlink);
8          q->link = copy(L->link);
9      }
10     return q;
11 }
```

VARIABLES

`L = NULL` `q = NULL`

CALL STACK

- `copy(NULL)`
- `copy((b,c))`
- `copy(a)`
- `main()`

Code Trace: `copy(L)`, L = (a,(b,c))



Return: `q→link` = NULL (the last node).

Step 18 / 20

```
1  list_pointer copy(list_pointer L) {
2      list_pointer q = NULL;
3      if (L) {                    // L not NULL?
4          q = get_node();         // get a node
5          q->tag = L->tag;
6          if (!L->tag) q->data = L->data; // atom
7          else q->dlink = copy(L->dlink);
▶          q->link = copy(L->link);
9      }
10     return q;
11 }
```

VARIABLES

`L = &node((b,c))`

`q = &clone((b,c))`

CALL STACK

- `copy((b,c))`
- `copy(a)`
- `main()`

Code Trace: `copy(L)`, L = (a,(b,c))



Return: `q→link` ← &clone((b,c)).

Step 19 / 20

```
1  list_pointer copy(list_pointer L) {
2      list_pointer q = NULL;
3      if (L) {                    // L not NULL?
4          q = get_node();         // get a node
5          q->tag = L->tag;
6          if (!L->tag) q->data = L->data; // atom
7          else q->dlink = copy(L->dlink);
▶          q->link = copy(L->link);
9      }
10     return q;
11 }
```

VARIABLES

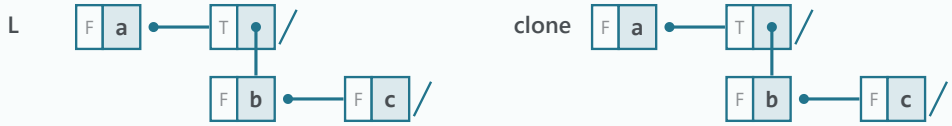
`L = &node(a)` `q = &clone(a)`

CALL STACK

`copy(a)`

`main()`

Code Trace: `copy(L)`, L = (a,(b,c))



The copy is done: the same structure, built from **new** nodes. Nodes are taken front to back and filled in back to front.

Step 20 / 20

```
1  list_pointer copy(list_pointer L) {
2      list_pointer q = NULL;
3      if (L) {                    // L not NULL?
4          q = get_node();         // get a node
5          q->tag = L->tag;
6          if (!L->tag) q->data = L->data; // atom
7          else q->dlink = copy(L->dlink);
8          q->link = copy(L->link);
9      }
▶      return q;
11 }
```

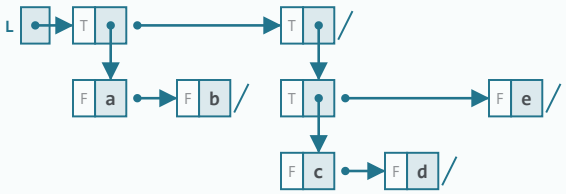
VARIABLES

`result = &clone(a)`

CALL STACK

`main()`

Copying: a Test List and the Limits



For testing: L = ((a,b),((c,d),e))

Shared list

Copying B = (A,A,()) copies A **twice**; the sharing is lost in the copy.

Recursive list

For C = (a,C) the calls never end; the stack overflows and the program crashes.

Instructor "Taken front to back, filled in back to front… Is anyone still saying 'Professor, I won't write it recursively'? If someone says 'I'll use a queue,' well, I have nothing more to say…"

QUESTION: Are Two Lists (No Sharing, No Recursion) equal?

```
boolean equal(list_pointer T, list_pointer L) {
    boolean x = TRUE;
    if (!T && !L) return TRUE;
    if (T && L && T->tag == L->tag) {
        if (!T->tag)              // atoms?
            x = (T->data == L->data) ? TRUE : FALSE;
        else                      // sublists
            x = equal(T->dlink, L->dlink);
        if (x == TRUE) return equal(T->link, L->link);
    }
    return FALSE;
}
```

Logic

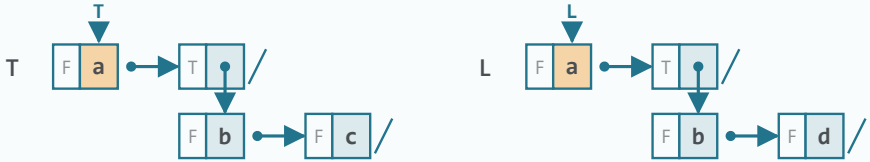
Equal if both end together. If the tags match: compare values for atoms and recurse for sublists; then compare the rest.

Limitation

On recursive lists (C = (a,C)) the calls go on forever.

Instructor "This isn't masterful code. Better code can be written. And recursive lists can't be compared because of the infinite calls."

Code Trace: `equal(T, L)`



Both exist and their `tag` fields match: compare the contents.

Step 1 / 15

```
1  boolean equal(list_pointer T, list_pointer L) {
2      boolean x = TRUE;
3      if (!T && !L) return TRUE;
▶      if (T && L && T->tag == L->tag) {
5          if (!T->tag)              // atoms?
6              x = (T->data == L->data) ? TRUE : FALSE;
7          else                      // sublists
8              x = equal(T->dlink, L->dlink);
9          if (x == TRUE) return equal(T->link, L->link);
10     }
11     return FALSE;
12 }
```

VARIABLES

`T = &node(a)` `L = &node(a)` `x = TRUE`

CALL STACK

`equal(a, a)`

`main()`

Code Trace: `equal(T, L)`



Atom: `'a' == 'a'` → `x = TRUE`.

Step 2 / 15

```
1  boolean equal(list_pointer T, list_pointer L) {
2      boolean x = TRUE;
3      if (!T && !L) return TRUE;
4      if (T && L && T->tag == L->tag) {
5          if (!T->tag)              // atoms?
▶              x = (T->data == L->data) ? TRUE : FALSE;
7          else                      // sublists
8              x = equal(T->dlink, L->dlink);
9          if (x == TRUE) return equal(T->link, L->link);
10     }
11     return FALSE;
12 }
```

VARIABLES

`T = &node(a)` `L = &node(a)` `x = TRUE`

CALL STACK

`equal(a, a)`

`main()`

Code Trace: `equal(T, L)`



x is true: compare the rest of the nodes (the `link` fields).

Step 3 / 15

```
1  boolean equal(list_pointer T, list_pointer L) {
2      boolean x = TRUE;
3      if (!T && !L) return TRUE;
4      if (T && L && T->tag == L->tag) {
5          if (!T->tag)              // atoms?
6              x = (T->data == L->data) ? TRUE : FALSE;
7          else                      // sublists
8              x = equal(T->dlink, L->dlink);
▶          if (x == TRUE) return equal(T->link, L->link);
10     }
11     return FALSE;
12 }
```

VARIABLES

`T = &node(a)` `L = &node(a)` `x = TRUE`

CALL STACK

`equal(a, a)`

`main()`

Code Trace: `equal(T, L)`



Both exist and their `tag` fields match: compare the contents.

Step 4 / 15

```
1  boolean equal(list_pointer T, list_pointer L) {
2      boolean x = TRUE;
3      if (!T && !L) return TRUE;
▶      if (T && L && T->tag == L->tag) {
5          if (!T->tag)              // atoms?
6              x = (T->data == L->data) ? TRUE : FALSE;
7          else                      // sublists
8              x = equal(T->dlink, L->dlink);
9          if (x == TRUE) return equal(T->link, L->link);
10     }
11     return FALSE;
12 }
```

VARIABLES

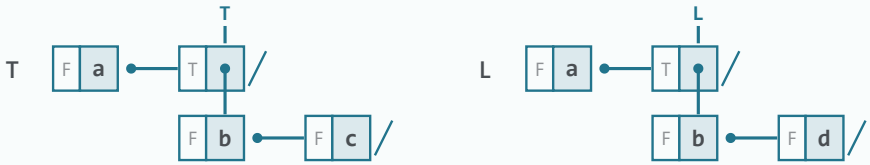
`T = &node((b,c))`

`L = &node((b,d))` `x = TRUE`

CALL STACK

- `equal((b,c), (b,d))`
- `equal(a, a)`
- `main()`

Code Trace: `equal(T, L)`



Sublist: compare the sublists first.

Step 5 / 15

