Linked Lists Revisited

Manolis Koubarakis

A List ADT

- A **list** *L* of items of type *T* is a sequence of items of type *T* on which the following operations are defined:
  - **Initialize** the list *L* to be the empty list.
  - Determine whether or not the list *L* is **empty**.
  - Find the length of a list *L* (where the **length** of *L* is the number of items in *L* and the length of the empty list is 0).
  - **Select** the i-th item of a list *L*, where $1 \leq i \leq length(L)$.
  - **Replace** the i-th item *X* of a list *L* with a new item *Y* where $1 \leq i \leq length(L)$.
  - **Delete** any item *X* from a nonempty list *L*.
  - **Insert** a new item *X* into a list *L* in any arbitrary position (such as before the first item of *L*, after the last item of *L* or between any two items of *L*).

Lists

- Lists are more general kinds of containers than stacks and queues.
- Lists can be represented by **sequential representations** and **linked representations**.

Sequential List Representations

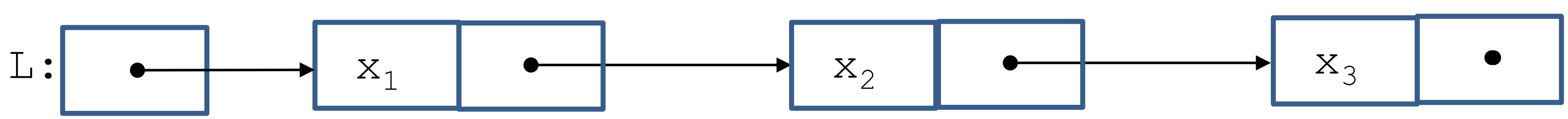
- We can use an array `A[0:MaxSize-1]` as we show graphically (items are stored **contiguously**):

Advantages and Disadvantages

- Advantages:
  - Fast access to the i-th item of the list in *O(1)* time.
- Disadvantages:
  - Insertions and deletions may require shifting all items i.e., an *O(n)* cost on the average.
  - The size of the array should be known in advance. So if we have small size, we run the risk of overflow and if we have large size, we will be wasting space.

One-Way Linked Lists Representation

- We can use chains of linked nodes as shown below:



Declaring Data Types for Linked Lists

The following statements declare appropriate data types for our linked lists from earlier lectures:

```
typedef char AirportCode[4];
typedef struct NodeTag {
                        AirportCode Airport;
                        struct NodeTag *Link;
                } NodeType;
typedef NodeType *NodePointer;
```

We can now define variables of these datatypes:

```
NodePointer L;
```

or equivalently

```
NodeType *L;
```

Accessing the *i*^th^ Item

```
void PrintItem(int i, NodeType *L)
{
    while ((i > 1) && (L != NULL)){
        L=L->Link;
        i--;
     }
     if ((i == 1) && (L != NULL)){
        printf("%s", L->Item);
     } else {
        printf("Error - attempt to print an
item that is not on the list.\n");
     }
}
```

Computational Complexity

- Suppose that list L has n items.
- The **worst-case time complexity** of the previous algorithm is $O(n)$. This happens when we are accessing the last element of the list or when the element is not in the list.

Computational Complexity (cont'd)

- If it is equally likely that each of the n items in the list can be accessed, then the average number of pointers followed to access the i^{th} item is:

$$Average = \frac{(1 + 2 + \cdots + n)}{n} = \frac{\frac{n(n+1)}{2}}{n} = \frac{n}{2} + \frac{1}{2}$$

- Therefore, the **average time complexity** to access the i^{th} item is $O(n)$ too.
- These complexity bounds are the same for inserting before or after the i^{th} item or deleting it or replacing it.

Comparing Sequential and Linked List Representations

List Operation	Sequential Representation	Linked List Representation
Finding the length of *L*	$O(1)$	$O(n)$
Inserting a new first item	$O(n)$	$O(1)$
Deleting the last item	$O(1)$	$O(n)$
Replacing the i^{th} item	$O(1)$	$O(n)$
Deleting the i^{th} item	$O(n)$	$O(n)$

The above table gives worst-case time complexity. The average case is the same.