```
1  boolean equal(list_pointer T, list_pointer L) {
2      boolean x = TRUE;
3      if (!T && !L) return TRUE;
4      if (T && L && T->tag == L->tag) {
5          if (!T->tag)              // atoms?
6              x = (T->data == L->data) ? TRUE : FALSE;
7          else                      // sublists
           x = equal(T->dlink, L->dlink);
9          if (x == TRUE) return equal(T->link, L->link);
10     }
11     return FALSE;
12 }
```

VARIABLES

`T = &node((b,c))`

`L = &node((b,d))` `x = TRUE`

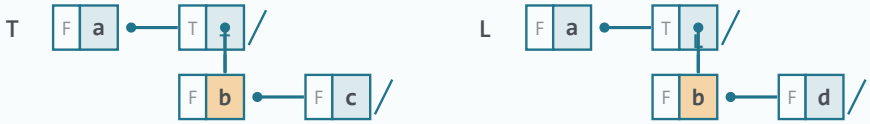
CALL STACK

`equal((b,c), (b,d))`

`equal(a, a)`

`main()`

Code Trace: `equal(T, L)`



Both exist and their `tag` fields match: compare the contents.

Step 6 / 15

```
1  boolean equal(list_pointer T, list_pointer L) {
2      boolean x = TRUE;
3      if (!T && !L) return TRUE;
▶      if (T && L && T->tag == L->tag) {
5          if (!T->tag)              // atoms?
6              x = (T->data == L->data) ? TRUE : FALSE;
7          else                      // sublists
8              x = equal(T->dlink, L->dlink);
9          if (x == TRUE) return equal(T->link, L->link);
10     }
11     return FALSE;
12 }
```

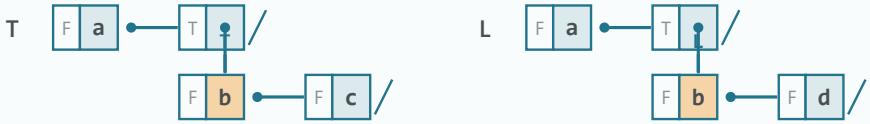
VARIABLES

`T = &node(b)` `L = &node(b)` `x = TRUE`

CALL STACK

- `equal(b, b)`
- `equal((b,c), (b,d))`
- `equal(a, a)`
- `main()`

Code Trace: `equal(T, L)`



Atom: `'b' == 'b'` → `x = TRUE`.

Step 7 / 15

```
1  boolean equal(list_pointer T, list_pointer L) {
2      boolean x = TRUE;
3      if (!T && !L) return TRUE;
4      if (T && L && T->tag == L->tag) {
5          if (!T->tag)              // atoms?
▶              x = (T->data == L->data) ? TRUE : FALSE;
7          else                      // sublists
8              x = equal(T->dlink, L->dlink);
9          if (x == TRUE) return equal(T->link, L->link);
10     }
11     return FALSE;
12 }
```

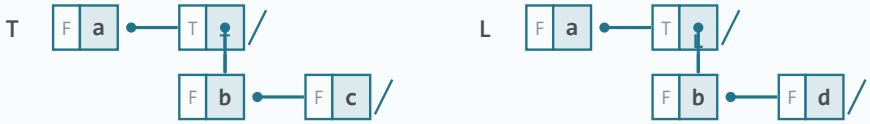
VARIABLES

`T = &node(b)` `L = &node(b)` `x = TRUE`

CALL STACK

- `equal(b, b)`
- `equal((b,c), (b,d))`
- `equal(a, a)`
- `main()`

Code Trace: `equal(T, L)`



x is true: compare the rest of the nodes (the `link` fields).

Step 8 / 15

```
1  boolean equal(list_pointer T, list_pointer L) {
2      boolean x = TRUE;
3      if (!T && !L) return TRUE;
4      if (T && L && T->tag == L->tag) {
5          if (!T->tag)              // atoms?
6              x = (T->data == L->data) ? TRUE : FALSE;
7          else                      // sublists
8              x = equal(T->dlink, L->dlink);
▶          if (x == TRUE) return equal(T->link, L->link);
10     }
11     return FALSE;
12 }
```

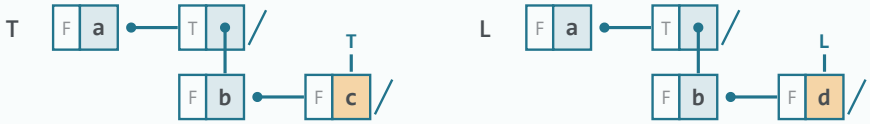
VARIABLES

- `T = &node(b)`
- `L = &node(b)`
- `x = TRUE`

CALL STACK

- `equal(b, b)`
- `equal((b,c), (b,d))`
- `equal(a, a)`
- `main()`

Code Trace: `equal(T, L)`



Both exist and their `tag` fields match: compare the contents.

Step 9 / 15

```
1  boolean equal(list_pointer T, list_pointer L) {
2      boolean x = TRUE;
3      if (!T && !L) return TRUE;
▶      if (T && L && T->tag == L->tag) {
5          if (!T->tag)              // atoms?
6              x = (T->data == L->data) ? TRUE : FALSE;
7          else                      // sublists
8              x = equal(T->dlink, L->dlink);
9          if (x == TRUE) return equal(T->link, L->link);
10     }
11     return FALSE;
12 }
```

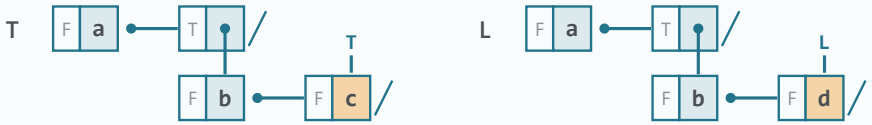
VARIABLES

`T = &node(c)` `L = &node(d)` `x = TRUE`

CALL STACK

- `equal(c, d)`
- `equal(b, b)`
- `equal((b,c), (b,d))`
- `equal(a, a)`
- `main()`

Code Trace: `equal(T, L)`



Atom: `'c' == 'd'` → `x = FALSE`.

Step 10 / 15

```
1  boolean equal(list_pointer T, list_pointer L) {
2      boolean x = TRUE;
3      if (!T && !L) return TRUE;
4      if (T && L && T->tag == L->tag) {
5          if (!T->tag)              // atoms?
▶              x = (T->data == L->data) ? TRUE : FALSE;
7          else                      // sublists
8              x = equal(T->dlink, L->dlink);
9          if (x == TRUE) return equal(T->link, L->link);
10     }
11     return FALSE;
12 }
```

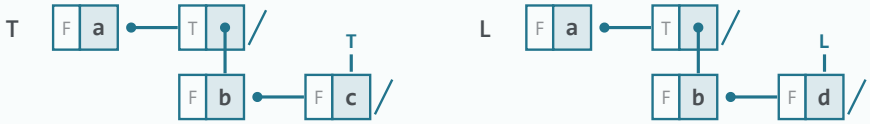
VARIABLES

`T = &node(c)` `L = &node(d)` `x = FALSE`

CALL STACK

- `equal(c, d)`
- `equal(b, b)`
- `equal((b,c), (b,d))`
- `equal(a, a)`
- `main()`

Code Trace: `equal(T, L)`



x is false: `return FALSE` ; the rest of the nodes is never examined.

Step 11 / 15

```
1  boolean equal(list_pointer T, list_pointer L) {
2      boolean x = TRUE;
3      if (!T && !L) return TRUE;
4      if (T && L && T->tag == L->tag) {
5          if (!T->tag)              // atoms?
6              x = (T->data == L->data) ? TRUE : FALSE;
7          else                      // sublists
8              x = equal(T->dlink, L->dlink);
9          if (x == TRUE) return equal(T->link, L->link);
10     }
▶      return FALSE;
12 }
```

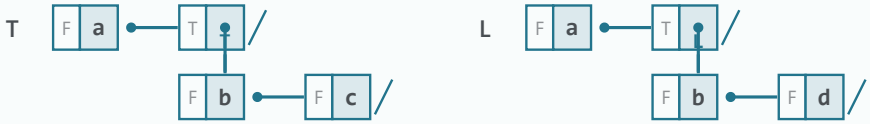
VARIABLES

`T = &node(c)` `L = &node(d)` `x = FALSE`

CALL STACK

- `equal(c, d)`
- `equal(b, b)`
- `equal((b,c), (b,d))`
- `equal(a, a)`
- `main()`

Code Trace: `equal(T, L)`



The return value is passed straight up: `FALSE`.

Step 12 / 15

```
1  boolean equal(list_pointer T, list_pointer L) {
2      boolean x = TRUE;
3      if (!T && !L) return TRUE;
4      if (T && L && T->tag == L->tag) {
5          if (!T->tag)              // atoms?
6              x = (T->data == L->data) ? TRUE : FALSE;
7          else                      // sublists
8              x = equal(T->dlink, L->dlink);
▶          if (x == TRUE) return equal(T->link, L->link);
10     }
11     return FALSE;
12 }
```

VARIABLES

`T = &node(b)` `L = &node(b)` `x = TRUE`

CALL STACK

- `equal(b, b)`
- `equal((b,c), (b,d))`
- `equal(a, a)`
- `main()`

Code Trace: `equal(T, L)`



From the sublists: `x = FALSE`.

Step 13 / 15