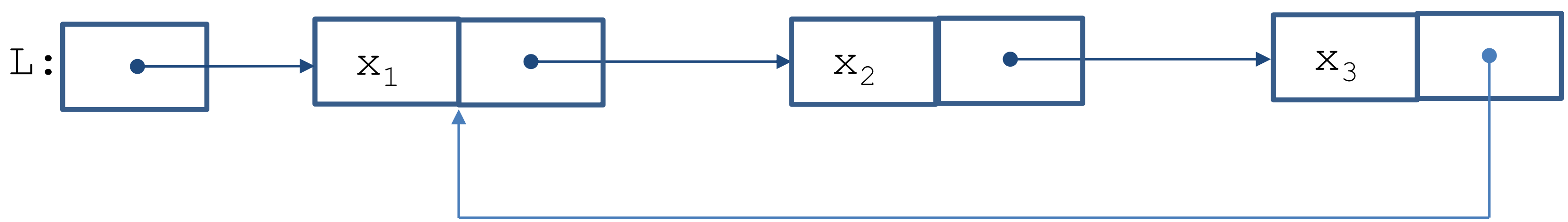
Other Linked List Representations

- Circular linked lists
- Two-way linked lists
- Linked lists with header and rear nodes

Circular Linked Lists

- A **circular linked list** is formed by having the link in the last node of a one-way linked list point back to the first node.

- The advantage of a circular linked list is that any node on it is accessible by any other node.



- In essence, **there is no first or last node in a circular list.** Only a pointer that allows us to access the elements of the list which is called a **cursor** (pointer `L` above).

The Josephus Problem

- We imagine that *N* people have decided to elect a leader by arranging themselves in a circle and eliminating every *Mth* person around the circle, closing ranks as each person drops out.
- The problem is to find out which person will be the last one remaining.
- A mathematically inclined potential leader will figure out ahead of time which position in the circle to take.
- The identity of the elected leader is a function of *N* and *M* that we refer to as the **Josephus function**.

Program for the Josephus Problem

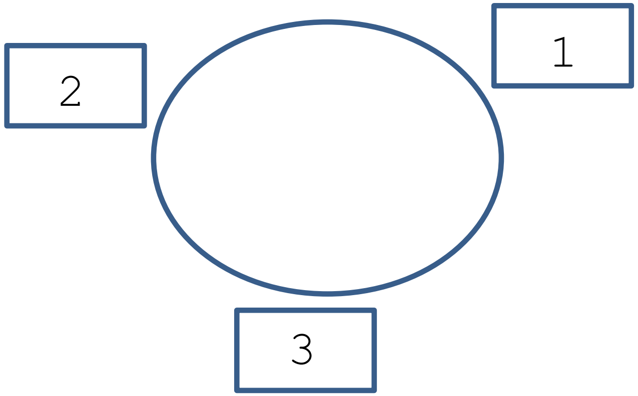
```
#include <stdlib.h>
typedef struct node* link;
struct node { int item; link next; };

main(int argc, char *argv[])
  { int i, N = atoi(argv[1]), M = atoi(argv[2]);

    link t = malloc(sizeof *t), x = t;
    t->item = 1; t->next = t;

    for (i = 2; i <= N; i++) {
       x = (x->next = malloc(sizeof *x));
       x->item = i; x->next = t;
    }

    while (x != x->next) {
       for (i = 1; i < M; i++) x = x->next;
       x->next = x->next->next; N--;
    }

    printf("%d\n", x->item);
  }
```

Program for the Josephus Problem (cont'd)

- To represent people arranged in a circle, we build a circular linked list, with a link form each person to the person on the left in the circle.
- The integer `i` represents the `i`th person in the circle.
- After building a one-node circular list for `1`, we insert `2` through `N` after that node, resulting in a circle with `1` through `N`, leaving `x` pointing to `N`.
- Then, we skip `M-1` nodes, beginning with `1`, and set the link of the `(M-1)`st to skip the `M`th, continuing until only one node is left.

Running the Program

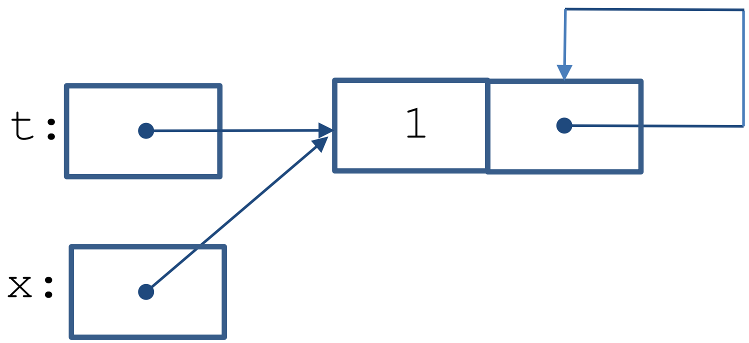
- Let us run the previous program with `N=3` and `M=2`.