```
1  boolean equal(list_pointer T, list_pointer L) {
2      boolean x = TRUE;
3      if (!T && !L) return TRUE;
4      if (T && L && T->tag == L->tag) {
5          if (!T->tag)              // atoms?
6              x = (T->data == L->data) ? TRUE : FALSE;
7          else                      // sublists
▶              x = equal(T->dlink, L->dlink);
9          if (x == TRUE) return equal(T->link, L->link);
10     }
11     return FALSE;
12 }
```

VARIABLES

`T = &node((b,c))`

`L = &node((b,d))` `x = FALSE`

CALL STACK

- `equal((b,c), (b,d))`
- `equal(a, a)`
- `main()`

Code Trace: `equal(T, L)`



x is false: `return FALSE` ; the rest of the nodes is never examined.

Step 14 / 15

```
1  boolean equal(list_pointer T, list_pointer L) {
2      boolean x = TRUE;
3      if (!T && !L) return TRUE;
4      if (T && L && T->tag == L->tag) {
5          if (!T->tag)              // atoms?
6              x = (T->data == L->data) ? TRUE : FALSE;
7          else                      // sublists
8              x = equal(T->dlink, L->dlink);
9          if (x == TRUE) return equal(T->link, L->link);
10     }
▶      return FALSE;
12 }
```

VARIABLES

`T = &node((b,c))`

`L = &node((b,d))` `x = FALSE`

CALL STACK

`equal((b,c), (b,d))`

`equal(a, a)`

`main()`

Code Trace: `equal(T, L)`



The return value is passed straight up: `FALSE`.

Step 15 / 15

```
1  boolean equal(list_pointer T, list_pointer L) {
2      boolean x = TRUE;
3      if (!T && !L) return TRUE;
4      if (T && L && T->tag == L->tag) {
5          if (!T->tag)              // atoms?
6              x = (T->data == L->data) ? TRUE : FALSE;
7          else                      // sublists
8              x = equal(T->dlink, L->dlink);
▶          if (x == TRUE) return equal(T->link, L->link);
10     }
11     return FALSE;
12 }
```

VARIABLES

`T = &node(a)` `L = &node(a)` `x = TRUE`

CALL STACK

`equal(a, a)`

`main()`

A Better Version, and Shared Lists

```
bool same(list_pointer T, list_pointer L) {
    if (T == L) return true;        // same node/both NULL
    if (!T || !L || T->tag != L->tag) return false;
    if (T->tag ? !same(T->dlink, L->dlink)
               : T->data != L->data)
        return false;
    return same(T->link, L->link);
}
```

The `T == L` shortcut

If both are NULL or point to the **same** (shared) sublist, return TRUE without descending into it.

QUESTION: shared lists

The same routine also gives the right answer for non-recursive (acyclic) shared lists; the shortcut avoids walking common sublists again.

Instructor “Look, I'm finding the program's stress points. You pressed a key and it ran; that's not enough! … A proof, on the other hand, shows command of a much wider body of knowledge.”

QUESTION: Traverse the List and print Its Atoms

A common buggy version

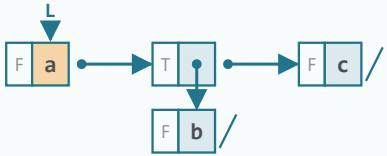
```
void print_atoms( list_pointer L ){
    if(L)
        if(!L->tag)
            printf("%c", L->data);
        else
            print(L->dlink);
        print( L->link );
}
```

Fixed

```
void print_atoms(list_pointer L) {
    if (L) {
        if (!L->tag)
            printf("%c", L->data); // print an atom
        else
            print_atoms(L->dlink);
        print_atoms(L->link);
    }
}
```

Two bugs: (1) The recursive calls use the name `print`, but the routine is called `print_atoms` (it won't compile). (2) No braces: `if(L)` covers only the inner `if-else`, so line 7 runs **even when L == NULL** → crash on `NULL→link`. The fix is to put both calls inside `if (L) { … }`.

Code Trace: `print_atoms(L)`, L = (a,(b),c)



Atom: `'a'` printed.

Step 1 / 11

```
1 void print_atoms(list_pointer L) {
2     if (L) {
3         if (!L->tag)
▶             printf("%c", L->data); // print an atom
5         else
6             print_atoms(L->dlink);
7         print_atoms(L->link);
8     }
9 }
```

VARIABLES

`L = &node(a)`

CALL STACK

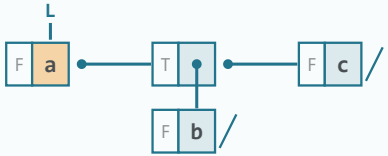
`print_atoms(a)`

`main()`

OUTPUT

```
a
```

Code Trace: `print_atoms(L)`, L = (a,(b),c)



Move to the next node on the same level (`L→link`).

Step 2 / 11

```
1 void print_atoms(list_pointer L) {
2     if (L) {
3         if (!L->tag)
4             printf("%c", L->data); // print an atom
5         else
6             print_atoms(L->dlink);
▶           print_atoms(L->link);
8     }
9 }
```

VARIABLES

`L = &node(a)`

CALL STACK

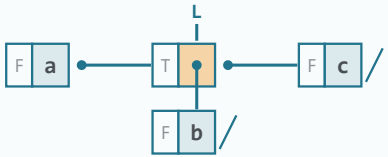
`print_atoms(a)`

`main()`

OUTPUT

```
a
```

Code Trace: `print_atoms(L)`, L = (a,(b),c)



Sublist: descend into it first (`L->dlink`).

Step 3 / 11

```
1 void print_atoms(list_pointer L) {
2     if (L) {
3         if (!L->tag)
4             printf("%c", L->data); // print an atom
5         else
▶           print_atoms(L->dlink);
7         print_atoms(L->link);
8     }
9 }
```

VARIABLES

`L = &node((b))`

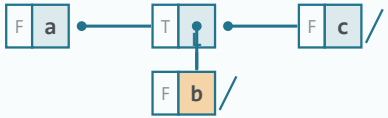
CALL STACK

- `print_atoms((b))`
- `print_atoms(a)`
- `main()`

OUTPUT

```
a
```

Code Trace: `print_atoms(L)`, L = (a,(b),c)



Atom: `'b'` printed.

Step 4 / 11

```
1 void print_atoms(list_pointer L) {
2     if (L) {
3         if (!L->tag)
▶             printf("%c", L->data); // print an atom
5         else
6             print_atoms(L->dlink);
7         print_atoms(L->link);
8     }
9 }
```

VARIABLES

`L = &node(b)`

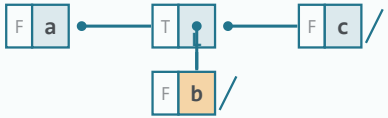
CALL STACK

- `print_atoms(b)`
- `print_atoms((b))`
- `print_atoms(a)`
- `main()`

OUTPUT

```
ab
```

Code Trace: `print_atoms(L)`, L = (a,(b),c)



Move to the next node on the same level (`L→link`).

Step 5 / 11

```
1 void print_atoms(list_pointer L) {
2     if (L) {
3         if (!L->tag)
4             printf("%c", L->data); // print an atom
5         else
6             print_atoms(L->dlink);
▶           print_atoms(L->link);
8     }
9 }
```

VARIABLES

`L = &node(b)`

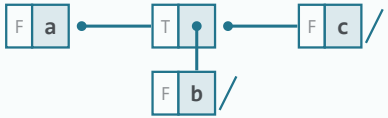
CALL STACK

- `print_atoms(b)`
- `print_atoms((b))`
- `print_atoms(a)`
- `main()`

OUTPUT

```
ab
```

Code Trace: `print_atoms(L)`, L = (a,(b),c)



`L == NULL` (end of the level): return without doing anything.

Step 6 / 11

```
1 void print_atoms(list_pointer L) {
▶       if (L) {
3           if (!L->tag)
4               printf("%c", L->data); // print an atom
5           else
6               print_atoms(L->dlink);
7           print_atoms(L->link);
8       }
9   }
```

VARIABLES

`L = NULL`

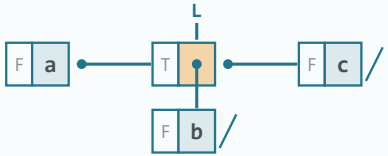
CALL STACK

- `print_atoms(NULL)`
- `print_atoms(b)`
- `print_atoms((b))`
- `print_atoms(a)`
- `main()`

OUTPUT

```
ab
```

Code Trace: `print_atoms(L)`, L = (a,(b),c)



Move to the next node on the same level (`L→link`).

Step 7 / 11

```
1 void print_atoms(list_pointer L) {
2     if (L) {
3         if (!L->tag)
4             printf("%c", L->data); // print an atom
5         else
6             print_atoms(L->dlink);
▶         print_atoms(L->link);
8     }
9 }
```

VARIABLES

`L = &node((b))`

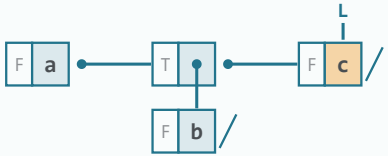
CALL STACK

- `print_atoms((b))`
- `print_atoms(a)`
- `main()`

OUTPUT

```
ab
```

Code Trace: `print_atoms(L)`, L = (a,(b),c)



Atom: `'c'` printed.

Step 8 / 11

```
1 void print_atoms(list_pointer L) {
2     if (L) {
3         if (!L->tag)
▶             printf("%c", L->data); // print an atom
5         else
6             print_atoms(L->dlink);
7         print_atoms(L->link);
8     }
9 }
```

VARIABLES

`L = &node(c)`

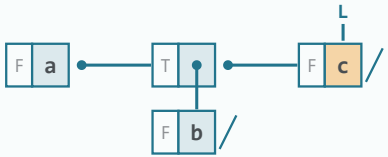
CALL STACK

- `print_atoms(c)`
- `print_atoms((b))`
- `print_atoms(a)`
- `main()`

OUTPUT

```
abc
```

Code Trace: `print_atoms(L)`, L = (a,(b),c)



Move to the next node on the same level (`L→link`).

Step 9 / 11

```
1 void print_atoms(list_pointer L) {
2     if (L) {
3         if (!L->tag)
4             printf("%c", L->data); // print an atom
5         else
6             print_atoms(L->dlink);
▶         print_atoms(L->link);
8     }
9 }
```

VARIABLES

`L = &node(c)`

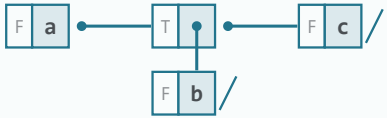
CALL STACK

- **print_atoms(c)**
- print_atoms((b))
- print_atoms(a)
- main()

OUTPUT

```
abc
```

Code Trace: `print_atoms(L)`, L = (a,(b),c)



`L == NULL` (end of the level): return without doing anything.

Step 10 / 11

```
1 void print_atoms(list_pointer L) {
▶     if (L) {
3         if (!L->tag)
4             printf("%c", L->data); // print an atom
5         else
6             print_atoms(L->dlink);
7         print_atoms(L->link);
8     }
9 }
```

VARIABLES

`L = NULL`

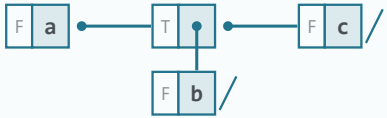
CALL STACK

- `print_atoms(NULL)`
- `print_atoms(c)`
- `print_atoms((b))`
- `print_atoms(a)`
- `main()`

OUTPUT

```
abc
```

Code Trace: `print_atoms(L)`, L = (a,(b),c)



All atoms are printed left to right regardless of depth: **abc**. Each node once: **O(n)**.

Step 11 / 11

```
1 void print_atoms(list_pointer L) {
2     if (L) {
3         if (!L->tag)
4             printf("%c", L->data); // print an atom
5         else
6             print_atoms(L->dlink);
7         print_atoms(L->link);
▶     }
9 }
```

VARIABLES

L = –

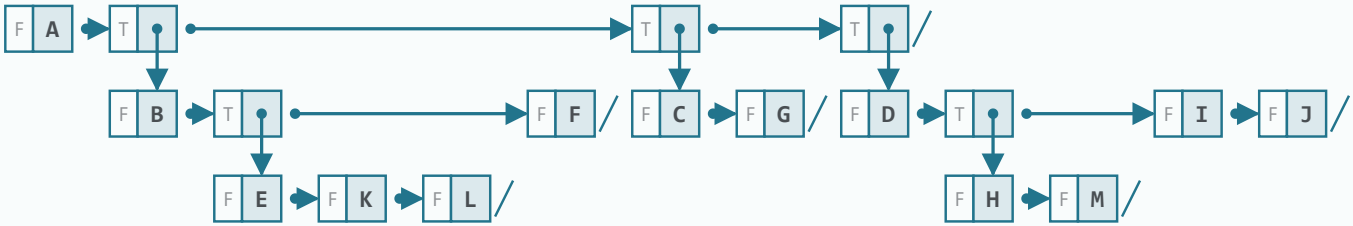
CALL STACK

main()

OUTPUT

abc

Example: A(B(E(K,L),F),C(G),D(H(M),I,J))



Each tree node is a sublist: its head is the node's label, the rest are its children. Nodes on the same level can be followed along one row.

For this list, `print_atoms` prints **ABEKLFCGDHMIJ**: the tree's preorder traversal.

QUESTION: The depth of a List, Recursively

```
int depth(list_pointer L) {
    list_pointer p = L;  int m = 0; // m: the max
    if (!L) return 0;
    while (p) {
        if (p->tag) {
            int n = depth(p->dlink);
            if (m < n) m = n;
        }
        p = p->link;
    }
    return m + 1;
}
```

Definition

depth(()) = 0; otherwise 1 + the largest depth among the sublists. Atoms add no depth.

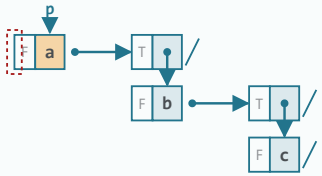
Examples

(a, b) → 1 · (a, (b, c)) → 2 · (a, (b, (c))) → 3 · (()) → 1.

Cost

Each node is visited once: O(n); the call stack grows at most as deep as the list.

Code Trace: `depth(L)`, L = (a,(b,(c)))



Level 1 call: `p = L`, `m = 0`.

Step 1 / 16

```
1  int depth(list_pointer L) {
▶      list_pointer p = L;  int m = 0; // m: the max
3      if (!L) return 0;
4      while (p) {
5          if (p->tag) {
6              int n = depth(p->dlink);
7              if (m < n) m = n;
8          }
9          p = p->link;
10     }
11     return m + 1;
12 }
```

VARIABLES

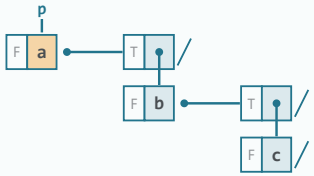
`p = &node(a)` `m = 0`

CALL STACK

`depth(a)`

`main()`

Code Trace: `depth(L)`, L = (a,(b,(c)))



'a' is an atom: no added depth.

Step 2 / 16

```
1  int depth(list_pointer L) {
2      list_pointer p = L;  int m = 0; // m: the max
3      if (!L) return 0;
4      while (p) {
▶          if (p->tag) {
6              int n = depth(p->dlink);
7              if (m < n) m = n;
8          }
9          p = p->link;
10     }
11     return m + 1;
12 }
```

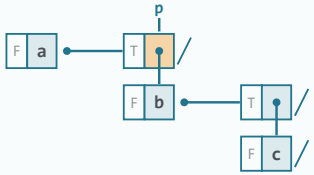
VARIABLES

`p = &node(a)` `m = 0`

CALL STACK

- `depth(a)`
- `main()`

Code Trace: `depth(L)`, L = (a,(b,(c)))



Found a sublist: its depth is computed recursively.

Step 3 / 16

```
1  int depth(list_pointer L) {
2      list_pointer p = L;  int m = 0; // m: the max
3      if (!L) return 0;
4      while (p) {
5          if (p->tag) {
▶              int n = depth(p->dlink);
7              if (m < n) m = n;
8          }
9          p = p->link;
10     }
11     return m + 1;
12 }
```

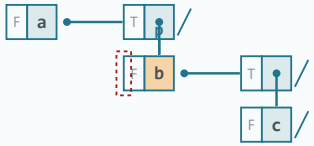
VARIABLES

`p = &node((b,(c)))` `m = 0`

CALL STACK

- `depth(a)`
- `main()`

Code Trace: `depth(L)`, L = (a,(b,(c)))



Level 2 call: `p = L`, `m = 0`.

Step 4 / 16