- Who will be the leader if the cursor points at 3 and we traverse the list in reverse clockwise mode?

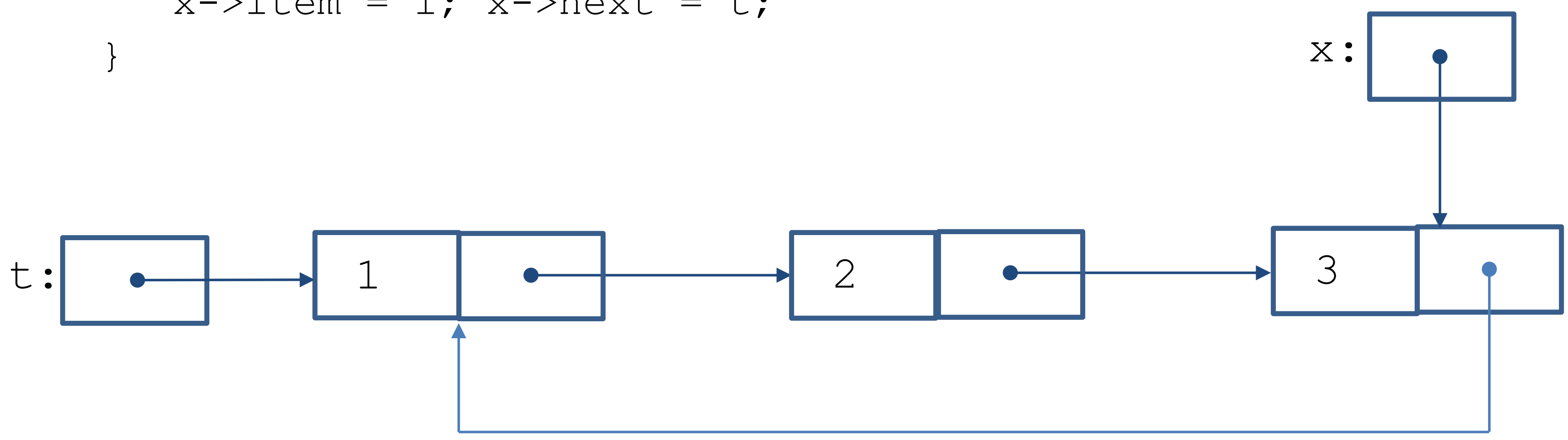
Running the Program (cont’d)

```
link t = malloc(sizeof *t), x = t;
t->item = 1; t->next = t;
```



Running the Program (cont'd)

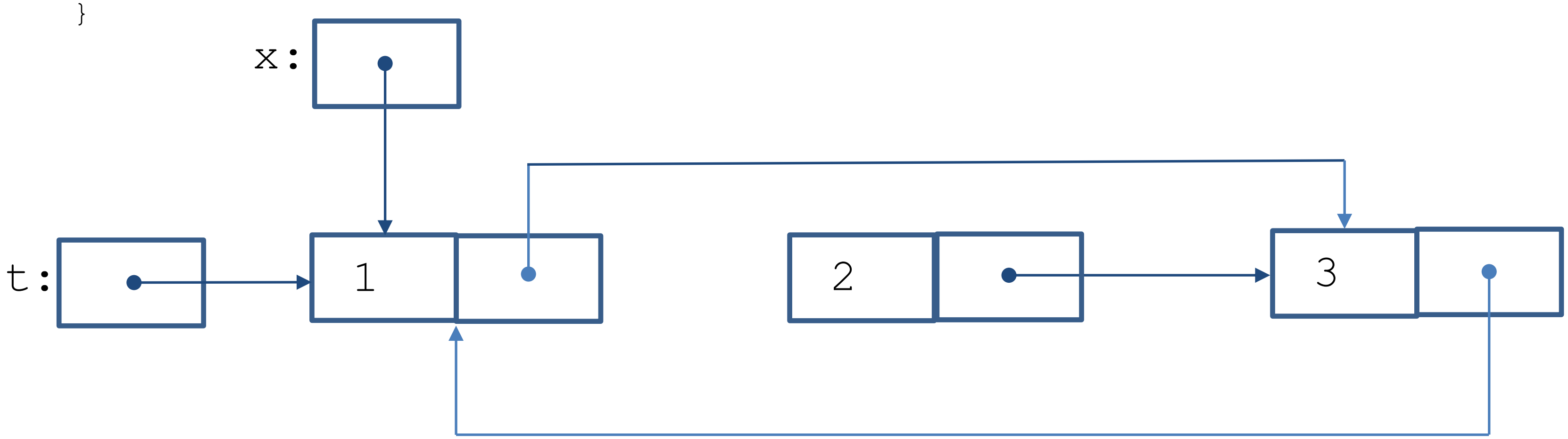
```
for (i = 2; i <= N; i++) {
        x = (x->next = malloc(sizeof *x));
        x->item = i; x->next = t;
    }
```