```
1  int depth(list_pointer L) {
▶      list_pointer p = L;  int m = 0; // m: the max
3      if (!L) return 0;
4      while (p) {
5          if (p->tag) {
6              int n = depth(p->dlink);
7              if (m < n) m = n;
8          }
9          p = p->link;
10     }
11     return m + 1;
12 }
```

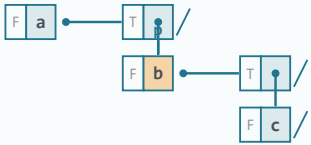
VARIABLES

`p = &node(b)` `m = 0`

CALL STACK

- `depth(b)`
- `depth(a)`
- `main()`

Code Trace: `depth(L)`, L = (a,(b,(c)))



'b' is an atom: no added depth.

Step 5 / 16

```
1  int depth(list_pointer L) {
2      list_pointer p = L;  int m = 0; // m: the max
3      if (!L) return 0;
4      while (p) {
▶          if (p->tag) {
6              int n = depth(p->dlink);
7              if (m < n) m = n;
8          }
9          p = p->link;
10     }
11     return m + 1;
12 }
```

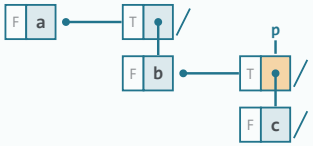
VARIABLES

`p = &node(b)` `m = 0`

CALL STACK

- `depth(b)`
- `depth(a)`
- `main()`

Code Trace: `depth(L)`, L = (a,(b,(c)))



Found a sublist: its depth is computed recursively.

Step 6 / 16

```
1  int depth(list_pointer L) {
2      list_pointer p = L;  int m = 0; // m: the max
3      if (!L) return 0;
4      while (p) {
5          if (p->tag) {
▶              int n = depth(p->dlink);
7              if (m < n) m = n;
8          }
9          p = p->link;
10     }
11     return m + 1;
12 }
```

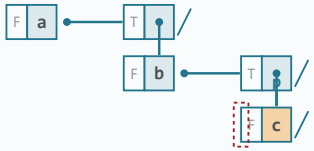
VARIABLES

`p = &node((c))` `m = 0`

CALL STACK

- `depth(b)`
- `depth(a)`
- `main()`

Code Trace: `depth(L)`, L = (a,(b,(c)))



Level 3 call: `p = L`, `m = 0`.

Step 7 / 16

```
1  int depth(list_pointer L) {
▶      list_pointer p = L;  int m = 0; // m: the max
3      if (!L) return 0;
4      while (p) {
5          if (p->tag) {
6              int n = depth(p->dlink);
7              if (m < n) m = n;
8          }
9          p = p->link;
10     }
11     return m + 1;
12 }
```

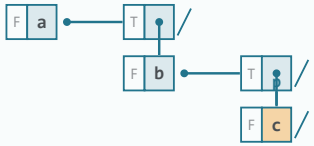
VARIABLES

`p = &node(c)` `m = 0`

CALL STACK

- depth(c)
- depth(b)
- depth(a)
- main()

Code Trace: `depth(L)`, L = (a,(b,(c)))



'c' is an atom: no added depth.

Step 8 / 16

```
1  int depth(list_pointer L) {
2      list_pointer p = L;  int m = 0; // m: the max
3      if (!L) return 0;
4      while (p) {
▶          if (p->tag) {
6              int n = depth(p->dlink);
7              if (m < n) m = n;
8          }
9          p = p->link;
10     }
11     return m + 1;
12 }
```

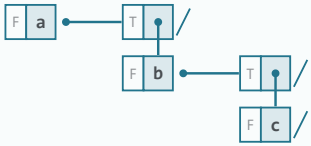
VARIABLES

`p = &node(c)` `m = 0`

CALL STACK

- `depth(c)`
- `depth(b)`
- `depth(a)`
- `main()`

Code Trace: `depth(L)`, L = (a,(b,(c)))



`p = p⟶link` → NULL: this level is done.

Step 9 / 16

```
1  int depth(list_pointer L) {
2      list_pointer p = L;  int m = 0; // m: the max
3      if (!L) return 0;
4      while (p) {
5          if (p->tag) {
6              int n = depth(p->dlink);
7              if (m < n) m = n;
8          }
▶          p = p->link;
10     }
11     return m + 1;
12 }
```

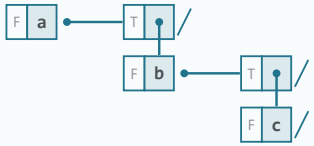
VARIABLES

`p = NULL` `m = 0`

CALL STACK

- `depth(c)`
- `depth(b)`
- `depth(a)`
- `main()`

Code Trace: `depth(L)`, L = (a,(b,(c)))



`return m + 1 = 1` : the deepest sublist + this level.

Step 10 / 16

```
1  int depth(list_pointer L) {
2      list_pointer p = L;  int m = 0; // m: the max
3      if (!L) return 0;
4      while (p) {
5          if (p->tag) {
6              int n = depth(p->dlink);
7              if (m < n) m = n;
8          }
9          p = p->link;
10     }
▶      return m + 1;
12 }
```

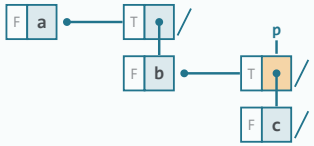
VARIABLES

`p = NULL` `m = 0`

CALL STACK

- `depth(c)`
- `depth(b)`
- `depth(a)`
- `main()`

Code Trace: `depth(L)`, L = (a,(b,(c)))



The sublist returned `n = 1` → `m = 1`.

Step 11 / 16

```
1  int depth(list_pointer L) {
2      list_pointer p = L;  int m = 0; // m: the max
3      if (!L) return 0;
4      while (p) {
5          if (p->tag) {
6              int n = depth(p->dlink);
▶              if (m < n) m = n;
8          }
9          p = p->link;
10     }
11     return m + 1;
12 }
```

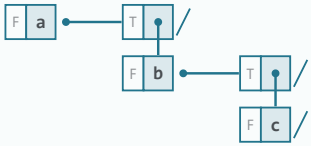
VARIABLES

`p = &node((c))` `m = 1` `n = 1`

CALL STACK

- depth(b)
- depth(a)
- main()

Code Trace: `depth(L)`, L = (a,(b,(c)))



`p = p⟶link` → NULL: this level is done.

Step 12 / 16

```
1  int depth(list_pointer L) {
2      list_pointer p = L;  int m = 0; // m: the max
3      if (!L) return 0;
4      while (p) {
5          if (p->tag) {
6              int n = depth(p->dlink);
7              if (m < n) m = n;
8          }
▶          p = p->link;
10     }
11     return m + 1;
12 }
```

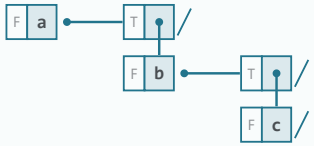
VARIABLES

`p = NULL` `m = 1`

CALL STACK

- `depth(b)`
- `depth(a)`
- `main()`

Code Trace: `depth(L)`, L = (a,(b,(c)))



`return m + 1 = 2` : the deepest sublist + this level.

Step 13 / 16

```
1  int depth(list_pointer L) {
2      list_pointer p = L;  int m = 0; // m: the max
3      if (!L) return 0;
4      while (p) {
5          if (p->tag) {
6              int n = depth(p->dlink);
7              if (m < n) m = n;
8          }
9          p = p->link;
10     }
▶      return m + 1;
12 }
```

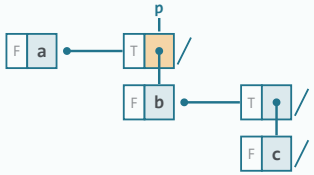
VARIABLES

`p = NULL` `m = 1`

CALL STACK

- `depth(b)`
- `depth(a)`
- `main()`

Code Trace: `depth(L)`, L = (a,(b,(c)))



The sublist returned `n = 2` → `m = 2`.

Step 14 / 16

```
1  int depth(list_pointer L) {
2      list_pointer p = L;  int m = 0; // m: the max
3      if (!L) return 0;
4      while (p) {
5          if (p->tag) {
6              int n = depth(p->dlink);
▶              if (m < n) m = n;
8          }
9          p = p->link;
10     }
11     return m + 1;
12 }
```

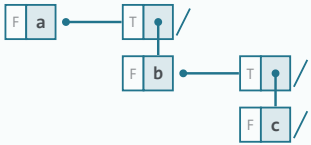
VARIABLES

`p = &node((b,(c)))` `m = 2` `n = 2`

CALL STACK

- `depth(a)`
- `main()`

Code Trace: `depth(L)`, L = (a,(b,(c)))



`p = p⟶link` → NULL: this level is done.

Step 15 / 16