- The above figure shows the circular list of the previous slide after the `for` loop is executed for `i=2` and `i=3`.

Running the Program (cont'd)

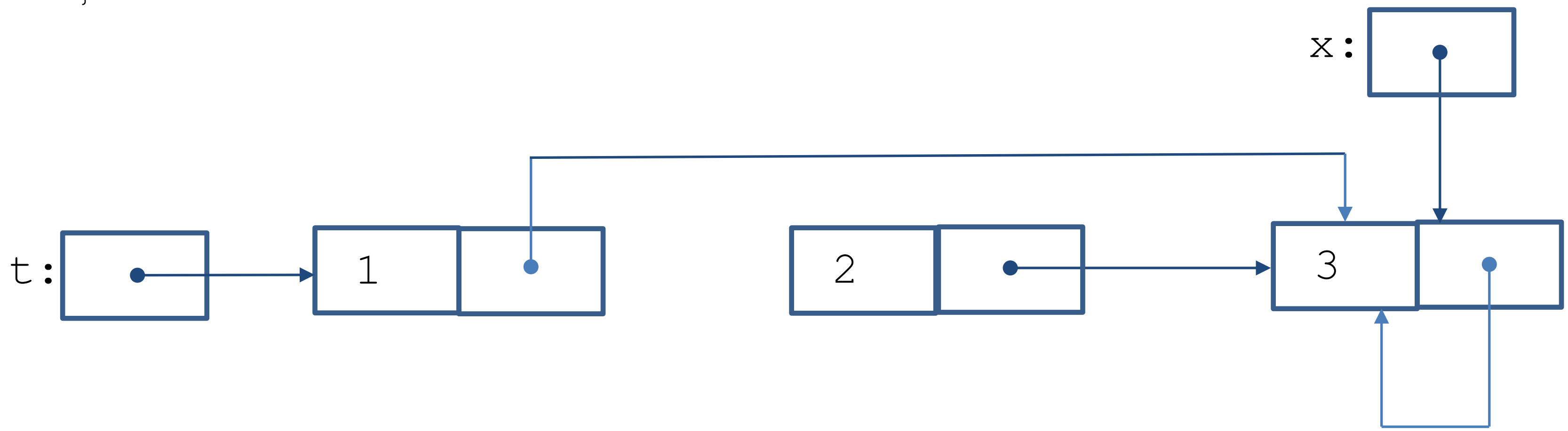
```
while (x != x->next) {
      for (i = 1; i < M; i++) x = x->next;
      x->next = x->next->next; N--;
  }
```



- The first execution of the above `while` loop on the circular list of the previous slide will have the above circular list as a result. In other words, `2` is eliminated.

Running the Program (cont'd)