```
1  int depth(list_pointer L) {
2      list_pointer p = L;  int m = 0; // m: the max
3      if (!L) return 0;
4      while (p) {
5          if (p->tag) {
6              int n = depth(p->dlink);
7              if (m < n) m = n;
8          }
▶          p = p->link;
10     }
11     return m + 1;
12 }
```

VARIABLES

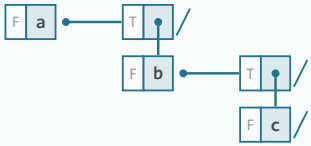
`p = NULL` `m = 2`

CALL STACK

`depth(a)`

`main()`

Code Trace: `depth(L)`, L = (a,(b,(c)))



`return m + 1 = 3` : the deepest sublist + this level.

Step 16 / 16

```
1  int depth(list_pointer L) {
2      list_pointer p = L;  int m = 0; // m: the max
3      if (!L) return 0;
4      while (p) {
5          if (p->tag) {
6              int n = depth(p->dlink);
7              if (m < n) m = n;
8          }
9          p = p->link;
10     }
▶      return m + 1;
12 }
```

VARIABLES

`p = NULL` `m = 2`

CALL STACK

- `depth(a)`
- `main()`

QUESTION: depth, Iteratively (a Student Solution)

```
int depth( tree_pointer L ){
    tree_pointer pop(); void push(tree_pointer);
    int max = 0;
    for( int current = 1, p = L ; p ; p->link ){
        if( p->tag ){ // enter the sublist
            push(p);
            p = p -> dlink;
            current++;
        }
        if( !p->link ){   // end of list, go up
            p = pop();
            if( max < current ) max = current;
            current--;
        }
    }
    return max;
}
```

Compiling / the loop

`int current = 1, p = L` makes p an `int`; the update `p→link` is not an assignment → an infinite loop.

Edge cases

It dereferences NULL on an empty sublist, pops an empty stack, and never moves on to T's sibling.

Instructor "This isn't going to be written as a loop that tracks current and max. That's what a first-year student writes. … And global variables are forbidden!"

Iterative depth with a Local Stack

```
int iter_depth(list_pointer L) {
    list_pointer stack[100];   // local stack (depth<100)
    int top = 0, d = 1, m = 0;
    list_pointer p = L;
    if (!L) return 0;
    for (;;) {
        if (p == NULL) {              // level done
            if (top == 0) return m;
            p = stack[--top]->link;  // the node after T
            d--;
        } else {
            if (d > m) m = d;
            if (p->tag && p->dlink) {
                stack[top++] = p;    // descend
                p = p->dlink;
                d++;
            } else p = p->link;
        }
    }
}
```

Method

When descending into a sublist, push its T node; when the level ends, pop it and move on to its **sibling**.

Rules

No globals: the stack and counters are local. The empty sublist and the empty list are handled separately.

Verification

Checked by simulation to match the recursive version on six sample lists.

On Thinking Recursively

Instructor “If it were up to me, I'd give this course 15 credits instead of 3, just to build the foundations of recursive thinking.”

Instructor “While eating, walking, watching TV, you have to keep asking yourself: ‘If I do this, what do I return?’ … And use the debugger to see what is returned where.”

Code that works “The skill of writing code that works on the given example and crashes on every other one! Bravo, really! Code has to work correctly on different inputs, including the edge cases.”

The original book “Get 2–3 books for every course. A book you paid for gets read; a photocopy doesn't.”

Edge-case lists: (), (a), (()), ((())), a long one-level list, a deep single chain.

QUESTION: invert an Unshared List — L = (a,(b,c)) → ((c,b),a)

```
void invert(list_pointer *L) {
    list_pointer r, p = *L, q = NULL;
    while (p) {
        if (p->tag)                  // not an atom
            invert(&p->dlink);
        r = q;
        q = p;
        p = p->link;
        q->link = r;
    }
    *L = q;
}
```

List inversion + 2 lines

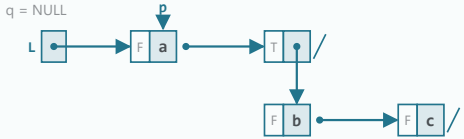
The three pointers of the singly linked `list_invert` routine, unchanged; the addition: if the node is a sublist, invert it first. Use: mirroring an image.

Why `&p→dlink`?

The sublist's head changes; the T node's `dlink` field is updated by the inner call's `*L = q` line before the call returns.

Instructor "If we didn't solve it this way, it would be brute force. That inverts it too, but by bulldozing everything in its path."

Code Trace: `invert(&L)`, (a,(b,c)) → ((c,b),a)



`p = *L` → &node(a), `q = NULL`.

Step 1 / 12

```
1  void invert(list_pointer *L) {
▶       list_pointer r, p = *L, q = NULL;
3       while (p) {
4           if (p->tag)                 // not an atom
5               invert(&p->dlink);
6           r = q;
7           q = p;
8           p = p->link;
9           q->link = r;
10      }
11      *L = q;
12  }
```

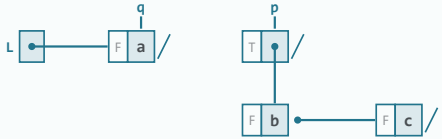
VARIABLES

`r = ?` `q = NULL` `p = &node(a)` `*L = &node(a)`

CALL STACK

- `invert(&list)`
- `main()`

Code Trace: `invert(&L)`, (a,(b,c)) → ((c,b),a)



`r = q; q = p; p = p→link; q→link = r;` → a's link is reversed.

Step 2 / 12

```
1  void invert(list_pointer *L) {
2      list_pointer r, p = *L, q = NULL;
3      while (p) {
4          if (p->tag)                 // not an atom
5              invert(&p->dlink);
▶          r = q;
▶          q = p;
▶          p = p->link;
▶          q->link = r;
10     }
11     *L = q;
12 }
```

VARIABLES

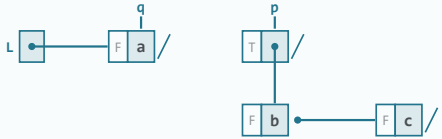
`r = NULL` `q = &node(a)` `p = &node((b,c))` `*L = &node(a)`

CALL STACK

`invert(&list)`

`main()`

Code Trace: `invert(&L)`, (a,(b,c)) → ((c,b),a)



The node is a sublist: invert it first (via the address `&p→dlink`).

Step 3 / 12

```
1  void invert(list_pointer *L) {
2      list_pointer r, p = *L, q = NULL;
3      while (p) {
4          if (p->tag)                  // not an atom
▶              invert(&p->dlink);
6          r = q;
7          q = p;
8          p = p->link;
9          q->link = r;
10     }
11     *L = q;
12 }
```

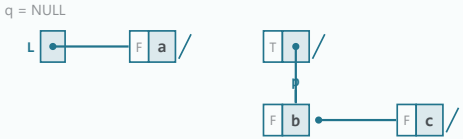
VARIABLES

`r = NULL` `q = &node(a)` `p = &node((b,c))` `*L = &node(a)`

CALL STACK

- `invert(&list)`
- `main()`

Code Trace: `invert(&L)`, (a,(b,c)) → ((c,b),a)



`p = *L` → &node(b), `q = NULL`.

Step 4 / 12

```
1  void invert(list_pointer *L) {
▶       list_pointer r, p = *L, q = NULL;
3       while (p) {
4           if (p->tag)                  // not an atom
5               invert(&p->dlink);
6           r = q;
7           q = p;
8           p = p->link;
9           q->link = r;
10      }
11      *L = q;
12  }
```

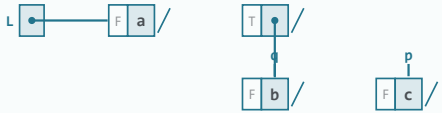
VARIABLES

`r = ?` `q = NULL` `p = &node(b)` `*L = &node(b)`

CALL STACK

- `invert(&p→dlink)`
- `invert(&list)`
- `main()`

Code Trace: `invert(&L)`, (a,(b,c)) → ((c,b),a)



`r = q; q = p; p = p→link; q→link = r;` → b's link is reversed.

Step 5 / 12

```
1  void invert(list_pointer *L) {
2      list_pointer r, p = *L, q = NULL;
3      while (p) {
4          if (p->tag)                 // not an atom
5              invert(&p->dlink);
▶          r = q;
▶          q = p;
▶          p = p->link;
▶          q->link = r;
10     }
11     *L = q;
12 }
```

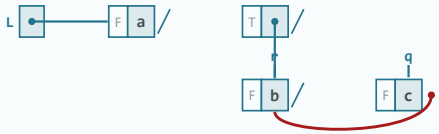
VARIABLES

`r = NULL` `q = &node(b)` `p = &node(c)` `*L = &node(b)`

CALL STACK

- `invert(&p→dlink)`
- `invert(&list)`
- `main()`

Code Trace: `invert(&L)`, (a,(b,c)) → ((c,b),a)



`r = q; q = p; p = p→link; q→link = r;` → c's link is reversed.

Step 6 / 12

```
1  void invert(list_pointer *L) {
2      list_pointer r, p = *L, q = NULL;
3      while (p) {
4          if (p->tag)                  // not an atom
5              invert(&p->dlink);
▶          r = q;
▶          q = p;
▶          p = p->link;
▶          q->link = r;
10     }
11     *L = q;
12 }
```

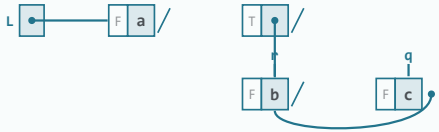
VARIABLES

`r = &node(b)` `q = &node(c)` `p = NULL` `*L = &node(b)`

CALL STACK

- `invert(&p→dlink)`
- `invert(&list)`
- `main()`

Code Trace: `invert(&L)`, (a,(b,c)) → ((c,b),a)



`p == NULL` : this level is done.

Step 7 / 12

```
1  void invert(list_pointer *L) {
2      list_pointer r, p = *L, q = NULL;
▶      while (p) {
4          if (p->tag)                  // not an atom
5              invert(&p->dlink);
6          r = q;
7          q = p;
8          p = p->link;
9          q->link = r;
10     }
11     *L = q;
12 }
```

VARIABLES

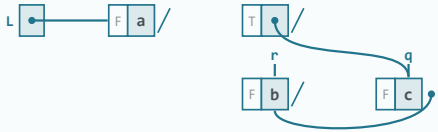
`r = &node(b)` `q = &node(c)`

`p = NULL` `*L = &node(b)`

CALL STACK

- `invert(&p→dlink)`
- `invert(&list)`
- `main()`

Code Trace: `invert(&L)`, (a,(b,c)) → ((c,b),a)



`*L = q` : c is this level's new head.

Step 8 / 12

```
1  void invert(list_pointer *L) {
2      list_pointer r, p = *L, q = NULL;
3      while (p) {
4          if (p->tag)                  // not an atom
5              invert(&p->dlink);
6          r = q;
7          q = p;
8          p = p->link;
9          q->link = r;
10     }
▶      *L = q;
12 }
```

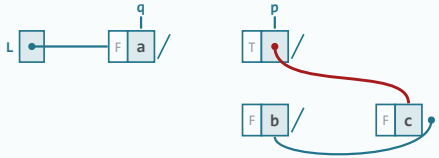
VARIABLES

`r = &node(b)` `q = &node(c)` `p = NULL` `*L = &node(c)`

CALL STACK

- `invert(&p→dlink)`
- `invert(&list)`
- `main()`

Code Trace: `invert(&L)`, (a,(b,c)) → ((c,b),a)



Back from the sublist: the T node's `dlink` already points to the new head.

Step 9 / 12

```
1  void invert(list_pointer *L) {
2      list_pointer r, p = *L, q = NULL;
3      while (p) {
4          if (p->tag)                  // not an atom
▶              invert(&p->dlink);
6          r = q;
7          q = p;
8          p = p->link;
9          q->link = r;
10     }
11     *L = q;
12 }
```

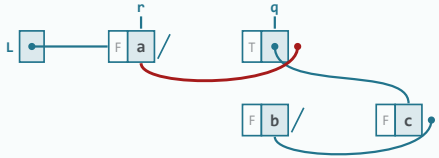
VARIABLES

`r = NULL` `q = &node(a)` `p = &node((c,b))` `*L = &node(a)`

CALL STACK

- `invert(&list)`
- `main()`

Code Trace: `invert(&L)`, (a,(b,c)) → ((c,b),a)



`r = q; q = p; p = p→link; q→link = r;` → (c,b)'s link is reversed.

Step 10 / 12

```
1  void invert(list_pointer *L) {
2      list_pointer r, p = *L, q = NULL;
3      while (p) {
4          if (p->tag)                  // not an atom
5              invert(&p->dlink);
▶          r = q;
▶          q = p;
▶          p = p->link;
▶          q->link = r;
10     }
11     *L = q;
12 }
```

VARIABLES

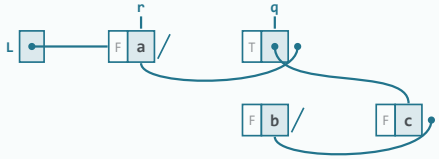
`r = &node(a)` `q = &node((c,b))` `p = NULL` `*L = &node(a)`

CALL STACK

`invert(&list)`

`main()`

Code Trace: `invert(&L)`, (a,(b,c)) → ((c,b),a)



`p == NULL` : this level is done.

Step 11 / 12

```
1  void invert(list_pointer *L) {
2      list_pointer r, p = *L, q = NULL;
▶      while (p) {
4          if (p->tag)                  // not an atom
5              invert(&p->dlink);
6          r = q;
7          q = p;
8          p = p->link;
9          q->link = r;
10     }
11     *L = q;
12 }
```

VARIABLES

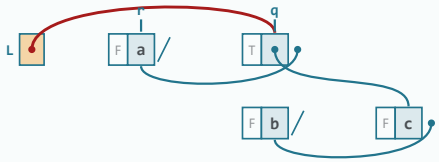
`r = &node(a)` `q = &node((c,b))` `p = NULL` `*L = &node(a)`

CALL STACK

`invert(&list)`

`main()`

Code Trace: `invert(&L)`, (a,(b,c)) → ((c,b),a)



`*L = q` : (c,b) is this level's new head.

Step 12 / 12

```
1  void invert(list_pointer *L) {
2      list_pointer r, p = *L, q = NULL;
3      while (p) {
4          if (p->tag)                  // not an atom
5              invert(&p->dlink);
6          r = q;
7          q = p;
8          p = p->link;
9          q->link = r;
10     }
▶      *L = q;
12 }
```

VARIABLES

`r = &node(a)` `q = &node((c,b))`

`p = NULL` `*L = &node((c,b))`

CALL STACK

- `invert(&list)`
- `main()`

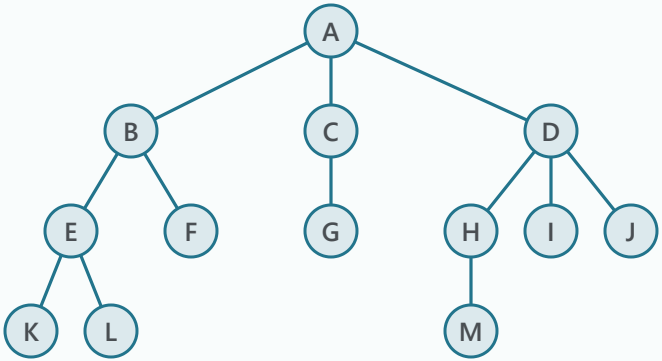
Data Structures: The Backbone

Job interview “Have you taken data structures?” “Yes, I have.” “So have I. That's lists, trees and so on, right?” … It's like saying: “Brain surgery? Oh, that's the blood vessels, right?” As if it were that simple. Data structures are the backbone of computer science. Knowing what a list or a tree is doesn't mean you've learned data structures. You have to be able to write the routines that build them.

Reading: Theory and Mathematics “Here, students get programming assignments as soon as they start their first year… the upside is writing code fast and finding a job fast. But there's a downside: theory is neglected… Yet you need a very good mathematical background: discrete math, statistics, stochastic processes… Richard Johnsonbaugh's Discrete Mathematics: get it and read it. Skip it and you'll still pass your courses, but you won't learn this craft.”

Reading “A department has limits; it can't give you everything! A student should realize this by the second year and take courses from the mathematics and statistics departments…”

Building a Tree Structure with a Generalized List



`A(B(E(K,L),F),C(G),D(H(M),I,J))`

Notation

`C(G)` : C is G's parent.

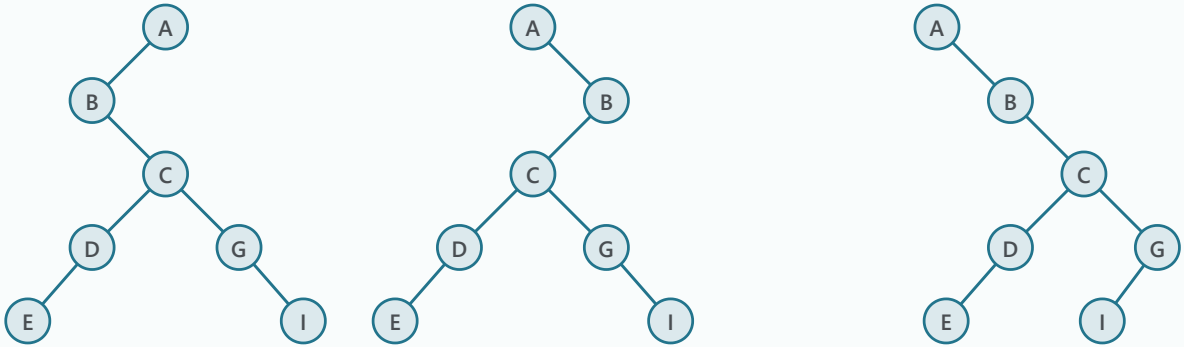
`A(B…,C…,D…)` : A is the parent of B, C, and D.

Flexibility

For this much data a one-line string is enough. Dots instead of commas, braces instead of parentheses: the notation is up to you.

Instructor “If you can go from this to that and back, you're fine; nobody can see the structure inside.”

Drawback: No Left/right Distinction



All three trees have the same linear form, A(B(C(D(E),G(I)))): there is no way to write whether B is a left or a right child.

The problem

For a general tree the order of children is enough, but in a binary tree the distinction between **left** and **right** children is lost.

The fix

Store L/R information with each node (or write a special symbol for an empty child): A(B(,C(...))).

Assignments

1. **Assignment 1:** Draw the memory layout and the tree structure of the list `A(B(E(H,I(J,K)),F),C(G),D)`.
2. **Assignment 2:** Write a routine that, given a tree structure, outputs its list notation.

▶ **Suggested solutions**

Shared Lists and Reference Counting