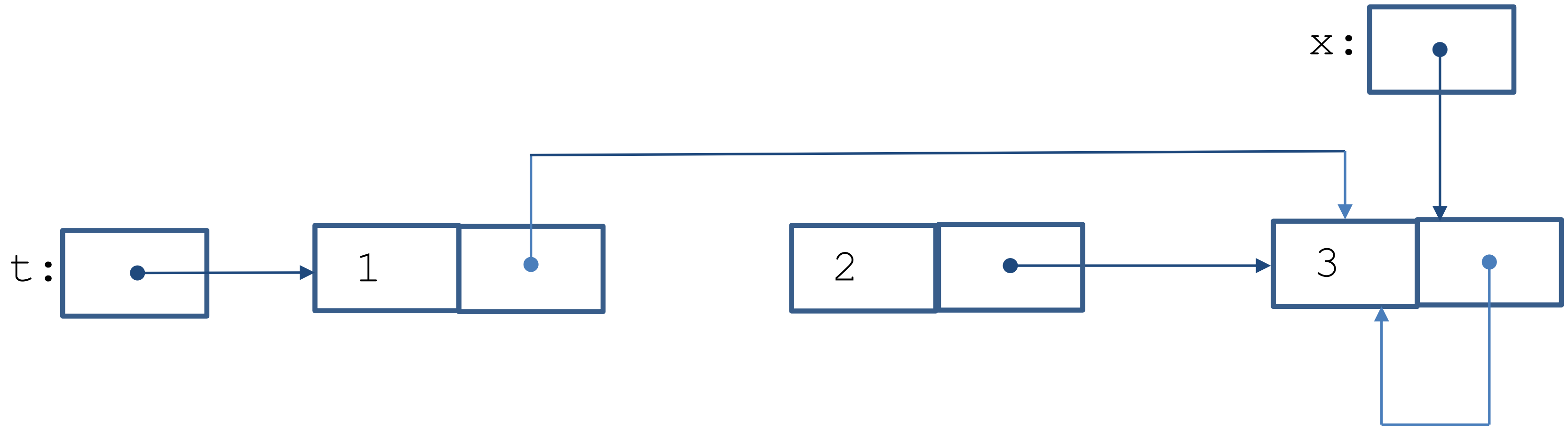
```
while (x != x->next) {
        for (i = 1; i < M; i++) x = x->next;
        x->next = x->next->next; N--;
     }
```



- The second execution of the above `while` loop on the circular list of the previous slide will have the above circular list as a result. In other words, `1` is eliminated.

Running the Program (cont'd)

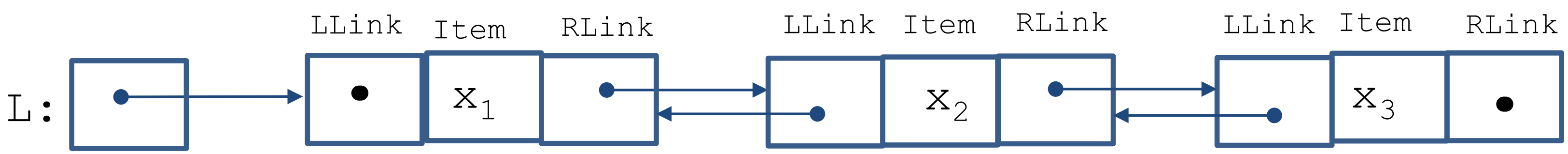
```
printf("%d\n", x->item);
```



- Then the program exits the while loop and prints the leader `3`.

Two-Way Linked Lists

- Two-way linked lists are formed from nodes that have pointers to both their right and left neighbors on the list.

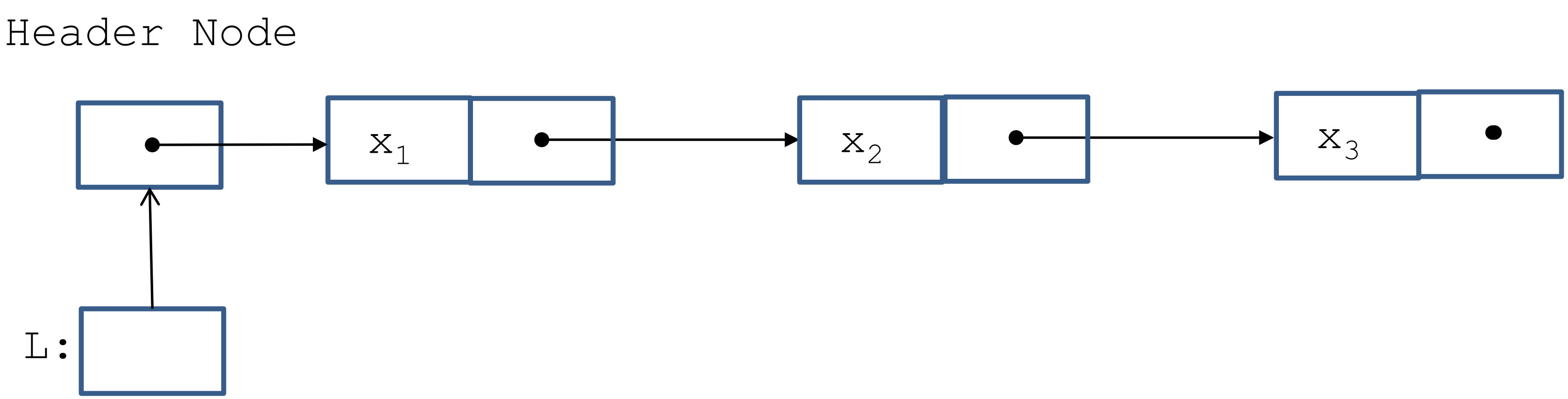


Two-Way Linked Lists (cont'd)

- Given a pointer to a node *N* in a two-way linked list, we can follow links in either direction to access other nodes.

Linked Lists with Header Nodes

- Sometimes it is convenient to have a special **header node** that points to the first node in a linked list of item nodes.



Linked Lists with Header Nodes (cont'd)

- Header nodes can be used to hold information such as the number of nodes in the list etc.
- We can have a **rear node** too.

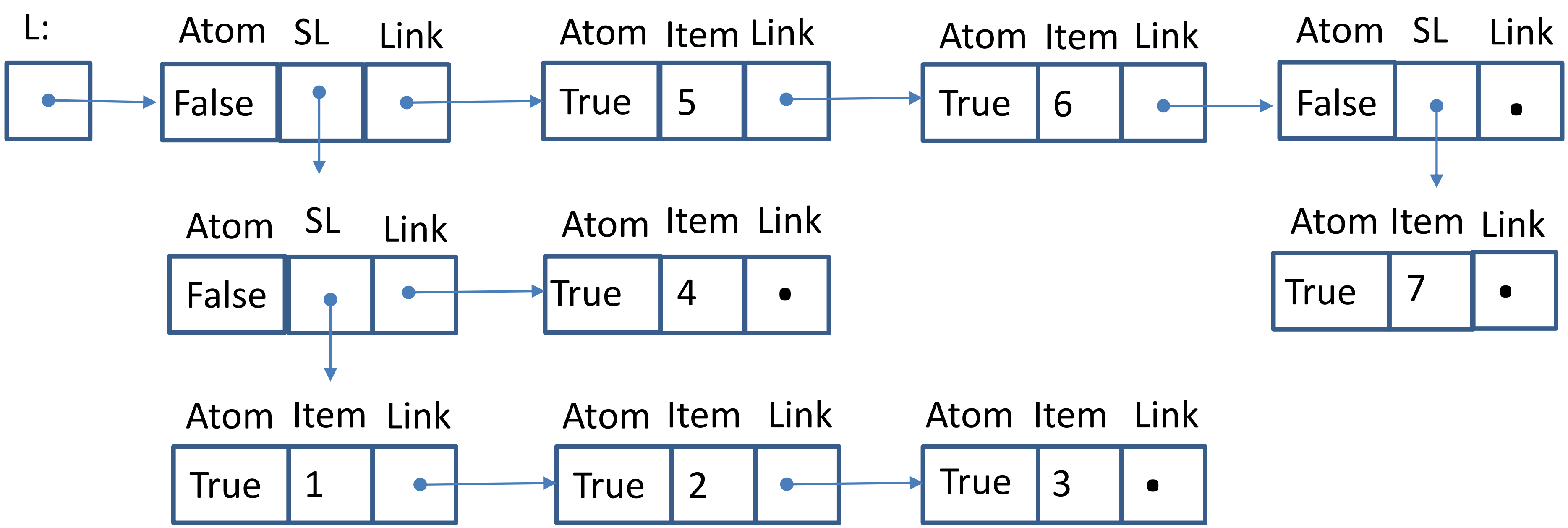
Generalized Lists

- A **generalized list** is a list in which the individual list items are permitted to be sublists.
- **Example**: $(a_1, a_2, (b_1, (c_1, c_2), b_3), a_4, (d_1, d_2), a_6)$
- If a list item is not a sublist, it is said to be **atomic**.

- Generalized lists can be represented by sequential or linked representations.