```
// header ref: how many pointers point to this list
void list_free(list_pointer *L) {
    list_pointer p, q;
    if (*L == NULL) return;
    if (--(*L)->ref > 0) {      // still in use
        *L = NULL; return;
    }
    p = (*L)->link;             // first real node
    free(*L);
    while (p) {
        if (p->tag) list_free(&p->dlink);
        q = p; p = p->link; free(q);
    }
    *L = NULL;
}
```

Horowitz–Sahni, Section 4.9 (header nodes with reference counts).

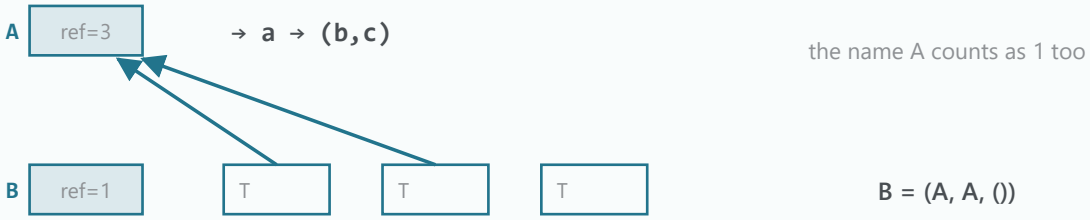
Header node

Every list starts with a header node; `ref` is the number of pointers pointing to it.

The release rule

A list is not freed until its count reaches 0; when it does, its nodes and sublists are released by the same rule.

Reference Counting Step by Step: B = (A, A, ())



Each list starts with a **header node**; `ref` is the number of pointers to it. A is pointed to by the name A and by two of B's elements: 3.

Step 1 / 4

The same idea works for copying: a shared sublist isn't copied again; its count is incremented.

Reference Counting Step by Step: B = (A, A, ())



`list_free(&B)` : B's count goes 1 → 0, and B's nodes are released one by one.

Step 2 / 4

The same idea works for copying: a shared sublist isn't copied again; its count is incremented.

Reference Counting Step by Step: B = (A, A, ())



B's two T nodes call `list_free` on A: 3 → 2 → 1. A is still in use, so it is not freed.

Step 3 / 4

The same idea works for copying: a shared sublist isn't copied again; its count is incremented.

Reference Counting Step by Step: B = (A, A, ())



Then `list_free(&A)`: 1 → 0, and A is freed. Sharing handled safely.

Step 4 / 4

The same idea works for copying: a shared sublist isn't copied again; its count is incremented.

The Limits of Reference Counting, and Garbage Collection

The cycle problem

C = (a, C): C points to itself, so its count never reaches 0; even if the name C is deleted, the list stays in memory (a leak).

Mark and sweep

(1) Every node reachable from the root pointers (variables) is **marked**; (2) memory is scanned and unmarked nodes are **swept** (returned to avail).

The cost

The program pauses while the collector runs, and the mark bit needs space. LISP, Java, and Python use variants of this idea.

In C

No automatic garbage collection: copying, sharing, and freeing are the programmer's responsibility.

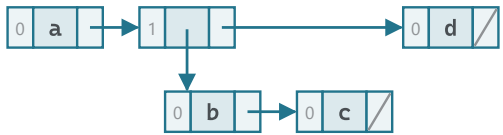
Lab: List Depth

(a, (b, c), d)

Input (a list, e.g. (a, (b, c), d)) `(a, (b, c), d)` Apply

CODE

```
 1 int depth(list_pointer L) {
 2     list_pointer p = L;  int m = 0;  // the …
 3     if (!L) return 0;
 4     while (p) {
 5         if (p->tag) {
 6             int n = depth(p->dlink);
 7             if (m < n) m = n;
 8         }
 9         p = p->link;
10     }
11     return m + 1;
12 }
```



`call depth 0`

OUTPUT (none yet)

`depth(L)` : each level gets its own recursive call. An element whose tag is 1 points to a sublist; the depth of that sublist is computed separately.

Exam Question

Question. A field `int count` has been added to each node. Write a **recursive** routine that stores, in the count of every node whose tag is TRUE, the number of nodes below it, and returns the total number of nodes.

▶ **Sample solution**

Exam Question

Question. Write a recursive routine that prints a generalized list in memory in its **linear form**.

▶ **Solution**

Exam Question

Question. Draw the generalized list corresponding to the linear form `L=(d,(c,(e,(m,k),(()),z,y,(x,(r),b,a,()))))`. Consecutive elements on the same level must be traceable along one row.

▶ **Solution**

WRAP-UP

Complexity Summary

Routine	Time	Extra memory	Note
copy	O(n)	O(depth + length)*	no sharing, no recursion
equal / same	O(n)	O(depth + length)*	stops at the first difference
print_atoms	O(n)	O(depth + length)*	preorder
depth (recursive)	O(n)	O(depth)	loop over a level, recursion into sublists
iter_depth (iterative)	O(n)	O(depth)	local stack
invert	O(n)	O(depth)	in place, no malloc
array_insert / array_delete	O(n)	O(1)	shifting
ll_insert (position i)	O(i)	O(1)	no shifting

n: total number of nodes. * For routines that also recurse through `link`, the stack depth also depends on the number of nodes on a level; using a loop for `link` brings it down to O(depth).

WRAP-UP

Common Mistakes

1. Treating a shared list as unshared: copying it, or freeing it twice.
2. Calling copy/equality on a recursive list (C = (a,C)): infinite recursion.
3. Forgetting the braces on `if`s and recursing on NULL too (`print_atoms`).
4. Passing the list by value to a routine that changes its head (inversion needs `list_pointer *L`).
5. Confusing the empty sublist `()` with the empty list NULL: the T node exists, its dlink is NULL.
6. Counting depth in a global variable; the routine breaks once it becomes part of another component.

Exercises

1. Write a routine that counts the atoms in a generalized list.
2. Write a routine that reads the linear form (`"(a,(b,c))"`) and builds the list.
3. How would you change `copy` so it preserves sharing?
4. Write `contains(L, x)`, which tells whether atom x occurs in the list.
5. Rewrite the array-based list so that it doubles when full.

▶ **Answers / hints**

Instructor “The routine that builds the list is a quiz question! Keep asking yourself how you'd write the routine that builds the list, and the one that copies it!”

WRAP-UP

Chapter Summary

The list ADT

The contract is fixed; the implementation can change: arrays give fast access, links give fast insertion and deletion.

Generalized lists

A node is an atom or a sublist: [tag | data/dlink | link].

The recursive pattern

Process an atom, recurse into a sublist, then continue along `link`.

Depth and inversion

The recursive versions are short and correct; the iterative ones need a local stack.

Trees

A one-line linear form; for binary trees, left/right information must be added.

Sharing

Copying and freeing have pitfalls; reference counting fails on cycles, so garbage collection is needed.

References and Next Steps

- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd ed., Section 4.9 (generalized lists, reference counts).
- T. H. Cormen et al., *Introduction to Algorithms*, Chapter 10 (list implementations).
- R. Johnsonbaugh, *Discrete Mathematics* (recommended in the reading).

← Chapter 8: Stack and Queue Applications

Course page

Chapter 10: Trees

Singly linked list routines and the list-freeing exam question: Linked Lists.

The End

Chapter 9: The List ADT and Generalized Lists

Next chapter: Chapter 10: Trees · Course page