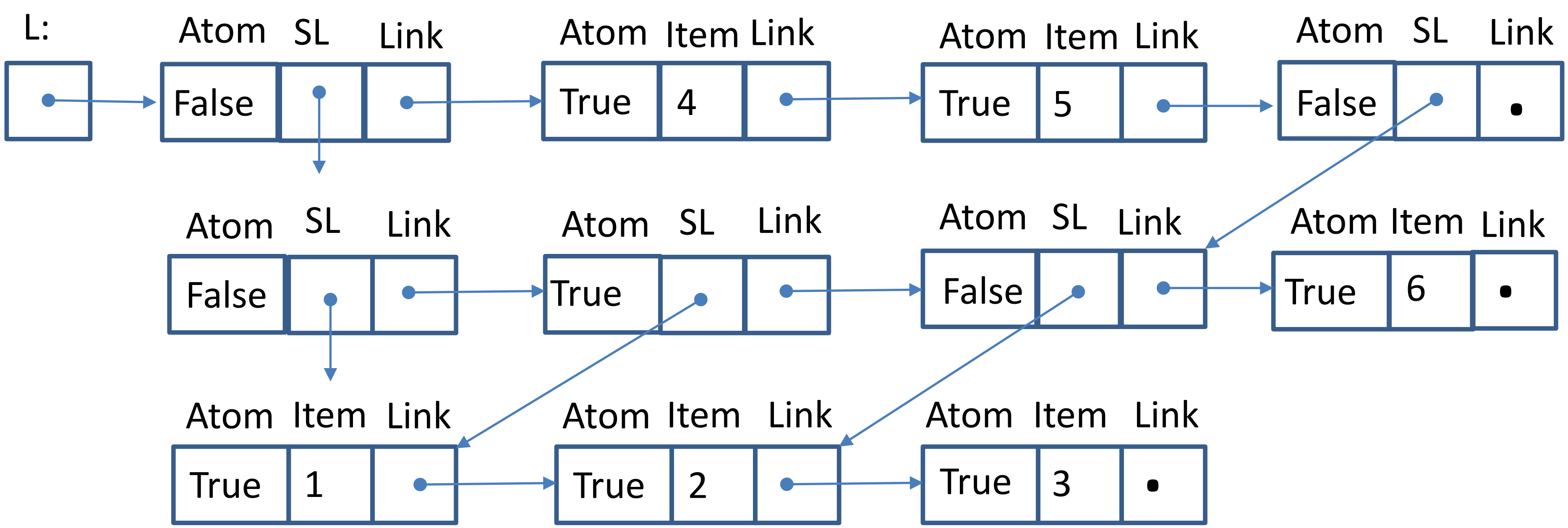
Generalized Lists (cont'd)

- The generalized list *L=(((1, 2, 3), 4), 5, 6, (7))* can be represented **without shared sublists** as follows:



Generalized Lists (cont'd)

- The generalized list *L=(((1, 2, 3), (1, 2, 3), (2, 3), 6), 4, 5, ((2, 3), 6))* can be represented with **shared sublists** as follows:



Question

- What C datatype can we use to represent a generalized list?

A Datatype for Generalized List Nodes

```
typedef struct GenListTag {
                GenListTag *Link;
                int Atom;
                union SubNodeTag {
                     ItemType Item;
                     struct GenListTag *Sublist;
                } SubNode;
       } GenListNode;
```

Unions in C

- A **union** is a variable which can hold objects of different types and sizes.
- Unions provide a way to manipulate **different kinds of data in a single area of storage.**
- The storage allocated to a union variable is enough to hold the largest of its members.
- Syntactically, members of a union are accessed as `union-name.member` or `union-pointer->member`.

Printing Generalized Lists

```
void PrintList(GenListNode *L)
{
      GenListNode *G;

      printf("(");
      G=L;
      while (G != NULL){
              if (G->Atom){
                 printf("%d", G->SubNode.Item);
              } else {
                 printList(G->SubNode.SubList);
              }
              if (G->Link != NULL) printf(",");
              G=G->Link;
      }
      printf(")");
}
```

Applications of Generalized Lists

- The historical Artificial Intelligence programming languages **LISP** and **Prolog** offer generalized lists as a language construct.
- Generalized lists are often used in Artificial Intelligence applications.
- **Python** also offers generalized lists as a language construct.
- More in the courses “Artificial Intelligence” and “Logic Programming”.

Readings

- T. A. Standish. *Data Structures, Algorithms and Software Principles in C*.

  Chapter 8, Sections 8.1-8.4.
- Robert Sedgewick. Αλγόριθμοι σε C.

  Κεφ. 3.